Performance of the iPSC/860 Node Architecture

Steven A. Moyer

IPC-TR-91-007
May 17, 1991

Performance of the iPSC/860 Node Architecture

Steven A. Moyer

Institute for Parallel Computation
School of Engineering and Applied Science
University of Virginia
Charlottesville, Virginia 22903
(sam2y@virginia.edu)

Abstract

Intel's iPSC/860 hypercube is the latest in a series of message-passing multicomputers. The performance of individual iPSC/860 computational nodes is the focus of this report; in particular, the performance of basic computational kernels common in scientific computing is examined. Understanding the operation of the iPSC/860 node memory system is key to achieving maximum node performance; from a comprehensive study of the processor-memory interrelationship, guidelines are established for implementing operations in a manner consistent with the processor architecture and memory system performance characteristics. It is demonstrated that the iPSC/860 node architecture exhibits a basic imbalance between processor speed and memory system bandwidth; due to this imbalance, even for highly optimized hand-coded routines the average performance of basic computational kernels can be as much as an order of magnitude below peak processor rate.

This work was supported in part by NASA under grant NAG-1-242.

1.0 Introduction

The iPSC/860 is the latest in a series of parallel computers produced by Intel. Like its predecessors, the iPSC/860 is a distributed memory message-passing multicomputer connected in a hypercube topology; the system can be expanded to a maximum of 128 nodes.

Each computational node of the iPSC/860 system, referred to as an RX-1, consists of an Intel i860 microprocessor, a memory subsystem, and communication hardware. The iPSC/860 system incorporates the same communication components as the iPSC/2; this interconnection network has been the subject of numerous studies and its performance parameters are well known [2][6].

The performance of individual RX-1 computational nodes is the focus of this report; in particular, the performance of basic computational kernels common in scientific computing is examined. Section 2.0 provides a brief overview of the i860 microprocessor. A comprehensive performance study of the RX-1 memory system is presented in section 3.0. Sections 4.0 and 5.0 examine the performance of vector operations and sparse matrix-vector multiplies, respectively. Section 6.0 summarizes these results.

2.0 i860 Architectural Overview

This section briefly examines features of the i860 architecture that can be exploited to increase the performance of inner-loop computations; it is not intended to provide a comprehensive architectural description. A complete functional overview of the i860 microprocessor can be found in [4][5].

2.1 General

The i860 is a 64-bit general purpose microprocessor implemented using RISC techniques; it incorporates on a single chip:

- Core integer/control unit
- Floating-point unit
- Graphics unit
- Memory management unit for protected, paged, virtual memory
- Data, instruction, and page translation caches

Performance is obtained using a combination of wide data paths, on chip data and instruction caches, pipelined floating-point arithmetic units and bus controller, and instruction parallelism.

2.2 Data Types

Hardware support is provided for the following data types:

- 32-bit integer and ordinal values in the core integer unit; 64-bit integer values in the floating-point unit.
- 32-bit single-precision and 64-bit double-precision real values in the floating-point unit.
- 8-, 16-, 32-bit pixel values in the floating-point unit.

2.3 Data Paths

Three separate data buses provide data and instruction transport:

- 64-*bit external data bus* - wide enough to transport up to 2 instructions or 1 double-precision number.
- 128-*bit internal data bus* - allows up to 2 double-precision numbers to be fetched from the data cache with a single instruction, given proper data alignment; this can decrease the number of inner-loop instructions and, in cases where data can be reused, increase the effective memory bandwidth.
- 64-*bit internal instruction bus* - allows two instructions to be fetched from the instruction cache simultaneously; this is required to support the i860's dual-instruction mode as described in 2.6.

2.4 Caches

The i860 contains three separate on-chip caches:

- 8 Kbyte *data cache* - 2-way set-associative cache with 32-byte line size; cache lines are aligned on a 32-byte boundary. Cache line fills occur in four 64-bit memory accesses using a wrap-around technique. The processor reads first the 64-bit entry containing the data object needed to fulfill the memory request which caused the cache miss; read requests are then initiated for each of the three remaining 64-bit entries in the cache line, sequentially by address. If the first entry read falls in the middle of the cache line, the read addresses wrap around.

  The data cache is implemented with a write-back policy to reduce the demand on the external data bus and memory system.
- 4 Kbyte *instruction cache* - 2-way set-associative cache with 32-byte line size. The instruction cache is large enough to allow most inner-loop code to execute without the delay of instruction loads; memory bandwidth is reserved for data access.
- 64-*entry page translation cache (TLB)* - 4-way set-associate cache for performing page translation and access violation checking; up to 256 Kbytes of virtual memory can be referenced at a time.

All caches operate in parallel and can be accessed once every clock cycle; this allows a simultaneous instruction cache access, read/write data cache access, and virtual address translation through the TLB.

Under normal operating conditions a random cache line replacement policy is implemented. Data cache control can be performed by specifying for replacement one of the two cache line blocks in a set. While this cache control mechanism can be used effectively to improve performance, it is only available in supervisor mode; this issue is discussed further in later sections.

2.5 Pipelining

The i860 architecture makes extensive use of pipelining to achieve higher rates of throughput. The floating-point unit has 4 distinct pipelines:

- *Adder* - 3-stage pipe at 1 clock per stage.
- *Multiplier* - 3-stage pipe at 1 clock per stage for single precision; 2-stage pipe at 2 clocks per stage for double-precision.
- *Graphics* - 1-stage pipe at 1 clock per stage.
- *Load* - 3-stage pipe; the rate of pipe advancement is determined by the rate at which memory requests are satisfied. The load pipe functions as a FIFO buffer for the pipelined version of the floating-point load instruction discussed in 3.2.1.

The first three functional units listed above incorporate standard pipelining techniques to increase throughput. The i860 does not provide any vector instructions; pipeline control and advancement is handled explicitly via special pipeline instructions which specify a pair of operand registers to be used as inputs to the first stage of a given pipeline, and a destination register into which is placed the result of the last stage of that pipeline. Advancement of a pipeline occurs only during execution of a pipeline instruction. For example, the code of Figure 1 utilizes the adder pipeline to form three sums from three operand pairs. Inner-loop code which makes use of floating-point pipelines must fill the pipelines before entering the loop and drain the pipelines upon exiting. A scalar mode is available for programming convenience, though at greatly reduced performance.

Bus cycles can also be pipelined to mask memory latency and increase the rate at which memory requests are satisfied. The bus state machine defines a two level pipeline protocol whereby a bus cycle can be initiated every other clock cycle with a maximum of three bus

cycles outstanding. The pipelining of memory cycles is optional and may not be supported by all memory systems.

Figure 1 Using the Floating-Point Addition Pipeline

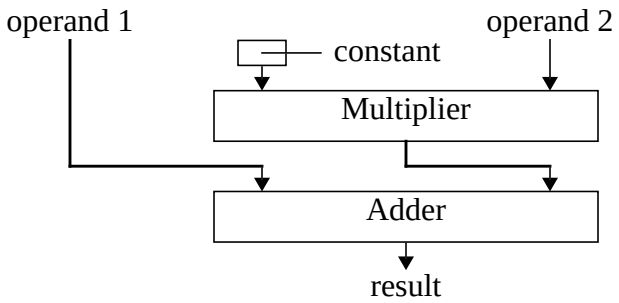
```
// Operands in fp registers 2-12
// Results in fp registers 20-24
//
// Fill adder pipe
   pfadd.dd f2,f4,f0            // f2+f4    : discard last stage
   pfadd.dd f6,f8,f0            // f6+f8    : discard last stage
   pfadd.dd f10,f12,f0          // f10+f12  : discard last stage

// Drain pipe to obtain results of last three additions
   pfadd.dd f0,f0,f20           // f20 = f2+f4
   pfadd.dd f0,f0,f22           // f22 = f6+f8
   pfadd.dd f0,f0,f24           // f24 = f10+f12
```

2.6 Parallelism

There are two forms of parallelism in the i860 that can be directly specified by the programmer: *dual-operation instructions* and *dual-instruction mode*.

Dual-operation instructions are the simplest form of parallelism in which a single instruction initiates both a pipelined floating-point adder and multiplier operation; the instruction specifies which of a number of possible data paths surrounding the adder and multiplier is to be used. For example, the instruction *r2p1* specifies that a source operand and a constant register are to be used as inputs to the first stage of the multiplier pipeline, the other source operand and the result from the last stage of the multiplier pipeline are to be used as inputs to the adder pipeline, and the result from the last stage of the adder pipeline is to be stored in the destination register. The data flow for the *r2p1* instruction is diagrammed in Figure 2; this instruction represents the standard BLAS *axpy* operation. In vector machines, this technique of linking functional units is commonly referred to as *chaining*.

Figure 2 Data Flow for R2P1 Instruction



In dual-instruction mode (DIM), the core execution unit and the floating-point unit can initiate instructions simultaneously. Utilizing the 64-bit internal instruction bus, one 32-bit

instruction can be dispatched to each unit every cycle; instruction execution is limited to lock-step operation.

2.7 Hardware Optimizations

A number of standard hardware optimizations are employed in the i860; these include:

- Delayed branching
- Register bypassing
- Scoreboarding
- Auto-increment addressing
- Write buffering

This last optimization, write buffering, involves the buffering of write operations to memory until the memory system has no outstanding requests. Write operations are forced from the buffers when either the buffers are full and another store instruction is executed, or when doing so is necessary to maintain data coherence. The i860 has two 128-bit write buffers.

2.8 Influence of the i860 Architecture on Coding Style

The i860's pipelined floating-point unit and dual-instruction mode are reflected in the coding style for operations requiring maximum performance. An effective technique for maintaining efficient floating-point pipeline operation is to combine loop unrolling with data prefetching. In unrolling inner-loop computations, the floating-point unit can operate on data for the current loop iteration while the core execution unit fetches data for the next iteration; this method of prefetching data items is also referred to as *software pipelining*. Dual-instruction mode provides a mechanism whereby floating-point and memory or control transfer operations can be initiated simultaneously; this technique is used throughout implementation examples presented in later sections.

3.0 RX-1 Memory Subsystem

Understanding the design and operation of the RX-1 memory system is key to achieving maximum node performance. It is important that inner-loop computations generate access patterns amenable to efficient memory system operation. The following examines the design of the RX-1 memory system and the factors which influence its performance.

3.1 General Specifications

The RX-1 memory system is constructed with fast page-mode DRAMs. DRAM page-mode allows multiple memory-array column accesses to be performed on a selected memory-array row, called a *page*. A memory request which hits in the current page, termed a *near* access, can be serviced at a higher rate than one which requires a memory-array row

select, termed a *far* access. Though this is a simplified description of the operation of page-mode DRAMs, it is sufficient for this discussion; detailed information can be found in [11][12]. For the duration of this paper the term *page* refers to a DRAM page, not a virtual memory page, unless otherwise stated.

The design of the DRAM controller also affects the overall performance of the memory system. For example, the RX-1 DRAM controller supports the pipelining of read requests to partially mask row select and write-read mode switch latencies. A state transition diagram describing the operation of the RX-1 DRAM controller can be found in [10]; characteristics of the controller which affect memory system performance are discussed in subsequent subsections.

The RX-1 memory system as a whole has the following specifications:

- 4 Kbyte DRAM page size
- 64-bit near reads at a maximum rate of 1 every 2 clock cycles
- 64-bit near writes at a maximum rate of 1 every 3 clock cycles
- 64-bit far reads and far writes at a maximum rate of 1 every 10 clock cycles

The exact overhead incurred in making a far access is a function of the DRAM controller state at the time a memory request is issued and the type of request being made. For the purposes of this discussion it is assumed that a DRAM page miss adds to the near access time an additional 8 clock cycles overhead; this approximation works well when deriving performance estimates.

As stated in 2.5, the bus state machine defines a minimum of 2 clock cycles per bus cycle; this places an upper limit on processor to memory bandwidth. Within this limit, the access patterns generated by a computation determine the level of memory system performance which is achieved. As demonstrated in the subsections which follow, “well behaved” operations which access memory in a manner amenable to its efficient operation can expect to achieve a level of performance close to the above stated maximums. Operations with access patterns contrary to efficient memory system operation experience substantially reduced performance.

