Chapter 10

Performance Considerations

This chapter documents the basic timings of the accelerator chip and its interface to the IU. The slant here is toward a better understanding of issues relating to system performance. The DPEAC coder will benefit from the Algorithms sections; the higher–level language coder will find the Runtime System Performance section of interest; and all readers, particularly armchair architects, will find useful information in the first section.

10.1 Performance Terminology

For consistency, the following terms are used to refer to the memory heirarchy:

Physical Memory — The 4 banks of Dynamic RAM comprising the memory controlled by the accelerator in its capacity as a memory controller. The currently available sizes are 8 Mbyte/bank and 32 Mbyte/bank.

Cache Memory — The memory used to store fetched cache lines for fast access by the IU. The current architecture is a single–set virtual–addressed linear cache, with a cache line size of 8 words (32 bytes). The total cache size is 64Kbytes, and is shared by both instruction and data access.

IU Memory — These are the pages of memory mapped randomly from 2 accelerator chips into the virtual address space of the user for the purpose of access by the SPARC. This is cached memory.

Parallel Memory — These are the contiguous physical memory blocks allocated to the process for the purpose of access by the accelerator chip's vector units. This memory is mapped into the IU virtual address space, but is accessed by the IU without cacheing.

IU Registers — The registers of the IU (SPARC), i.e.`%i0`, `%g2`, etc.

FPU Registers — The registers of the Floating Point Coprocessor, e.g. `%f0`, `%f1`, etc. These are not accessible by the IU, but the IU can issue instructions to load or store them from memory.

VU Registers — The registers allocated to one of the two VUs on either of the accelerator chips. In DPEAC naming, these are `R0` through `R127` (also referenced as `Vn` or `Sn`).

10.2 Basic Operation Performance

10.2.1 IU Memory (Cached) Accesses

The performance of store instructions varies depending on the cache mode used.

In the copy–back mode, stores are done to the cache line. If the cache line is not in the cache, 20 cycles are required to prefill the line. If the cache line being overwritten was marked as modified, it is written in a write buffer, then transferred to physical memory in the background occupying the Mbus for 16 cycles. Once the line is in cache, single precision stores (`st`) are done in 3 cycles, double–precision stores (`std`) in 4 cycles.

In the write–through mode, stores are written to the cache line (the cache line is invalidated if the right block is not present) and then written into physical memory (as a one or two–word Mbus transfer). The IU takes the same 3 cycles (single) or 4 cycles (double) to execute the store instruction, which places the data into a Write Buffer. The Write Buffer acts as a queue of length 4 doublewords. When it is non–empty, and the accelerator chip is not busy, it will empty data into physical memory in 4 cycles (double) or 7 cycles (single).

The IU executes a single precision load instruction (`ld`) in 2 cycles when the word fetched is in cache. Double precision loads (`ldd`) complete in 3 cycles. If the value is not cached, an additional 20 cycles are required to fetch the cache line. Such cache line fills will wait until the write buffer is empty before starting.

For every access to physical memory, if the access is to a different DRAM page (pages are aligned and 8Kb in size), an additional 5 cycles is added to the access time.

10.2.2 SPARC Parallel Memory Access (dpld, dpst)

At certain times, the IU needs to access Parallel Memory. This is generally (and hopefully) fairly rare, since most access to the parallel memory is expected to be performed by the VUs.

Loads from Parallel memory to the IU register take 2 cycles (single) or 3 cycles (double) plus 12 cycles from the even banks (VU 0 and VU 2) or 13 cycles from the odd bank (VU 1 or VU 3), regardless of size. If reads are done in a streaming fashion, this will move data in 15 or 16 cycles per doubleword (about 16 Mbytes/sec).

Writes from IU registers to parallel memory work as in the write–through cached mode, except are not written to the cache line (and do not invalidate the cache line). Thus, they take 3 cycles (single) or 4 cycles (double) plus the time for the actual store to be done. If the write is to multiple VU banks (”broadcast”), an additional 4 cycles is required. If writes are done in a streaming fashion this will move data in 8 cycles per doubleword (32Mbytes per second).

10.2.3 SPARC Register File Access (dprd,dpwrt)

Transfers between the IU and VU Registers are uncached memory transfers, except that the Mbus times are shorter.