3.2 Performance Measurements

The following presents RX-1 memory system performance measurements to demonstrate clearly those factors which affect its efficient operation. Measurements are made with the data cache flushed both to eliminate the effects of write-back, and to force all memory accesses to be satisfied directly by the memory subsystem. Data objects being accessed are double-precision floating-point numbers, the standard for scientific computation. All data is properly aligned so as not to trigger a data-access trap. Most performance measurements presented fall short of the theoretical maximums due to overheads involved in making test procedure calls, initializing test loops, and performing DRAM refresh cycles.

3.2.1 Pipelined Floating-Point Load Rate

The i860 implements two versions of the load instruction for floating-point values: pipelined floating-point load (*pfld*) and floating-point load (*fld*). The *fld* instruction is a memory to register load instruction in the traditional sense, whereas the *pfld* instruction reflects the i860's pipelined architecture. Both the functionality and performance of the *pfld* instruction are examined here; the *fld* instruction is discussed further in 3.2.4.

The *pfld* instruction has two properties which distinguish it from the standard *fld* instruction. First, the *pfld* instruction utilizes the 3-stage load pipe described in 2.5; a *pfld* returns the data referenced by the third previous *pfld* instruction executed. Second, the *pfld* instruction has no affect on the contents of the data cache. If the data requested by a *pfld* resides in the data cache, it is taken from there; however, a *pfld* does not update the cache on a cache-miss.

Memory cycles initiated by any instruction can be pipelined, within the limits discussed in 2.5, if this feature is supported by the memory system; with the exception of the *pfld* instruction, memory cycle pipelining is transparent. The *pfld* instruction semantics reflect the pipelined nature of the i860's bus state machine; the 3 stages of the load pipe correspond to the 3 outstanding bus cycles supported by the bus state machine definition.

The graph in Figure 3 demonstrates the measured *pfld* rate of a single vector for various strides of access; the theoretical maximum load rate as a function of stride is also presented. Given that there are no TLB misses and that all virtual memory pages occupied by a vector reside in physical memory, the theoretical maximum load rate is derived as follows:

s = stride of access

b = size of floating-point vector elements in bytes, either 4 or 8

n = number of clock cycles for a near access at maximum rate

f = number of clock cycles for a far access at maximum rate

o = number of clock cycles overhead incurred on a DRAM page miss

d = DRAM page size in bytes

h = clock rate in MHz

$c(s)$ = average number of clock cycles per vector element load

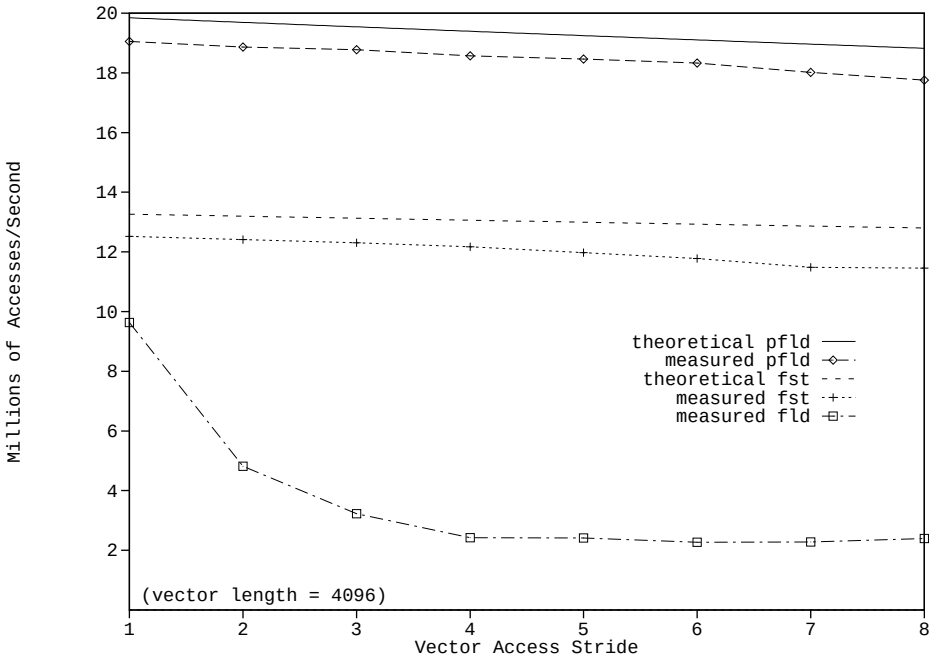
$r(s)$ = load rate in Mpfld per second

$$c(s) = \begin{cases} (n + \frac{osb}{d}) & \text{when } \frac{sb}{d} < 1 \\ f & \text{when } \frac{sb}{d} \geq 1 \end{cases} \qquad \text{(EQ 1)}$$

$$r(s) = \frac{h}{c(s)}$$

For the RX-1 memory system, $f = 10$, $o = 8$, $d = 4096$, and $h = 40$; for the *pfld* instruction $n = 2$. Since the vector elements being loaded in this example are double-precision floating-point numbers, $b = 8$.

Figure 3 Memory Access Rates



The graph in Figure 4 demonstrates the effect of TLB misses on *pfld* rate; each point on the graph represents the average *pfld* rate for multiple iterations of a single vector load at a given stride of access. In measuring the *pfld* rate of a vector load, the first iteration sets the TLB so that, given a sufficiently small stride, later iterations will not suffer a TLB miss; as the length of the stride is increased, a point is reached where all virtual-memory pages containing elements of the vector can no longer be referenced by the TLB, thrashing ensues. The approximate TLB thrash point can be determined as follows:

s^* = maximum stride of access without TLB thrashing

b = size of floating-point vector elements in bytes, either 4 or 8

l = length of vector (i.e. the number of elements)

t = maximum number of TLB entries

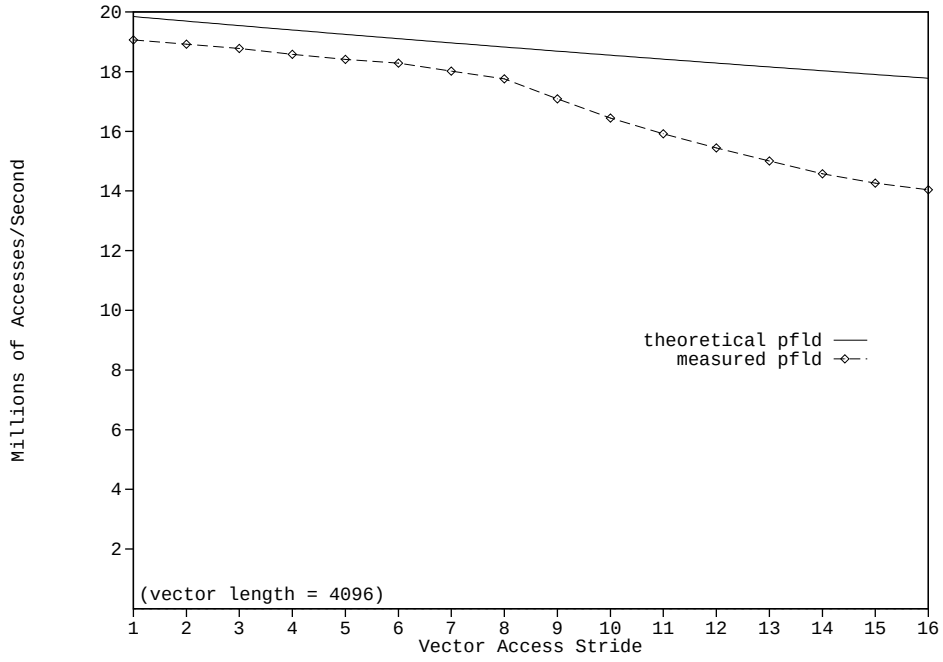
p = page-frame size in bytes

$$s^* = \begin{cases} \left(\left\lfloor \frac{pt}{lb} \right\rfloor\right) & \text{when } (l > t) \text{ and } \left(\frac{pt}{lb}\right) \geq 1 \\ \text{always thrashes} & \text{when } (l > t) \text{ and } \left(\frac{pt}{lb}\right) < 1 \\ \text{never thrashes} & \text{when } (l \leq t) \end{cases} \quad \text{(EQ 2)}$$

For the i860, $t = 64$ and $p = 4096$. In the example from Figure 4, $l = 4096$ and $b = 8$; thus, $s^* = 8$.

The equation (EQ 2) is only an approximation to the TLB thrash point since the i860 implements a random replacement policy for elements of a given TLB cache set. Note that (EQ 2) can be written to solve for l^* in terms of s to determine the maximum length vector which can be repeatedly accessed at a given stride without TLB thrashing.

Figure 4 TLB Thrash Point



For vectors of a given length, the following observations can be made:

- Increasing the stride of access increases the number of DRAM page misses which results in a decreased *pfld* rate.
- Increasing the stride of access increases the chance of a TLB miss which in turn results in a decreased *pfld* rate.

The first observation is a consequence of the fact that, with increased stride, a greater number of DRAM pages are required to store the vector. The second observation is a consequence of the vector occupying more virtual memory pages, given that the TLB can only reference a finite number of page frames.

Note that the above observations assume the vector is long enough so that an increase in stride results in an increase in the number of DRAM pages occupied; if this is not the case, there is no affect on the *pfld* rate.

3.2.2 Floating-Point Store Rate

The graph of Figure 3 demonstrates the measured floating-point store (*fst*) rate to a single vector for various strides of access; the theoretical maximum store rate as a function of stride is also presented. Given that there are no TLB misses and that all virtual memory pages occupied by the vector reside in physical memory, the theoretical maximum store rate is derived in the same fashion as the *pfld* rate of (EQ 1); for the *fst* instruction $n = 3$.

The observations made in 3.2.1 concerning degradation in performance with increasing stride of access, due to TLB and DRAM page misses, apply to the *fst* as well.

3.2.3 Effects of the DRAM Controller Idle-state Transition

It seems intuitive that the best memory system performance is achieved by minimizing the demand placed on the memory system; this, however, is not necessarily the case. When performing near reads or near writes, the best performance is achieved by issuing memory requests at the maximum rate at which they can be satisfied; not doing so can allow the DRAM controller to enter into an idle state. Once the DRAM controller has initiated a transition towards the idle state, the next memory request is handled in the same manner as a far access.

The length of the delay from the last near access to the DRAM controller idle-state transition is dependant on the access mode: 4 clock cycles for a near read and 3 clock cycles for a near write. A near access request issued anytime up to the clock cycle in which the DRAM controller is to initiate the idle-state transition prevents a far access; a penalty of approximately 3 additional clock cycles is still incurred if the time between the previous and current near access requests is longer than required to sustain the maximum access rate.

In general, when memory access delay is sufficient to allow the DRAM controller to initiate an idle-state transition, the net additional penalty to the next memory access is the 8 cycle far access overhead minus the delay between accesses; when the length of the memory access delay equals or exceeds the far access overhead, no additional penalty is incurred.

As an example, Figure 5 (a,b) lists the inner-loop codes for two procedures which perform a *pfld* on each element of a vector. Loop (a) performs 6 *pfld* operations per iteration, 4 useful data loads and 2 dummy loads which reread every fourth vector element. The code in (b) performs 4 *pfld* operations per iteration, all useful data loads. Both (a) and (b) contain the same number of instructions and both have the net effect of loading 4 data objects per iteration; therefore, both have the same theoretical useful load rate. Loop (a) initiates a *pfld* instruction every other clock cycle, the maximum rate at which the memory system can satisfy read requests; in (b) there is a delay of 5 clock cycles after every fourth read. The memory access delay in loop (b) is sufficient to allow the DRAM controller to initiate an idle-state transition; as a result, a net penalty of 3 additional clock cycles is incurred by the *pfld* instruction which executes immediately after the delay.

Table 1 lists the measured *pfld* rate, the useful *pfld* rate, and the theoretical useful *pfld* rate for each loop (a) and (b); for both performance measurements, vectors accessed are of the same length and stored with a stride of 1. While the performance of loop (a) is within 4.5% of the theoretical useful load rate, the performance of loop (b) falls short by 26.3%; however, in terms of memory accesses per unit of time, the demand placed on the memory system by loop (a) exceeds that of loop (b).

Table 1 Effects of the DRAM Controller Idle-state Transition on PFLD Rate

	Measured	Useful	Theoretical Useful
(a)	19.1	12.7	13.3
(b)	9.8	9.8	13.3

Rates are in Mpfld/Second

As the results in Table 1 demonstrate, memory access timing influences the overall performance of the memory system; careful load/store instruction placement in inner-loop code can yield an increase in effective memory bandwidth.

Figure 5 Inner Loop Codes to Test the Effects of the DRAM Controller Idle-state Transition

```
.a_loop:
// pfld next 4 data items
   nop
   pfld.d   r22(r17)++,f0
   nop
   pfld.d   r22(r17)++,f0
   nop
   pfld.d   r22(r17)++,f0
   nop
   pfld.d   r22(r17)++,f0
// dummy loads
   nop
   pfld.d   0(r17),f0
   bla      r20,r21,.a_loop
      pfld.d   0(r17),f0

            (a)
```

```
.b_loop:
// pfld next 4 data items
   pfld.d   r22(r17)++,f0
   nop
   pfld.d   r22(r17)++,f0
   nop
   pfld.d   r22(r17)++,f0
   nop
   pfld.d   r22(r17)++,f0
// delay
   nop
   nop
   nop
   bla      r20,r21,.b_loop
      nop

            (b)
```

3.2.4 Floating-Point Load Rate

Unlike the *pfld* instruction discussed in 3.2.1, the floating-point load (*fld*) instruction can have the side-effect of altering the contents of the data cache. When data requested by a *fld* does not reside in the data cache, i.e. a cache-miss occurs, a cache line fill is initiated; four 64-bit memory accesses are made using the wrap-around technique described in 2.3.

Referring again to Figure 3, the graph demonstrates the *fld* rate of a single vector for various strides of access. All else being equal, the *fld* rate at a stride of 1 should be equivalent to the *pfld* rate at the same stride; in fact, the measured *fld* rate is only half that of the measured *pfld* rate. The explanation for this discrepancy lies in the implementation of the i860 cache line fill procedure; between the fourth memory access of the previous cache line fill and the first memory access of the subsequent cache line fill there is a delay of 7 cycles [14]. As discussed in 3.2.3, this memory access delay between consecutive cache line fills is sufficient to allow the DRAM controller to initiate an idle-state transition; the net result is that the first of the 4 memory accesses in each line load is a far read. Thus, for consecutive cache line fills, each individual cache line load requires a total of 16 cycles; an average *fld* rate of 4 cycles per double-precision load is achieved.

At strides greater than 1 the *fld* rate is further degraded by extraneous data loads which occur as a result of the cache line fill procedure; for double-precision values this reduction in performance continues up to a stride of 4, from which point on 3 extraneous data loads occur for every 1 requested.

While the observation made in 3.2.1 relating stride of access and TLB misses applies to vectors accessed via the *fld* instruction, the effects are minimal and all but masked by the factors discussed above; the observation concerning DRAM page misses does not apply since consecutive cache line fills always begin with a far access.

Note that for the 7 cycles following the fourth memory access of a given cache line fill, no other memory access can be initiated.

3.2.5 Effects of DRAM Page Misses

As demonstrated in the graph of Figure 3 for *pfld* and *fst*, with a single vector and small strides the effect of DRAM pages misses on memory access performance is minimal; large strides are required in order to reduce the number of vector elements per DRAM page to where the cost of a page miss becomes a significant portion of the total time required to load all vector elements from a given page. Fortunately, since large stride accesses have traditionally led to reduced performance in vector computers, numerical algorithms are normally implemented to access data with a stride of 1 whenever possible.

In practice, the effect of DRAM page misses is most strongly felt when performing operations which require multiple vector operands; operations such as the BLAS routines *ddot* and *daxpy* are typical examples. Given an operation with multiple vector operands, it is common for an implementation of the operation to alternate reads from, and stores to, each of the appropriate vectors; for vector operands having no DRAM pages in common, a page miss occurs with each access to a vector different from the one previously accessed, performance of the operation is severely degraded.

Table 2 presents *pfld* rates for two procedures which load two vector operands each. The first procedure alternates reads from each vector operand; the second reads in sets, loading 4 elements from one vector then 4 elements from the other. To emphasize the reduction in performance resulting from DRAM page misses, the address of a single vector is passed as

both operands to each procedure. For all performance measurements presented in Table 2, vectors are of the same length and stored with a stride of 1.

Table 2 Effects of DRAM Page Misses on PFLD Rates

	Single Vector	Two Vectors
Alternating	19.1	3.9
Sets of 4	19.1	9.6

Rates are in Mpfld/Second

In implementing operations with multiple vector operands, unrolling inner-loops allows the cost of a DRAM page miss to be amortized over a number of *pfld* operations. In the example from Table 2, loop unrolling to a depth of 4 realizes an almost 250% increase in memory system performance.

Note that because consecutive cache line fills begin with a far access, as discussed in 3.2.4, the *fld* rate achieved when alternating cache line loads from multiple vector operands is the same as the *fld* rate from a single vector operand given operands of the same stride.

3.2.6 Effects of the DRAM Controller Read/Write Mode Switch

The RX-1 memory system has two main modes of operation: read and write. Switching from one access mode to the other can cause a delay as the DRAM controller makes the transition. For a near access, the mode transition overhead is approximately 4 additional cycles; a far access incurs no additional penalty, the mode transition delay is masked by the far access overhead.

While the DRAM controller mode switch delay is relatively short, it can lead to a substantial reduction in performance if read and write operations are intermixed indiscriminately; Table 3 presents performance results from two procedures which illustrate this effect by reading and writing each element of a vector. The first procedure alternates access modes, reading (*pfld*) a single vector element and immediately performing a write to that element before reading the next. The second procedure reads (*pfld*) 4 consecutive vector elements, writing back to these elements before moving on to read the next set. For both performance measures presented in Table 3, vectors are of the same length and stored with a stride of 1.

Table 3 Effects of DRAM Controller Read/Write Mode Switch on Memory Access Rates

	Millions of Accesses/Second
Alternating	6.5
Sets of 4	11.4

In grouping like memory operations, the cost of a read/write mode switch is amortized over a number of accesses; loop unrolling can increase memory system performance by providing a natural means to achieve such a grouping. In the example from Table 3, loop unrolling to a depth of 4 realizes a 175% increase in memory system performance.

The buffering of writes, as discussed in 2.7, has negligible affect on the results of this test. Since the write buffers are full after two *fst* operations have been issued, each additional *fst* forces a write bus cycle.

Note that read and write operations to vectors accessed via the *fld* instruction are naturally grouped by the cache line load/write-back procedure, as described in 2.4.

3.3 Memory Access Guidelines

The following summarizes the performance characteristics of the RX-1 memory sub-system; memory access guidelines are provided for implementing operations to access memory in a manner amendable to its efficient operation. Due to the nature of the cache line fill procedure, few of the memory system characteristics discussed throughout 3.2 affect *fld* performance; the guidelines presented below assume vector operands being accessed are loaded via the *pfld* instruction. The caching of vector operands is discussed further in 4.1.

The following guidelines are made under the implicit assumption that the data being accessed is not initially resident in the data cache:

- Vectors should be accessed with small strides to reduce the effects of DRAM page and TLB misses.
- Operations which require multiple vector operands should avoid accessing them alternately. The cost of a DRAM page miss, which is likely to occur with each access to a vector operand different from the one previously accessed, should be amortized over a number of *pfld* operations; loop unrolling can be used effectively to achieve this amortization.
- Like memory operations should be grouped to amortize the cost of a read/write mode switch; loop unrolling provides a natural means to achieve such a grouping.
- Vectors should be accessed at the maximum rate at which requests can be satisfied, not doing so may allow the DRAM controller to initiate an idle-state transition; extraneous data accesses can be used to prevent an idle-state transition, avoiding an unnecessary far access penalty.

As demonstrated in subsequent sections, an implementation of an operation which adheres to these guidelines exhibits substantially increased performance over an implementation of the same operation which does not.

4.0 Vector Operations

Vector operations represent the most basic computational kernels common in scientific computing; they are the building blocks of most numerical methods. This section examines the implementation and performance of vector operations on the RX-1 architecture; two are examined in detail, the BLAS routine *daxpy* and an operation which we will refer to as *vaxpy.*

To distinguish between various implementations of a vector operation, the following notation is adopted throughout this discussion:

<*operation name*>{_<*cached operand*>{*}{<*cached operand*>{*}}...}{_<*version*>}

where the arguments enclosed in '{}' are optional. In the naming scheme above, an '*' following the identifier for a cached vector operand specifies that the implementation assumes the operand to be aligned on either a 16- or 32-byte boundary; a version number is only appended when required to make the name unique. For example, a daxpy implementation which does not cache either vector operand is referred to as 'daxpy'; one which caches both vector operands is referred to as 'daxpy_xy'.

4.1 Caching Vector Operands

One of the major factors which affects the performance of a vector operation is the choice of operands, if any, to cache. Caching a vector operand has both positive and negative aspects which need to be considered in making this decision.

If a vector operand is to be accessed again in a later operation, a number of benefits are derived from caching:

- Effective memory bandwidth is increased in later operations by taking advantage of both the speed of the data cache and the width of the internal data bus. The data cache can be accessed once every clock cycle; data which is properly aligned can be accessed in 128-bit blocks.
- Overall memory system performance is improved by reducing the number of DRAM page misses and DRAM controller read/write mode switches; memory accesses to a cached operand, regardless of the access pattern, hit in the data cache and therefore have no potential to generate either effect.
- As demonstrated in 3.2.5, for an operation with multiple vector operands stored at a stride of 1 and having no DRAM pages in common, the initial *fld* rate for accessing the operands alternately is equal to the *pfld* rate when accessing them in sets of 4; inner-loop coding is simplified while a reasonable degree of performance is maintained.

Caching a vector operand has negative aspects which must also be considered:

- As discussed in 3.2.4, for a single vector of stride 1 the initial *fld* rate is only half that of the *pfld* rate; the effective *fld* rate is considerably less at larger strides. For caching to be of any benefit, an operand must be accessed often enough to compensate for the initially reduced load rate; e.g. at least twice for vectors with a stride of 1.
- The operand being cached displaces the current data cache contents and may generate cache line write-backs. Write-backs utilize memory bandwidth, further reducing the initial *fld* rate.
- If the operand being cached has a stride greater than 1, then extraneous data loads occur as a result of the cache line load procedure. Extraneous data can occupy as much as three-fourths of the total data cache capacity.
- As is standard for processors with a set-associative data cache, i860 memory accesses are mapped to data cache sets via the low order address bits. Given the i860's 128 data cache sets and a cache line size of 4 double-precision values, a stride of access s such that $s = 2^q$ and $8 \leq s \leq 512$ reduces the effective cache size to $1/2^{q-2}$ maximum capacity; the potential for data cache thrashing is increased accordingly.

The decision concerning which vector operands to cache depends on the context in which the vector operation is to be used. Only those operands of a vector operation which will be accessed repeatedly before being displaced should be considered as candidates for caching. Often it is best to implement multiple versions of the same vector operation which cache different operands, allowing one to choose the version most suited to the context in which it will be used. Because of the reduced effective load rate, inefficient use of data cache space, and increased potential for data cache thrashing it is generally recommended that vector operands with stride greater than 1 not be cached. For the remainder of this paper, all references to cached vector operands assume the vector to be stored with a stride of 1.

As stated in 2.4, in user mode the i860's 2-way set-associative data cache implements a random cache line replacement policy. Under a random replacement policy, vector operations with a single cached operand can fill only half the data cache before earlier accessed vector elements become subject to possible displacement by later accessed elements of the vector; vector operations with multiple cached operands have no control over data cache thrashing. In the i860's supervisor mode, data cache control can be performed by specifying for replacement one of the two cache line blocks in a set. This cache control mechanism allows up to two vector operands to reside in the data cache simultaneously; vector operations with more than 2 cached operands will still experience data cache thrashing.

4.2 Vector Operation Performance Measurements

Two measures of performance are defined for use in this discussion: *static* performance and *asymptotic* performance. Static performance is the measure of the performance given a flushed data cache; i.e. at the initiation of the operation no data which is to be accessed resides in the data cache and no cache lines require write-back. Asymptotic performance is the average performance of a large number of consecutive applications of an operation to

the same operands. In measuring asymptotic performance, the content of the data cache is unaltered between successive iterations.