Loads from VU registers to an IU register (single or double via **`ld`** or **`ldd`** instructions) take 2 (single) or 3 (double) IU cycles plus 7 Mbus cycles.

Writes from IU registers to VU registers take 3 (single) or 4 (double) IU cycles to write to an write buffer, then 3 cycles (single or double) Mbus time in the background. An additional 4 cycles of Mbus time is required if the write is to more than 1 VU (broadcast).

The code generated by dpas for the DPEAC **`dpwrt`** and **`dprd`** instructions includes 2 instructions to build a pointer to the register specified in the first operand of the instruction. This results in 2 additional IU cycles before the load or store is begun.

It is also important to remember that DPEAC automatically inserts a dpsync instruction before all **`dpwrt`** or **`dprd`** instructions. This will consume as many as 23 Mbus cycles (see previous section). This action can be annulled by the use of the **`nosync`** operand.

10.2.4 SPARC Mode Register Access (dpset, dpget, dpsync)

The accelerator chip internally synchronizes the reading and writing of its control registers with its instruction stream. It does this in a fairly primitive way, namely, by stalling for 23 cycles from the start of the pipeline for the last element of the last Arithmetic operation. Aside from this stall, these operations perform exactly like the VU Register read or write operations.

Like **`dpwrt`** and **`dprd`**, the code generated by **`dpas`** for the DPEAC **`dpset`** and **`dpget`** instructions includes 2 instructions to build a pointer to the register specified in the first operand of the instruction. This results in 2 additional IU cycles before the load or store is begun.

The **`dpsync`** operation is nothing more than a write to a dummy control register, thus taking advantage of the internal synchronization.

10.2.5 Basic Instruction Timing

As a general rule, VU Instructions of vector length *N* execute in 2*N* cycles. Instructions executed one after another will generally not be executed with any delay between them, i.e. 2 instructions of vector length *N* will commonly execute in 4*N* cycles. There are exceptions to this rule, as follows:

Pipeline Bubbles

Some combinations of instructions cannot be executed without breaking the pipeline. As a result, 'bubbles' (skipped cycles) are inserted between them in the pipeline. The following tables lists typical cases. This is not meant to be precise, but gives an idea of where bubbles occur:

Case	Bubble Size
Before LDVM Instructions	7 cycles
After LDVM Instructions	6 cycles
Before STVM Instructions	7 cycles
After STVM Instructions	5–8 cycles
Load or Triadic (3–operand) Instruction followed by single store or EPC	1 cycle
Load or Triadic (3–operand) Instruction followed by double store	6 cycle

Case	Bubble Size
Single Store Instruction followed by any Load, Store or Triadic	5 cycles
EPC followed by any Indirect Load, Store or by Triadic	7 cycles
EPC followed by any Register Indirect	1 cycle
EPC followed by any Double Store	5 cycles

Microcoded Instructions

These operations do not execute one–per–cycle as do the other operations. Specifically, for a vector length of N, these instructions will take $2N*k$ Mbus (SPARC) cycles to execute, where k is taken from the following table:

Operation	*k*	Operation	*k*
fdiv{**v**,**s**}	4	**dfdiv**{**v**,**s**}	5
finv{**v**,**s**}	4	**dfinv**{**v**,**s**}	5
fsqrt{**v**,**s**}	6	**dfsqrt**{**v**,**s**}	8
fisqt{**v**,**s**}	5	**dfisqt**{**v**,**s**}	7

Single–Precision Stores

VU Instructions that perform single precision store operations (**ustore**{**v**,**s**}, **fstore**{**v**,**s**}, **istore**{**v**,**s**}) execute more slowly because of the necessity of doing a read–modify–write sequence on the 64–bit wide memory. As a result, instructions of vector length N that include single precision store operations take $7N$ cycles, rather than the standard $2N$ cycles.

DRAM Page Faults

Memory accesses by the accelerator chip when doing memory operations (loads and stores) are subject to DRAM page misses, just like IU memory references are. Every access to a word or doubleword, during or between VU instructions which access memory on a different DRAM page from that last accessed will stall the accelerator chip for 5 cycles. DRAM pages are 8K bytes in size and are aligned on 8K byte boundaries.