For a given implementation of a vector operation which caches a subset of its operands, the static performance measurement represents the performance of an initial iteration of the operation for which none of its operands reside in the data cache; the asymptotic performance measurement represents the theoretical maximum performance of subsequent iterations for the same cached operands. In performing actual computations, asymptotic performance can be achieved when a problem is strip-mined in such a way that inner-loop computations reuse cached operands for some number of consecutive iterations.

Achieving theoretical maximum asymptotic performance requires eliminating data cache thrashing when the total number of vector operand elements cached does not exceed data cache capacity; in this case, the data cache control mechanism described in 4.1 can eliminate thrashing for up to 2 cached vectors. Because supervisor mode is not available to general users of the iPSC/860 system, performance measurements presented throughout this paper reflect the effects of the random cache line replacement policy. Under a random replacement policy, the measured asymptotic performance of a vector operation with at most two cached operands converges to the theoretical maximum asymptotic performance; given a sufficient number of iterations, all pairs of cache lines which map to the same data cache set will be placed in different blocks and thrashing will cease. For problems which employ strip-mining to promote reuse of cached operands, the random cache line replacement policy prevents inner-loop computations from achieving the theoretical maximum asymptotic performance in early iterations; as a result, the effective computation rate is generally well below that achievable when utilizing the data cache control mechanism. The degradation in performance which results from strip-mining computations without data cache control is demonstrated in 5.1 for a diagonally sparse matrix-vector multiply operation.

In measuring static performance, data cache control only affects vector operations which modify a cached operand; for these operations, static performance can be improved by employing the data cache control mechanism to prevent cache line write-back prior to reaching data cache capacity.

For vector operations which do not cache operands, asymptotic performance is equivalent to static performance; only static performance measurements are presented. Both static and asymptotic performance measurements are given in terms of millions of floating-point operations per second (*Mflops*).

4.3 DAXPY Operation

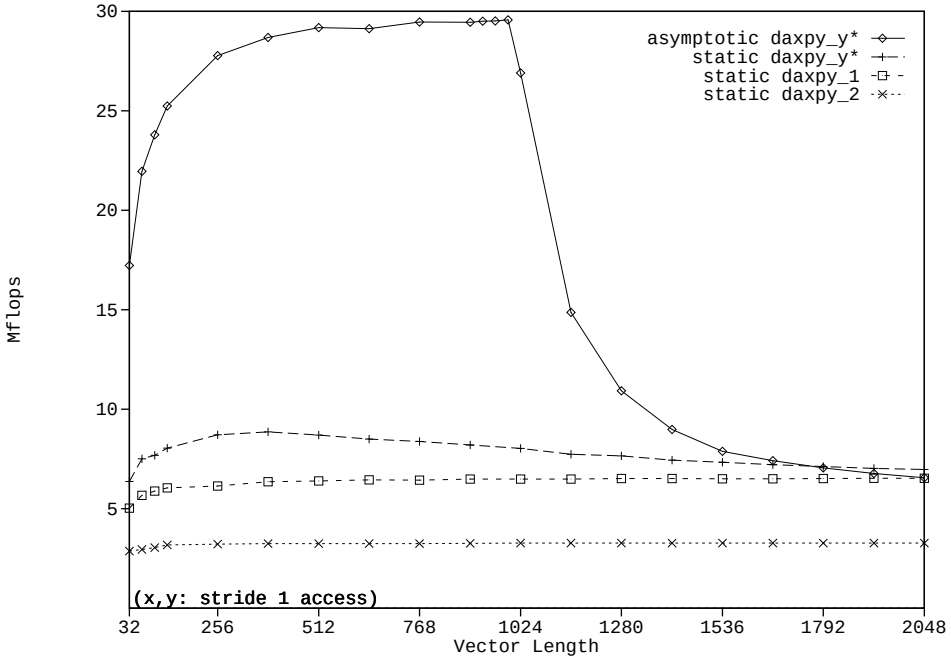
The BLAS routine *daxpy* [7] implements a double-precision vector operation of the form:

$$\mathbf{y} \leftarrow a\mathbf{x} + \mathbf{y}$$

where **x** and **y** are vectors and *a* is a scalar. The graph of Figure 6 depicts the asymptotic and static performance of the vector operation daxpy_y* and the static performance of the vector operations daxpy_1 and daxpy_2. Recall from the notation definition that the daxpy_y* implementation caches the **y** vector; daxpy_1 and daxpy_2 are different implementations, neither of which caches either vector operand. For all performance measures, vector operands are stored with a stride of 1.

Given that in pipelined mode the i860 can produce one double-precision multiply result every 2 clock cycles, as described in 2.5, the upper-bound on processor performance for the daxpy operation is 40 Mflops at a clock rate of 40 MHz. Due to the insufficient bandwidth of the RX-1 memory system, the measured performance of a daxpy operation which accesses a vector operand not resident in the data cache will be below peak processor rate. For the daxpy implementations of Figure 6, specific RX-1 memory system characteristics which affect daxpy performance are discussed in later subsections.

Figure 6 Performance of Various DAXPY Implementations



4.3.1 DAXPY Performance Curves

The shape of the curves for the daxpy_y* implementation of the daxpy operation are characteristic of operations which cache one or more vector operands. For asymptotic performance, the curve clearly shows the point at which the total number of operand elements cached in performing an operation exceeds data cache capacity; performance degrades rapidly as cache-misses, and potentially write-backs, begin to have a substantial effect. Given the i860's 8 Kbyte data cache and 8-byte double-precision values, cache capacity is reached at a vector length of 1024 with one cached operand or 512 with two. On the asymptotic daxpy_y* curve of Figure 6, the point plotted at vector length 1024 exhibits somewhat degraded performance even though the vector length appears to be within data cache capacity; this effect is a result of the cached vector's alignment. If a double-precision vector of length l such that (l mod 4)=0 is not aligned on a 32-byte boundary, then it will span one more than the minimum number of lines required to cache it.

Asymptotic performance degrades to the level of static performance as vector lengths increase; earlier accessed vector elements are displaced by later accessed vector elements and the percentage of data reuse from one iteration of the vector operation to the next decreases. The asymptotic and static curves tend to come together in the area of twice data cache capacity, the point at which a subsequent iteration of the operation no longer benefits from the data residing in the cache. For vector operations which modify cached operands, asymptotic performance drops somewhat below the level of static performance; this effect results from a combination of low data reuse between consecutive iterations of the operation, and the initiation of consecutive iterations with the data cache in a state where cache lines require write-back.

Static performance curves for operations which modify cached operands, while generally flat, exhibit somewhat better performance prior to reaching data cache capacity due to the lower probability of cache line write-backs.

For static measurements and asymptotic measurements prior to reaching data cache capacity, the performance of vector operations is generally lower for shorter vector lengths due to the fixed costs involved in making a procedure call, initializing loops and filling pipelines; as vector lengths increase, these costs are amortized over a larger number of arithmetic operations. In addition, vector operations which incorporate loop unrolling often require a minimum vector length before arithmetic operations are pipelined; below this minimum arithmetic operations are performed in scalar mode.

4.3.2 DAXPY_1

The inner-loop code for the daxpy_1 implementation of the daxpy operation is listed in Figure 7; each set of DIM instruction pairs is labeled for reference. The daxpy_1 inner-loop is unrolled to a depth of 4; each iteration performs pipelined arithmetic operations on the 4 sets of $\{x_i,y_i\}$ operand pairs loaded by the previous iteration while simultaneously prefetching the next 4 sets of operand pairs for the next iteration. All memory references in the daxpy_1 implementation conform to the memory access guidelines of 3.3; accesses to each vector operand are performed in sets, like memory operations are grouped and all

accesses are performed at the maximum rate at which they can be satisfied by the memory system.

Based on the memory system performance data of 3.2, it is possible to analyze the daxpy_1 code and derive an approximation for its execution rate; the assumption is made that vector operands have no DRAM pages in common. Beginning with line 4, there are 4 *fst* operations to the **y** vector performed at the maximum rate of 1 every 3 cycles; the store instruction at line 4 incurs a far access penalty of 8 additional cycles for accessing a vector different from the one previously accessed in line 2. Lines 16-22 perform 4 *pfld* operations from the **y** vector at the maximum rate of 1 every 2 cycles; the load instruction at line 16 incurs a DRAM controller read/write mode switch penalty of 4 additional cycles as a result of reading from the same vector written to by the previous memory access instruction at line 13. Finally, lines 24-2 perform 4 *pfld* operations from the **x** vector at the maximum rate of 1 every 2 cycles; the load instruction at line 24 incurs a far access penalty of 8 additional cycles for accessing a vector different from the one previously accessed in line 22.

Summing together access times, far access penalties and read/write mode switch penalties yields a total of 48 cycles per daxpy_1 inner-loop iteration; at 8 floating-point operations per 48 cycles, this is an estimated computation rate of 6.7 Mflops at 40 MHz. The estimated computation rate for the daxpy_1 operation compares favorably to the measured rates found in the graph of Figure 6.

Figure 7 DAXPY_1 Inner Loop Code

```
// (prev) - refers to a value associated with the previous iteration
// prime(‘) - refers to a value associated with the next iteration

.inner_loop:

1)  d.pfmul.dd   f12,f30,f2                     // a * x3 : ax1
    nop
2)  d.pfadd.dd   f18,f2,f10                     // y1 + ax1 : s3 (prev)
    pfld.d       r20(r24)++,f24                 // x’0 & load x’3
3)  d.fnop                                      // pause for data
    nop
4)  d.fnop
    fst.d        f4,r20(r26)++                  // store s0 (prev)
5)  d.fnop                                      // pause for data
    nop
6)  d.fnop                                      // pause for data
    nop
7)  d.fnop
    fst.d        f6,r20(r26)++                  // store s1 (prev)
8)  d.fnop                                      // pause for data
    nop
9)  d.fnop                                      // pause for data
    nop
10) d.fnop
    fst.d        f8,r20(r26)++                  // store s2 (prev)
11) d.fnop                                      // pause for data
    nop
12) d.fnop                                      // pause for data
    nop
13) d.fnop
    fst.d        f10,r20(r26)++                 // store s3 (prev)
14) d.fnop                                      // pause for data
    nop
15) d.fnop                                      // pause for data
    nop
```

```
16) d.fnop
    pfld.d       r20(r25)++,f26                  // x'1 & load y'0
17) d.fnop                                       // pause for data
    nop
18) d.fnop
    pfld.d       r20(r25)++,f28                  // x'2 & load y'1
19) d.fnop                                       // pause for data
    nop
20) d.fnop
    pfld.d       r20(r25)++,f30                  // x'3 & load y'2
21) d.fnop                                       //pause for data
    nop
22) d.pfmul.dd   f12,f24,f2                      // a * x'0 : ax2
    pfld.d       r20(r25)++,f16                  // y'0 & load y'3
23) d.pfadd.dd   f20,f2,f0                       // y2 + ax2 : 0
    nop
24) d.pfmul.dd   f12,f26,f2                      // a * x'1 : ax3
    pfld.d       r20(r24)++,f18                  // y'1 & load x''0
25) d.pfadd.dd   f22,f2,f4                       // y3 + ax3 : s0
    nop
26) d.pfadd.dd   f0,f0,f6                        // push : s1
    pfld.d       r20(r24)++,f20                  // y'2 & load x''1
27) d.pfmul.dd   f12,f28,f2                      // a * x'2 : ax'0
    bla          r17,r23,.inner_loop             // branch to top of loop
28)  d.pfadd.dd  f16,f2,f8                       // y'0 + ax'0: s2
     pfld.d      r20(r24)++,f22                  // y'3 & load x''2
```

4.3.3 DAXPY_2

Figure 8 lists the inner-loop code for the daxpy_2 implementation of the daxpy operation. Like the daxpy_1 implementation, daxpy_2 is unrolled to a depth of 4 and performs pipelined operations and data prefetching. However, memory references in the daxpy_2 implementation do not conform to the memory access guidelines of 3.3; though like memory accesses are grouped and performed at the maximum rate at which they can be satisfied, *pfld* requests from the **x** and **y** vectors are alternated.

Referring to Figure 8, and again assuming that vector operands have no DRAM pages in common, a performance estimate for the daxpy_2 implementation can be derived. Beginning with line 4, there are 4 *fst* operations to the **y** vector performed at the maximum rate of 1 every 3 cycles; the store instruction at line 4 incurs a DRAM controller read/write mode switch penalty of 4 additional cycles as a result of storing to the same vector read by the previous memory access instruction of line 2. Lines 16-2 perform *pfld* operations alternately from the **x** and **y** vectors for a total of 8 loads; all loads incur a far access penalty, requiring a total of 10 cycles each.

Summing together access times and penalties yields a total of 96 cycles per daxpy_2 inner-loop iteration; at 8 floating-point operations per 96 cycles, this is an estimated com-

putation rate of 3.3 Mflops at 40 MHz. The estimated computation rate of the daxpy_2 operation compares favorably to the measured rates found in Figure 6.

Figure 8 DAXPY_2 Inner Loop Code

```
// (prev) - refers to a value associated with the previous iteration
// prime(‘) - refers to a value associated with the next iteration

.inner_loop:

1)  d.pfmul.dd   f12,f30,f2                     // a * x3 : ax1
    nop
2)  d.pfadd.dd   f18,f2,f10                     // y1 + ax1 : s3 (prev)
    pfld.d       r20(r25)++,f24                 // x’0 & load y’1
3)  d.fnop                                      // pause for data
    nop
4)  d.fnop
    fst.d        f4,r20(r26)++                  // store s0 (prev)
5)  d.fnop                                      // pause for data
    nop
6)  d.fnop                                      // pause for data
    nop
7)  d.fnop
    fst.d        f6,r20(r26)++                  // store s1 (prev)
8)  d.fnop                                      // pause for data
    nop
9)  d.fnop                                      // pause for data
    nop
10) d.fnop
    fst.d        f8,r20(r26)++                  // store s2 (prev)
11) d.fnop                                      // pause for data
    nop
12) d.fnop                                      // pause for data
    nop
13) d.fnop
    fst.d        f10,r20(r26)++                 // store s3 (prev)
14) d.fnop                                      // pause for data
    nop
15) d.fnop                                      // pause for data
    nop
16) d.fnop
    pfld.d       r20(r24)++,f16                 // y’0 & load x’2
17) d.fnop                                      // pause for data
    nop
18) d.fnop
    pfld.d       r20(r25)++,f26                 // x’1 & load y’2
19) d.fnop                                      // pause for data
    nop
20) d.fnop
    pfld.d       r20(r24)++,f18                 // y’1 & load x’3
21) d.fnop                                      //pause for data
    nop
22) d.pfmul.dd   f12,f24,f2                     // a * x’0 : ax2
    pfld.d       r20(r25)++,f28                 // x’2 & load y’3
23) d.pfadd.dd   f20,f2,f0                      // y2 + ax2 : 0
    nop
24) d.pfmul.dd   f12,f26,f2                     // a * x’1 : ax3
    pfld.d       r20(r24)++,f20                 // y’2 & load x’’0
25) d.pfadd.dd   f22,f2,f4                      // y3 + ax3 : s0
    nop
26) d.pfadd.dd   f0,f0,f6                       // push : s1
    pfld.d       r20(r25)++,f30                 // x’3 & load y’’0
27) d.pfmul.dd   f12,f28,f2                     // a * x’2 : ax’0
    bla          r17,r23,.inner_loop            // branch to top of loop
28)  d.pfadd.dd  f16,f2,f8                      // y’0 + ax’0: s2
     pfld.d      r20(r24)++,f22                 // y’3 & load x’’1
```

4.3.4 DAXPY_Y*

Figure 9 lists the inner-loop code for the daxpy_y* implementation of the daxpy operation. In contrast to daxpy_1 and daxpy_2, the daxpy_y* implementation caches the **y** operand; it is assumed that this operand is aligned on either a 16- or 32-byte boundary, allowing two double-precision values to be loaded with a single *fld* instruction. Like daxpy_1 and daxpy_2, the daxpy_y* implementation is unrolled to a depth of 4 and performs pipelined operations and data prefetching. Though it is not immediately obvious from the listing, all memory references in the daxpy_y* implementation conform to the memory access guidelines of 3.3; this will become apparent as the daxpy_y* code is examined.

Referring to Figure 9, line 5 issues a *fld* request for the next two elements of the **y** vector; on a cache hit this instruction executes in a single cycle, a cache miss initiates a cache line load of the next 4 elements of **y**. In performing a cache line load, a cache line write-back may occur; if this is the case, 4 elements of the **y** vector are written back to main memory. Once the *fld* instruction of line 5 has completed, lines 6-4 perform 5 *pfld* operations from the **x** vector at the maximum rate of 1 every 2 cycles; note the use of the dummy load on line 2 to prevent the DRAM controller from initiating an idle-state transition, as described in 3.2.3. The instructions referencing the **y** vector, which are interleaved with the instructions reading the **x** vector, will all be cache hits and as such will not interfere with the loading of **x** elements; this is an important aspect of the daxpy_y* inner-loop code. Thus, the daxpy_y* operation adheres to the guidelines of 3.3 as accesses to each vector operand are performed in sets, like memory operations are grouped and all accesses are performed at the maximum rate at which they can be satisfied by the memory system.

A performance estimate can be derived for the daxpy_y* implementation in the same fashion as for daxpy_1 and daxpy_2; in doing so, one must take into account both the cache line load and write-back which may occur with the execution of a *fld* instruction.

In the description above, all cache line operations occur as a result of the *fld* instruction of line 5; this is the case when the **y** vector is aligned on a 32-byte boundary. When **y** is aligned on a 16-byte boundary only the first cache line operation is initiated at line 5, all others result from the *fld* instruction of line 7; in this case, lines 8-6 perform the 5 *pfld*

operations from the **x** vector without interference. The performance of the daxpy_y* operation is the same for either **y** vector alignment.

Figure 9 DAXPY_Y* Inner Loop Code

```
// (prev) - refers to a value associated with the previous iteration
// prime(‘) - refers to a value associated with the next iteration

.inner_loop:

1)  d.pfmul.dd   f12,f30,f2                    // a * x3 : ax1
    fst.q        f4,r27(r26)++                 // store s0, s1 (prev)
2)  d.pfadd.dd   f18,f2,f10                    // y1 + ax1 : s3 (prev)
    pfld.d       0(r24),f26                    // x’1 & RE-load x’3
3)  d.pfmul.dd   f12,f24,f2                    // a * x’0 : ax2
    fst.q        f8,r27(r26)++                 // store s2, s3 (prev)
4)  d.pfadd.dd   f20,f2,f0                     // y2 + ax2 : 0
    pfld.d       r20(r24)++,f28                // x’2 & load x’’0
5)  d.pfmul.dd   f12,f26,f2                    // a * x’1 : ax3
    fld.q        r27(r25)++,f16                // y’0 & y’1
6)  d.pfadd.dd   f22,f2,f4                     // y3 + ax3 : s0
    pfld.d       r20(r24)++,f30                // x’3 & load x’’1
7)  d.pfadd.dd   f0,f0,f6                      // push : s1
    fld.q        r27(r25)++,f20                // y’2 & y’3
8)  d.fnop
    pfld.d       r20(r24)++,f0                 // ditch dummy & load x’’2
9)  d.pfmul.dd   f12,f28,f2                    // a * x’2 : ax’0
    bla          r17,r23,.inner_loop           // loop test - exit when 4 remain
10)  d.pfadd.dd  f16,f2,f8                     // y0 + ax0 : s2
     pfld.d      r20(r24)++,f24                // x’’0 & load x’’3
```

4.4 VAXPY Operation

A routine which we will refer to as *vaxpy*, standing for *vector axpy*, implements a double-precision vector operation of the form:

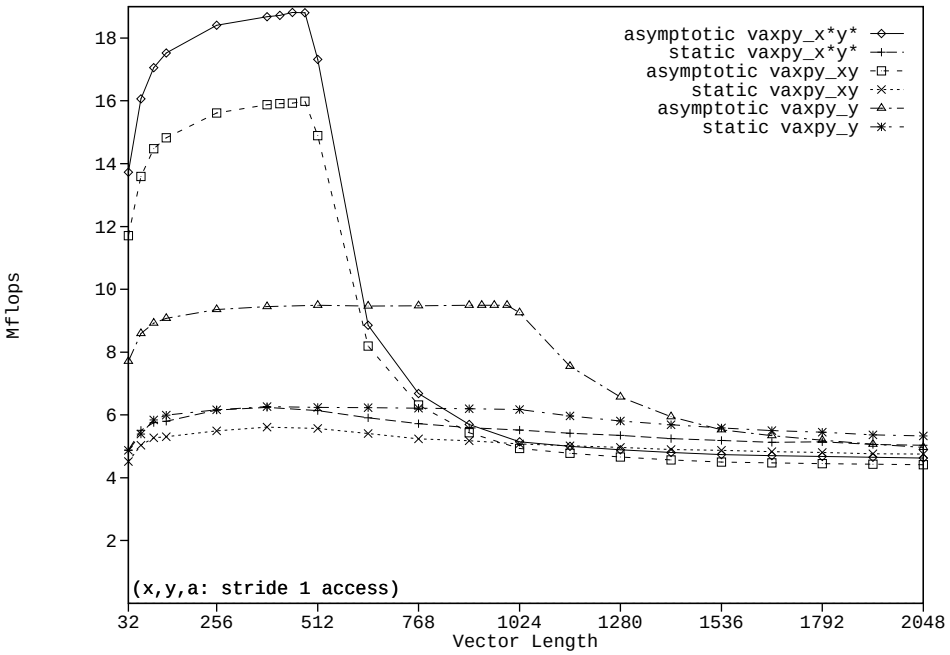
$$\mathbf{y} \leftarrow \mathbf{ax} + \mathbf{y}$$

where **a**, **x**, and **y** are vectors and **ax** implies elementwise vector multiplication; as demonstrated in 5.1, the vaxpy operation is useful in implementing matrix-vector multiplication for diagonally sparse matrices. The graph of Figure 10 depicts the asymptotic and static performance of the vaxpy implementations vaxpy_x*y*, vaxpy_xy, and vaxpy_y; the curves for all three have the characteristic shape associated with operations which cache one or more vector operands, as described in 4.3.1. Both the vaxpy_x*y* and vaxpy_xy implementations cache the **x** and **y** operands, the vaxpy_y implementation caches only the **y** operand; vaxpy_x*y* assumes cached vector operands to be aligned on either a 16- or 32-byte boundary, allowing two double-precision values to be loaded with a single *fld* instruction. For all performance measures, vector operands are stored with a stride of 1.

Implementing vector operations with more than two vector operands, at least one of which is cached, often requires compromise in attempting to adhere to the memory access guidelines of 3.3; coding is further complicated when trying to implement an operation which

performs well under conditions of both asymptotic and static performance. The following examines the vaxpy_x*y* implementation and discusses these issues.

Figure 10 Performance of Various VAXPY Implementations



4.4.1 VAXPY_X*Y*

Figure 11 lists the inner-loop code for the vaxpy_x*y* implementation of the vaxpy operation. Like the daxpy implementations of 4.3, vaxpy_x*y* is unrolled to a depth of 4 and performs pipelined operations and data prefetching. The memory access pattern for the vaxpy_x*y* implementation will not in all cases conform to the memory access guidelines of 3.3; this will become apparent as the code is examined. To simplify the discussion of the vaxpy_x*y* code, the issue of data cache thrashing is ignored.

Given that the **x** and **y** vectors for vaxpy_x*y* may each be aligned on either a 16- or 32-byte boundary, it is not known a priori which *fld* instructions have the potential to initiate cache-line loads. Unlike the daxpy_y* implementation of 4.3.4, the vaxpy_x*y* *pfld* and *fld* instructions can not be interleaved indiscriminately; for vector operations with more than 2 vector operands, doing so can result in more DRAM page misses per inner-loop iteration than the total number of vectors being accessed. Therefore, a conservative approach is taken in implementing the vaxpy_x*y* inner-loop code, *fld* instructions which reference the same vector operand are grouped.