10.2.6 Avoiding Lost Performance from DPAS's Hazard Prevention

There are situations where the **dpas** assembler corrects for certain common pipeline hazards by inserting padding before instructions, essentially creating an artificial bubble between them and the instrcutions preceding. Because **dpas** does not do flow analysis of the code, it cannot determine what the previous instruction actually was, and therefore must be conservative in doing such padding. The following are instances where this occurs:

1. Single precision store operations read the register to be stored 5 cycles prior to storing and double precision store operations read the register 7 cycles before storing. Since both are less than the 4 cycle maximum "safe" latency, a hazard can occur between an instruction that writes a vector of registers and one that subsequently stores the same register to memory (a very common case). This is Hazard 2 in the section on pipeline hazards. As a result, **dpas** inserts an **fnops** instruction before the store. This represents a pipeline bubble of 8 Mbus cycles. If you are sure that such a hazard will not occur, adding the modifier **nopad** to the store instruction will remove the padding.

2. The other case of Hazard 2 occurs when an instruction does memory indirection. In particular, the register used as the index (*Ria*) is read 7 cycles before the memory operation is actually performed. This 7–cycle latency can hazard if the previous instruction writes the vector of index registers. As a result, **dpas** inserts an **fnops** instruction that can similarly be avoided with the **nopad** modifier.

3. In either of the above two cases, if the second operation) uses either the **vmnew** or **vmnop** modifier, the padding cannot be done with an **fnops** since it has the side–effect of shifting the Vector Mask by one bit. In such a case, **dpas** will pad by inserting a **dpsync** instruction. This will cause a very large bubble in the pipline (typically about 23 cycles).

4. The **dprd**[**d**] or **dpwrt**[**d**] instructions (which transfer between the memory–mapped data register file and IU registers) are not synchronized by the hardware. This is described in more detail in the Pipeline Hazards section as hazards 4 and 5. To prevent these hazards **dpas** inserts a **dpsync** instruction before each of these instructions. This represents an enormous pipeline bubble when the register access follows a VU instruction. Note that subsequent **dprd** or **dpwrt** instructions do not cause additional padding – see the previous discussion of **dpsync** for details.

10.3 Data Movement Algorithms

The following sections discuss algorithms for moving data on and between accelerator chips. Algorithms and their speeds are given.

10.3.1 On–VU Moves

The most common method for moving data within a single VU bank uses loops of vectored loads and stores. The following algorithm is typical and shows the movement of double precision data:

```
! %i0 point ot detsination (an instruction–space address)
! %i1 points to source
! %i2 contains the count of double words to be moved
! %l0 used to compute stride between 16 element (16 * 8)

    memnop=16; vmmode:=always    ! length=16, vmmode=”always”
    mov     16*8,%l0             ! calculate ”band stride”
    tst     %i2                  ! initial test of count left
    ble     done
    nop
band_loop:
    cmp     %i2,16               ! full band of 16 left?
    bge     1f                   ! yes, keep vector length 16
    nop
    memnop=%i2                   ! no, shorten to len of tail
1:  duloadv [%i1]:8,V2
    dustorev [%i0]:8,V2          ! write result (see below)
    add     %i0,%l0,%i0          ! move dest ptr to next band
    subcc   %i2,16,%i2           ! dec count (and test)
    bgt     band_loop            ! jump if any left to do
    add     %i1,%l0,%i1          ! move srce ptr to next band
done:
```

The basic technique of looping through vector operations to operate on a larger vector is known as *strip–mining*. The technique can be appropriately modified to handle conditionalized transfers and for striding of memory elements in either the source or destination by other than 8.

By using the vector length of 16, all of the IU instructions within the strip mine loop will be executed while the accelerator chip is busy performing the long vector operations, thus getting full utilization of the chip.

As a result, this algorithm will move a 16 values in $2N+2N$ cycles (the sum of the durations of the **duloadv** and **dustorev** instuctions), or 64 cycles. At 32 Mhz this works out to 64 MBytes/sec on each VU, or an aggregate 256 Mbytes/sec when all 4 VU's are transferring in parallel.