Referring to Figure 11, lines 7-8 issue *fld* requests for the next 4 elements of the **x** vector and lines 9-10 issue *fld* requests for the next 4 elements of the **y** vector; on a cache-hit each of these instructions executes in a single cycle, a cache-miss initiates a cache line load of 4

elements from the corresponding vector. In performing a cache line load, a cache line write-back may also occur. Lines 12-6 issue *pfld* requests for the next 4 elements of the **a** vector; the *fst* instructions of lines 1 and 5 will be cache hits and as such will not interfere with the loading of **a** elements. Thus, accesses to each vector operand are performed in sets and like memory operations are grouped.

Under the conditions for measuring static performance, all memory accesses for the vaxpy_x*y* implementation are performed at the maximum rate at which they can be satisfied by the memory system; the 2 consecutive cache line loads from, and potential write-backs to, the **x** and **y** vectors are immediately followed by 4 *pfld* operations from the **a** vector issued at a rate of 1 every 2 clock cycles. Consequently, for static performance the vaxpy_x*y* memory access pattern conforms to all memory access guidelines of 3.3.

In deriving the upper-bound on asymptotic performance for the vaxpy_x*y* implementation, it is assumed that all **x** and **y** vector accesses result in data cache hits; in this case, the execution of 4 *pfld* instructions from the **a** vector is followed by a 5 cycle delay as elements of **x** and **y** are loaded from the data cache and a control-transfer instruction is executed. Thus, under conditions for measuring asymptotic performance, vaxpy_x*y* accesses are not performed at the maximum rate at which they can be satisfied. As discussed in 3.2.3, this delay in vaxpy_x*y* memory access is sufficient to allow the DRAM controller to initiate an idle-state transition; the net penalty to asymptotic performance is 3 cycles additional overhead incurred by the *pfld* instruction at line 12, the 8 cycle far access penalty resulting from the DRAM controller idle-state transition minus the 5 cycle delay between accesses.

Figure 11 VAXPY_X*Y* Inner Loop Code

```
// (prev) - refers to a value associated with the previous iteration
// prime(‘) - refers to a value associated with the next iteration

.inner_loop:

1)  d.pfmul.dd   f26,f10,f0                    // a1 * x1 : 0
    fst.q        f4,r28(r26)++                 // store s0, s1 (prev)
2)  d.pfadd.dd   f0,f0,f24                     // push : s2 (prev)
    pfld.d       r20(r27)++,f14                // a3 & load a’2
3)  d.pfmul.dd   f28,f12,f2                    // a2 * x2 : a0x0
    nop
4)  d.pfadd.dd   f16,f2,f26                    // y0 + a0x0 : s3 (prev)
    pfld.d       r20(r27)++,f8                 // a’0 & load a’3
5)  d.pfmul.dd   f30,f14,f2                    // a3 * x3 : a1x1
    fst.q        f24,r28(r26)++                // store s2, s3 (prev)
6)  d.pfadd.dd   f18,f2,f0                     // y1 + a1x1 : 0
    pfld.d       r20(r27)++,f10                // a’1 & load a’’0
7)  d.pfmul.dd   f0,f0,f2                      // push : a2x2
    fld.q        r28(r24)++,f24                // x’0 & x’1
8)  d.pfadd.dd   f20,f2,f0                     // y2 + a2x2 : 0
    fld.q        r28(r24)++,f28                // x’2 & x’3
9)  d.pfmul.dd   f0,f0,f2                      // push : a3x3
    fld.q        r28(r25)++,f16                // y’0 & y’1
10) d.pfadd.dd   f22,f2,f4                     // y3 + a3x3 : s0
    fld.q        r28(r25)++,f20                // y’2 & y’3
11) d.pfmul.dd   f8,f24,f0                     // a’0 * x’0 : 0
    bla          r17,r23,.inner_loop           // loop test - exit when 4 remain
12)  d.pfadd.dd  f0,f0,f6                      // push : s1
     pfld.d      r20(r27)++,f12                // a’2 & load a’’1
```

Without data cache control, implementing an efficient vector operation which caches more than one operand is complicated by data cache thrashing. In the discussion of the daxpy_y* implementation in 4.3.4 it is assumed that for a single cached operand, *fst* operations hit in the data cache when performed on vector elements from the most recent cache line load; in the case of multiple cached operands, a cache line load from an operand different from a *fst* instruction's target operand can potentially force a write-back of the target cache line prior to the *fst* instruction's execution. When implementing vector operations which modify one of two or more cached operands, the last cache line load prior to the execution of a *fst* instruction should be the *fst* instruction's target.

4.5 Implementation Guidelines for Vector Operations

As demonstrated in the examples of 4.3 and 4.4, the structure of inner-loop code has a significant impact on the overall performance of a vector operation. The following presents guidelines for implementing vector operation inner-loop code in a manner consistent with the i860 architecture and RX-1 memory system characteristics:

- Unroll inner-loop code and utilize the i860's dual-instruction mode to simultaneously perform pipelined floating-point operations and data prefetching; for implementations which cache vector operands, unrolling to a depth of 4 is convenient given that a data cache line accommodates 4 double-precision values.
- Structure inner-loop code around memory access instructions; a vector operation's memory access pattern should conform whenever possible to the memory access guidelines of 3.3.
- Excepting DRAM page misses which result from cache line write-backs, the number of page misses in a well structured inner-loop should not exceed the number of vector operands being accessed. For cached vector operands, if vector alignment is known a priori then it can be determined which *fld* instructions have the potential to initiate cache line operations; in this case, *pfld* and *fld* instructions can be interleaved in such a way so as not to force extraneous DRAM page misses.
- To reduce data cache thrashing in vector operations which modify one of two or more cached operands, the last cache line load prior to the execution of a *fst* instruction should be the *fst* instruction's target.
- If a cached vector operand is aligned on either a 16- or 32-byte boundary, two double-precision values can be accessed with a single *fld* or *fst* instruction; this is often convenient for structuring inner-loop code and may improve performance. For a single cached operand aligned on an 8-byte boundary, the alignment can be adjusted to a 16-byte boundary by performing the vector operation on the first element of each vector operand. In the case of multiple cached operands, alignment adjustment can be performed on at least one vector; if a cached vector operand is modified, its alignment should be adjusted since it will be referenced by both *fld* and *fst* inner-loop instructions.

The task of implementing an efficient vector operation is simplified if the context in which it will be used is known in advance. More often, implementing efficient vector operations is the art of compromise; whenever possible, inner-loop code should be structured so that

under conditions of both static and asymptotic performance, memory access patterns are consistent with the characteristics of the RX-1 memory system.

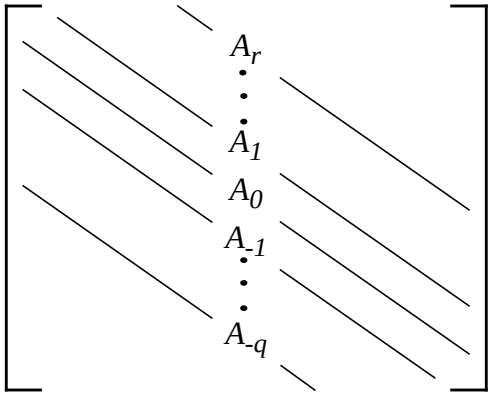
5.0 Sparse Matrix-Vector Multiplication

This section examines matrix-vector multiply algorithms which form $\mathbf{y} = A\mathbf{x}$ where A is a sparse matrix. In particular, algorithms for diagonally sparse and completely sparse matrices are presented. For each type of matrix, storage schemes and implementation issues are discussed for optimizing performance on the RX-1.

5.1 Diagonally Sparse Matrix-Vector Multiply

A *diagonally sparse matrix* is a matrix composed of a relatively few non-zero diagonals; such matrices arise frequently in practice, for example in the discretization of elliptic partial differential equations [9]. Due to the relatively large percentage of zero elements in each row and column of a diagonally sparse matrix, it is common practice to store only the non-zero diagonals; in performing operations involving such matrices, the diagonals become the natural vectors.

A matrix-vector multiply algorithm which operates on the diagonals of a matrix is described in [8]; given an $n \times n$ matrix A



and a vector $\mathbf{x}$ of length n, the product $A\mathbf{x}$ can be represented as:

$$A\mathbf{x} = A_0\mathbf{x} <+ A_1\mathbf{x}^2 <+ \ldots <+ A_r\mathbf{x}^{r+1} +> A_{-1}\mathbf{x}_{n-1} +> \ldots +> A_{-q}\mathbf{x}_{n-q} \qquad \text{(EQ 3)}$$

where $\mathbf{x}^i = (x_i,\ldots,x_n)$ and $\mathbf{x}_{n-i} = (x_1,\ldots,x_{n-i})$

In (EQ 3), $A_i\mathbf{x}^{i+1}$ and $A_{-i}\mathbf{x}_{n-i}$ are elementwise vector multiplications where the vectors A_i are the diagonals of A; A_0 denotes the main diagonal, $A_1,\ldots,A_r$ denote diagonals above A_0, and $A_{-1},\ldots,A_{-q}$ denote diagonals below. For diagonally sparse matrices, only the non-zero vectors A_i participate in the computation. Since the vectors of (EQ 3) are of different lengths, the operator <+ is defined as vector addition in which the shorter vector is added

to the first components of the longer vector; similarly, the operator +> is defined as vector addition in which the shorter vector is added to the last components of the longer vector. For example, in the operation $A_0\mathbf{x}$ <+ $A_1\mathbf{x}^2$ the vector $A_1\mathbf{x}^2$ of length $n-1$ is added to the first $n-1$ components of the vector $A_0\mathbf{x}$.

The formation of an elementwise vector product and its subsequent addition to another vector is simply the vaxpy operation of 4.4; all implementations of (EQ 3) presented here incorporate either vaxpy_y or vaxpy_xy as the inner-loop. To distinguish between various implementations of the multiplication by diagonals algorithm, the following notation is adopted:

mbd_<*cached operand*>{<*cached operand*>}

where ‘mbd’ stands for ‘multiplication by diagonals’; the argument enclosed in ‘{}’ is optional. In the naming scheme above, the ‘<*cached operand*>’ is the same operand cached by the vaxpy inner-loop.

5.1.1 Strip-mining MBD_Y and MBD_XY

In implementing the multiplication by diagonals algorithm of (EQ 3) to form $\mathbf{y} = A\mathbf{x}$, *strip-mining* is employed to promote reuse of cached data. Dividing the matrix *A* and the result vector **y** into partitions consisting of *p* rows each allows a portion of **y** to be cached as the corresponding portions of $A_i\mathbf{x}^{i+1}$ and $A_{-i}\mathbf{x}_{n-i}$ are being computed and summed; this strip-mining process is represented graphically in Figure 12. Depending on the relative positions of the non-zero diagonals in *A*, performance may further be improved by caching **x** vector elements as well; this is discussed in detail below.

To facilitate the discussion on strip-mining (EQ 3), the term *partition result* is defined to mean the value of a given partition of the **y** vector resulting from the summation of the products $A_i\mathbf{x}^{i+1}$ and $A_{-i}\mathbf{x}_{n-i}$ computed within the corresponding partition of the *A* matrix.

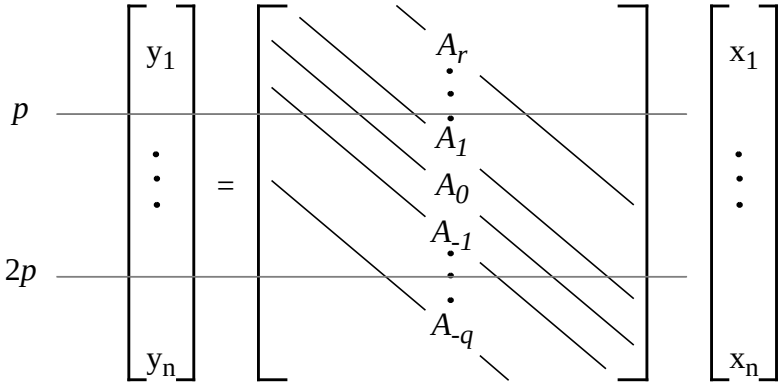
Ignoring the issue of cache control, three factors affect performance in computing a given partition result: the number, length and position of the non-zero diagonal vectors which pass through the corresponding *A* partition. The number of non-zero diagonal vectors in a given *A* partition places an upper-bound on the number of times a cached **y** or **x** vector element can potentially be accessed in computing that partition result. The length of a diagonal vector in a given *A* partition places an upper-bound on the maximum computation rate of the vaxpy inner-loop for which that vector is an operand. Finally, the absolute position of a diagonal vector in a given *A* partition determines which elements of **y** and **x** are accessed by the vaxpy inner-loop operating on that diagonal; in computing a partition result, the relative positions of all non-zero diagonals determines which elements of **y** and **x** are accessed more than once.

Elements of the **x** vector should only be cached in the case where adjacent non-zero diagonals of matrix *A* are close enough so that vaxpy operations performed in computing a given partition result access a large percentage of the same **x** elements; vaxpy inner-loop

operations involving diagonals which are far apart will have few, if any, **x** elements in common.

The partition size *p* is chosen as the largest value such that the total number of vector elements cached in computing a given partition result does not exceed data cache capacity; if only elements of **y** are cached then $p = 1024$, when caching both **x** and **y** values $p = 512$. Maximizing *p* increases the length of the diagonal vectors in a given *A* partition and reduces the total number of partitions and any associated bookkeeping overhead.

Figure 12 Strip-mining the Multiplication by Diagonals Algorithm (y = *A*x)



5.1.2 Effects of Data Cache Thrashing on MBD_Y and MBD_XY

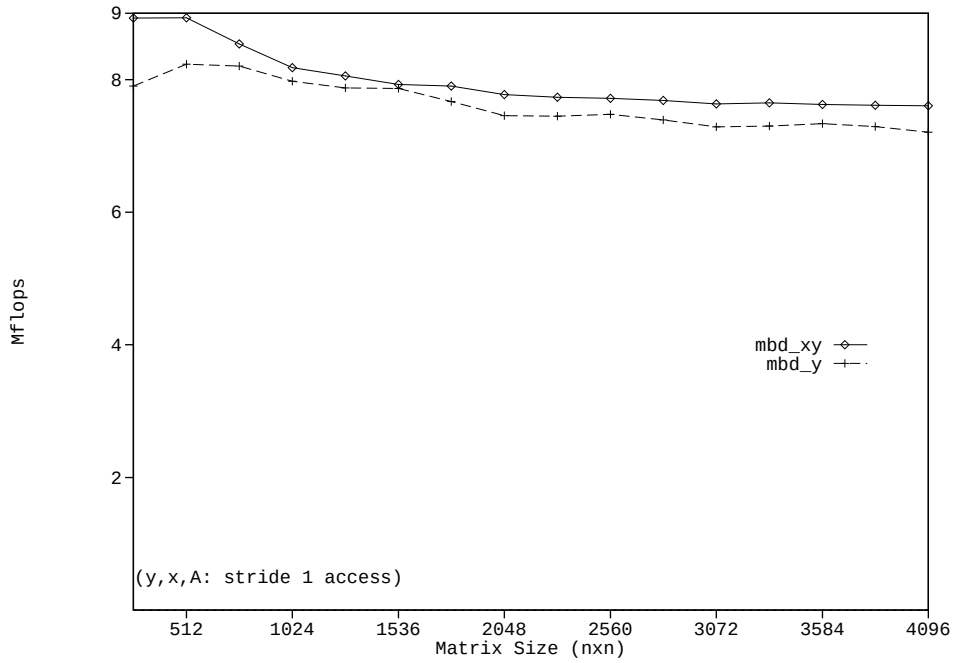
A matrix *A* is defined as a diagonally sparse matrix such that A_{-2}, A_{-1}, A_0, A_1 and A_2 are the only non-zero diagonals. Assuming data cache control as described in 4.1, it is possible to derive an approximate computation rate for the matrix-vector multiply operations mbd_y and mbd_xy in forming the product $\mathbf{y} = A\mathbf{x}$. In computing a given partition result, the first of the vaxpy inner-loop operations executes at approximately the static performance rate for vectors of length 2*p*; this takes into account the initial cache line write-backs which occur for all but the first partition. The remaining 4 vaxpy inner-loop operations execute at the theoretical maximum asymptotic performance rate; given the matrix *A* defined above, this is true even in the case of a vaxpy_xy inner-loop since products involving adjacent non-zero diagonals have all but one **x** vector element in common. Vaxpy performance data is taken from the graph of Figure 10; for simplicity, all diagonal vectors in a given *A* partition are assumed to be of length *p*. Deriving an approximation under the conditions stated above yields a computation rate of 8.7 Mflops for the mbd_y operation, and 13.7 Mflops for mbd_xy; actual performance will be somewhat less due to the bookkeeping overhead involved in strip-mining.

The graph of Figure 13 depicts the static performance of the operations mbd_y and mbd_xy when applied to the matrix A for various dimensions *n*; these measurements were made under the random cache line replacement policy described in 4.1. As a result of data cache thrashing, both operations perform below the projected computation rates derived above; the difference in projected performance and measured performance for the mbd_y

operation is approximately 16%, for the mbd_xy operation the difference is approximately 45%.

While the performance of both the mbd_y and mbd_xy operations is below that achievable if it were possible to utilize the data cache control mechanism, it is the mbd_xy operation which is most adversely affected by data cache thrashing; to achieve theoretical maximum asymptotic performance the vaxpy_y inner-loop of mbd_y requires 1/3 of the vector elements accessed to be cache resident, the vaxpy_xy inner-loop of mbd_xy requires 2/3. Given a matrix consisting of only 5 non-zero diagonals, in computing a given partition result the vaxpy inner-loop operation is not executed a sufficient number of times for the ‘cache sorting’ effect described in 4.2 to significantly affect vaxpy performance. Thus, as demonstrated in Figure 13, caching **x** vector elements only marginally improves performance even under optimal conditions in which all adjacent non-zero diagonals reside at a distance of 1.

Figure 13 Performance of MBD_Y and MBD_XY



5.1.3 Effects of Strip-mining on MBD_Y

The graph of Figure 14 depicts the static performance of the operations mbd_y and mbd_y_ns when applied to the matrix *A* defined for 5.1.2; in mbd_y_ns the ‘_ns’ stands for ‘non-strip-mined’ and signifies that this operation is an implementation of (EQ 3) which does not employ strip-mining. Both mbd_y and mbd_y_ns incorporate vaxpy_y as the inner-loop.

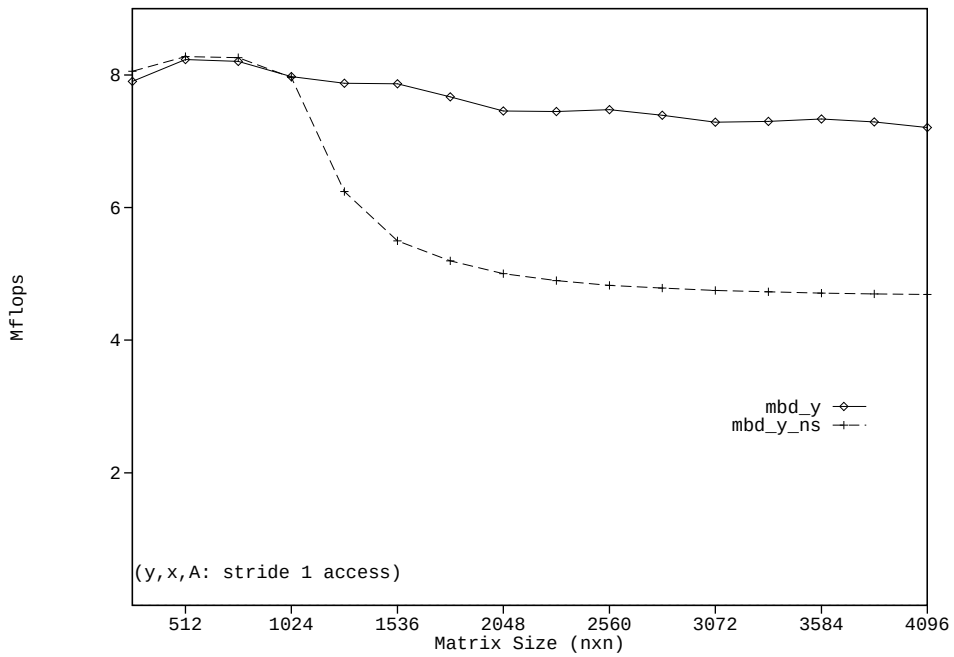
The curve for the mbd_y operation in Figure 14 is relatively flat, though performance is somewhat better in the case where $n < 2p$. Since the data cache is flushed prior to per-

forming each mbd_y operation, fewer cache line write-backs occur in computing the initial partition result than occur in computing subsequent partition results; thus, the computation rate for the initial partition result is higher. For $n > p$, the higher performance experienced in computing the initial partition result is amortized across all partition result computations.

The mbd_y_ns curve is similar in shape to the asymptotic performance curve for vaxpy_y depicted in Figure 10; as cache capacity is reached at a diagonal vector length of 1024, performance drops off rapidly and degrades to a level somewhat below the static performance of vaxpy_y.

Strip-mining improves performance of the mbd_y operation by dividing the **y** vector into partitions which can be accessed by the vaxpy_y inner-loop for a number of consecutive iterations without exceeding data cache capacity. Given $n > p$, strip-mining reduces the number of times an element of **y** is loaded from main-memory to the data cache and amortizes the cost of each **y** element load with every reuse.

Figure 14 Performance of MBD_Y and MBD_Y_NS



5.2 Completely Sparse Matrix-Vector Multiply

A *completely sparse matrix* is a matrix with no discernible structure and a relatively large percentage of zero elements in each row and column; such matrices arise frequently in engineering problems, for example in the analysis of power distribution systems [1]. Storage schemes for completely sparse matrices usually involve the storage of each non-zero element and its associated row and column position; the data structure used in storing this information is generally optimized for the matrix operation to be performed [3][13].

Given an $n \times n$ completely sparse matrix A with N non-zero elements, in implementing a matrix-vector multiply $\mathbf{y} = A\mathbf{x}$ on the RX-1, elements of A are stored in a single vector of triples (i_k, j_k, a_k) where a_k is a non-zero element of A in position (i_k, j_k) and $1 \le k \le N$. This vector of triples is referred to as **t** and can be defined in the *C* programming language as an array of elements of type

```
typedef struct {
   long     i,j;
   double   a;
} t_element;
```

or in Fortran as an *equivalence* between a double-precision and an integer array. Elements of **t** are stored in row order such that given two adjacent elements (i_k, j_k, a_k) and $(i_{k+1}, j_{k+1}, a_{k+1})$ then $i_k \le i_{k+1}$; if $i_k = i_{k+1}$ then $j_k < j_{k+1}$. The vectors **x** and **y** are stored as n element double-precision arrays.

In the storage scheme for the RX-1 defined above, placing non-zero elements with associated row and column positions in a single vector reduces the total number of vectors accessed in performing $\mathbf{y} = A\mathbf{x}$; consequently, for a single iteration of the matrix-vector multiply algorithm, the minimum number of DRAM page misses is reduced. An alternative scheme is to store (j_k, a_k) pairs in one array and define a second array of n elements pointing to the beginning of each row in the first; the total amount of storage space is reduced at the cost of increasing the number of vectors to be accessed.

5.2.1 Implementation of CS_MVM

Figure 15 lists the inner-loop code for cs_mvm, standing for ‘completely sparse matrix-vector multiply’, which implements a matrix-vector multiplication $\mathbf{y} = A\mathbf{x}$ utilizing the storage scheme described above. For a given row i of matrix A, the inner-loop code of cs_mvm computes the corresponding value y_i; each iteration of the cs_mvm inner-loop code performs a single $y_{i_k} = y_{i_k} + a_k \cdot x_{j_k}$ operation, the loop executes once for each k in the set of (i_k, j_k, a_k) with the same i_k value.

The cs_mvm inner-loop performs data prefetching in the same manner as the daxpy and vaxpy operations of 4.3 and 4.4, respectively. However, since the number of elements in a given row is unknown, the cs_mvm inner-loop is not unrolled; a test is performed after reading each (i_k, j_k, a_k) to detect a change in rows. In addition, floating-point operations are performed in scalar rather than pipelined mode; this avoids having to drain the floating-point pipelines with each row change. The ratio of core instructions to floating-point instructions is sufficiently large so that core instructions can be executed during the time periods in which the floating-point unit is engaged in performing a scalar operation.

In performing cs_mvm for the completely sparse matrix-vector multiply $\mathbf{y} = A\mathbf{x}$, only elements of **x** are cached; triple values (i_k, j_k, a_k) from the **t** vector are loaded via the *pfld* instruction. Elements of the **x** vector may be accessed in any order, the reference pattern is a function of the A matrix structure; since references are not predictable, data cache control is not an issue. As discussed in 4.1, data cache thrashing can potentially occur for val-

ues of the matrix size *n* such that the number of elements in **x** exceeds one-half data cache capacity; i.e. $n > 512$.

Of particular interest in the cs_mvm inner-loop code is the manner in which the i_k and j_k values of (i_k,j_k, a_k) are loaded from the **t** vector. Referring to Figure 15, the *pfld* instruction of line 1 loads i_k and j_k as a single double-precision value; given that integers require 32 bits, both i_k and j_k fit in a single floating-point register pair. The *fxfr* instructions of lines 5-6 transfer the i_k and j_k values to integer registers in the core unit; once transferred, the value of j_k is used in the *fld* instruction of line 9 as an offset into the **x** vector and the value of i_k is used in lines 8 and 10 to test for a change in rows.

The *fxfr* instruction allows composite values to be accessed via the *pfld* instruction in an ad hoc fashion, though the effective load rate is reduced by the overhead incurred in transferring values to the core unit's integer registers; a more elegant implementation of the *pfld* instruction would allow values from the load pipe to be directed to either the floating-point or core unit register files.

Based on the memory system performance data of 3.2, an approximate upper- and lower-bound can be derived for the performance of the cs_mvm inner-loop code; the assumption is made that the **x**, **y** and **t** vectors do not have any DRAM pages in common. Given that the *fld* instruction of line 9 is a cache hit, the execution of 3 *pfld* instructions from the **t** vector is followed by a 6 cycles delay as the appropriate element of **x** is loaded from the data cache and a test is made to detect a change in rows. As discussed in 3.2.3, this delay in cs_mvm memory access is sufficient to allow the DRAM controller to initiate an idle-state transition; the net penalty is 2 cycles additional overhead incurred by the *pfld* instruction at line 1, the 8 cycle far access penalty resulting from the DRAM controller idle-state transition minus the 6 cycle delay between accesses. Thus, the upper-bound on cs_mvm inner-loop performance is 13 cycles for 2 floating-point operations, achieving a computation rate of $6.2\ \text{Mflops}$ at $40\ \text{MHz}$.

If the *fld* instruction of line 9 is a cache-miss then the resulting cache line load requires 13 cycles, a net DRAM page miss overhead of 5 cycles plus 2 cycles for each of the 4 double-precision floating-point loads. The *pfld* instruction of line 1 is stalled by the cache-line load for a net delay of 6 cycles, as discussed in 3.2.4. Summing together instruction execution times and access overheads yields a lower-bound on cs_mvm inner-loop performance of 29 cycles for 2 floating-point operations, or a computation rate of $2.8\ \text{Mflops}$ at $40\ \text{MHz}$.

When a change in row values i_k is detected in the inner-loop code of Figure 15, the loop exits; before the inner-loop is reentered the computed value y_i is stored, the y_i accumulation register is zeroed and a test is performed to detect further values of (i_k,j_k, a_k) to process. For rows of the matrix *A* with few elements, the bookkeeping overhead which is incurred after computing a given value y_i can become a significant portion of the y_i com-

putation cost; thus, the lower-bound on performance for the entire cs_mvm computation is below that of the cs_mvm inner-loop of Figure 15.

Figure 15 CS_MVM Inner Loop Code

```
// prime(') - refers to a value associated with the next iteration

.while_i_eq_old_i:

1)  d.fmul.dd    f16,f20,f24                    // a * x[j]
    pfld.d       r30(r16)++,f18                 // t.i_t.j' & load t.i_t.j''
2)  d.fnop                                      // pause for mult (3)
    nop
3)  d.fnop                                      // pause for mult (2)
    pfld.d       r30(r16)++,f16                 // a' & load a''
4)  d.fnop                                      // pause for mult (1)
    nop
5)  d.fxfr       f19,r19                        // transfer to j
    pfld.d       0(r16),f0                      // discard & RE-load a''
6)  d.fxfr       f18,r20                        // transfer to i
    nop
7)  d.fadd.dd    f22,f24,f22                    // y + ax
    shl          3,r19,r19                      // j * sizeof(double)
8)  d.fnop                                      // pause for add(2)
    xor          r20,r21,r0                     // i == old_i?
9)  d.fnop                                      // pause for add (1)
    fld.d        r19(r17),f20                   // x[j]
10) d.fnop
    bc.t     .while_i_eq_old_i                  // branch to top of inner while
11)  d.fnop
     nop
```

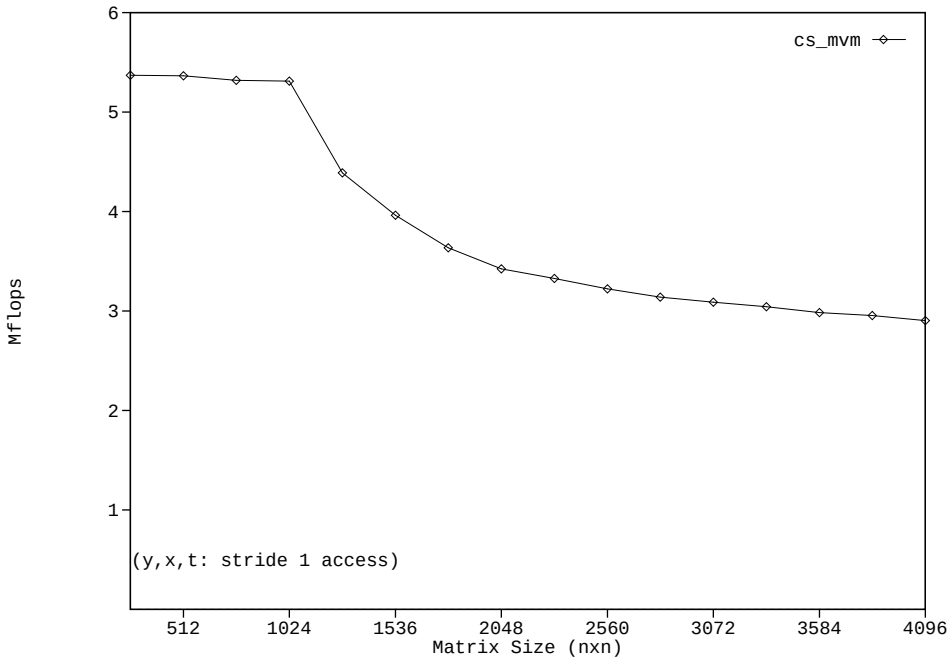
5.2.2 Performance of CS_MVM

The graph of Figure 16 depicts the static performance of the cs_mvm operation for various dimensions n of the matrix A. For a given matrix A, the parameter d is defined as the *matrix density* where d is the maximum number of non-zero elements which reside in any given row of A. In generating a matrix for measuring performance, the number of non-zero elements a_k in each row of A is drawn from a uniform distribution over the integers in the interval $[0,d]$; likewise, the column position j_k of each element a_k is drawn from a uniform distribution over the integers in the interval $[1,n]$.

For a given matrix size n, increasing the density d tends to increase reuse of cached x_j values and decrease the number of extraneous x_j loads which occur as a result of the cache line load procedure; the performance of the cs_mvm operation increases accordingly. Conversely, for a given density d, increasing the matrix size n tends to decrease reuse of cached x_j values and increase the number of extraneous x_j loads; the result is a decrease in cs_mvm performance as demonstrated by the graph of Figure 16 in which $d = 50$ for all values of n. Furthermore, for $n > 1024$ the number of elements x_j cached while perform-

ing cs_mvm may exceed data cache capacity; if this is the case, performance of the cs_mvm operation degrades as a result of data cache thrashing.

Figure 16 Performance of CS_MVM (*d*=50)



6.0 Summary

From examining the RX-1 processor-memory interrelationship in detail, guidelines have been established for implementing operations in a manner consistent with the i860 architecture and memory sub-system characteristics; as demonstrated throughout this paper, operations which adhere to these implementation guidelines achieve a level of performance well above those which do not. It is interesting to note that nearly all of the guidelines presented in 3.3 and 4.5 relate to generating access patterns amenable to efficient memory system operation.

The ability to perform some level of data cache control is important in implementing operations to make effective use of the data cache; this is well demonstrated in the diagonally sparse matrix-vector multiply of 5.1 where operating under a random cache line replacement policy reduces performance by as much as 45%. Though the i860 does provide a data cache control mechanism, it is only available in supervisor mode; this is an unfortunate implementation decision since making data cache control available in user mode compromises neither system performance nor security.

The RX-1 computational node exhibits a basic imbalance between processor speed and memory system bandwidth. For example, given that in pipelined mode the i860 can produce one double-precision multiply result every 2 clock cycles, the upper-bound on performance for the vaxpy operation of 4.4 is 40 Mflops at a clock rate of 40 MHz. To

sustain a vaxpy computation rate of 40 Mflops , the RX-1 memory system must satisfy 60 million load and 20 million store operations per second; this is 4.5 times the bandwidth available from even the fastest access modes, ignoring DRAM page miss and DRAM controller read/write mode switch overheads.

For a computer system to function efficiently as a whole, a balance must be maintained in the performance of its individual components. As this study of the RX-1 computational node suggests, further research is needed to develope memory system architectures which can better support the demands imposed by high performance processors such as the i860. Basic scientific computational kernels which operate on vectors tend to generate very regular access patterns, given the proper memory system architecture it should be possible to exploit these regular access patterns to increase memory system performance.

Bibliography

[1] Baumann, R. “Sparseness in Power Systems Equations”, Large Sparse Sets of Linear Equations: Proc. Oxford Conf. Inst. Math. Appl. - April 1970, Academic Press, New York, NY, 1971, pp 105-126.

[2] Bokhari, S. “Communication Overhead on the Intel iPSC/860 Hypercube”, ICASE, NASA Langley Research Center, Hampton, VA.

[3] George, A., and Liu, J. “Computer Solution of Large Sparse Positive Definite Systems”, Prentice-Hall, Englewood Cliffs, NJ, 1981, pp 138-152.

[4] Intel Corporation, “i860 64-Bit Microprocessor Hardware Reference Manual”, ISBN 1-55512-106-3, 1989.

[5] Intel Corporation, “i860 64-Bit Microprocessor Programmer’s Reference Manual”, ISBN 1-55512-080-6, 1989.

[6] Nugent, S. “The iPSC/2 Direct Connect Communications Technology”, Proc. 3rd Conf. on Hypercube Concurrent Comp. and Appl., ACM, New York, NY, 1988, pp 51-60.

[7] Lawson, C., Hanson, R., Kincaid, D., and Krogh, F. “Basic Linear Algebra Subprograms for Fortran Usage”, ACM Trans. Math. Softw., 5, 3, Sept. 1979, pp 308-323.

[8] Ortega, J. “Introduction to Parallel and Vector Solution of Linear Systems”, Plenum Press, New York, NY, 1988, pp 50-53.

[9] Ortega, J., and Poole, W. “An Introduction to Numerical Methods for Differential Equations”, Pitman Publishing, Marshfield, MA, 1981, pp 268-275.

[10] Scott, D., and Withers, G. “Performance and Assembly Language Programming of the iPSC/860 System (Preliminary)”, Intel Scientific Computers, Beaverton, OR, 1990.

[11] Sedra, A., and Smith, K. “Microelectronic Circuits”, CBS College Publishing, New York, NY, 1982, pp 748-753.

[12] Serang, O. “Various Features of Fujitsu DRAMs”, MOS Memory Products, Fujitsu Microelectronics, 1989, pp 1:3-24.

[13] Tewarson, R. “Sparse Matrices”, Academic Press, New York, NY, 1973, pp 1-11.

[14] Zilka, A., Intel Scientific Computers, personal communication, March 22, 1991.