The situation is different for single precision because a store instruction takes $7N$ cycles rather than $3N$. In this case the algorithm moves 16 values in $2N+7N$ or 144 cycles for a bandwith of 14 Mbytes/sec per VU. As a result, it may be desireable when handling single–precision data to attempt to handle them in pairs. This introduces some corner cases, e.g. when the block being moved does not start on a doubleword boundary, when there are an odd number of words to move, when the data is not contiguous, and so forth.

Note: In the code above, the **dustorev** will hazard for vector lengths less than 5. However, the **dpas** assembler automatically inserts an **fnops** instruction before it to prevent the hazard. If the vector length is known never to be less than 5 in the tail of the strip–mine loop, this can be prevented by adding the **pad:4** modifier to that instruction, saving 8 Mbus cycles during each loop pass.

10.3.2 10.2.2 On–chip Moves

The exchange modifier offered by the Mode Set instruction format provides the most efficient way of moving data between the two VU banks controlled by a single accelerator chip. The code for doing the strip–mined exchange of two vectors, one on each VU bank of a accelerator chip is the same as the on–VU move, except that the load/store in the center of the loop becomes:

```
1:  dumovev V2,V6; duloadv [%i1]:8,V2; exchange
    dustorev[%i0]:8,V6
```

The performance is identical to the on–VU (non–exchanging) case.

Note: The current implementation requires that the source and destination registers in the **dumovev** (here V2 and V6) be different when doing a chain–load into an operation with **exchange**.

10.3.3 Off–chip Moves

The technique for moving a vector of data from one VU bank to another when the two VUs are not on the same physical accelerator chip requires doing loads and stores from the IU processor. The best known algorithm uses the registers of the FPU as an intermediate buffer. This prevents having to flip back and forth frequently between loads and stores. Such flipping interjects additional cycles, because the write buffer must empty before a load can be performed. Furthermore, it is faster to load from the memory–mapped accelerator register file than to read from the uncached memory (Mbus transfer is 7 cycles as opposed to 13).

As a result, the optimal method is to load the source vector into the register file first using a **`uloadv`** or **`duloadv`** instruction. Here again, bandwidth is doubled when double precision loads can be performed. Also, as before, it will be faster to store doublewords to memory rather than singlewords due to the read–modify–write degradation. It is not, however, faster to write the register file than it is to write memory directly, so using it as a buffer in the store direction is not beneficial. The double precision algorithm follows:

```
offdm_move_double_aligned:

#include <cmsys/dp.h>

!Move vector from VU 0 to VU 2

!registers assumed set up:
!   %i0: destination address (VU 0 data-space address)
!   %i1: destination stride in bytes (a multiple of 8)
!   %i2: source address (all-VUs instruction-space address)
!   %i3: source stride in bytes (a multiple of 8)
!   %i5: element count (vector length) >= 0

!Register used:
!   %l2: source band stride (16 times source increment)
!   %l3: ptr to memory mapped V2 on source chip, first VU

    set  DPV_DATA_REGS_0+(4*16),%l3  ! point to V2 on srce VU
    sll  %i3,4,%l2                  ! calc srce stride (16*inc)
    tst  %i5                        ! do any at all?
    ble  done
    nop

! Band Loop
loop:
    duloadv*=16 [%i2],V2   ! Load 16 values into V2 (all VU's)
    add       %i2,%l2,%i2   ! advance source band ptr
! copy from V2 in VU 0 to memory of VU 2 (or vice versa),
! and from VU 1 to memory of VU 3 (or vice versa)
```

```
dpsync
dpldd   [%l3],%f0          ! load from first VU (0 or 2)
dpldd   [%l3+8],%f2
dpldd   [%l3+16],%f4
dpldd   [%l3+24],%f6
dpldd   [%l3+32],%f8
dpldd   [%l3+40],%f10
dpldd   [%l3+48],%f12
dpldd   [%l3+56],%f14
dpldd   [%l3+64],%f16
dpldd   [%l3+72],%f18
dpldd   [%l3+80],%f20
dpldd   [%l3+88],%f22
dpldd   [%l3+96],%f24
dpldd   [%l3+104],%f26
dpldd   [%l3+112],%f28
dpldd   [%l3+120],%f30
dpstd   %f0,[%i0           ! store to first VU on dest
add     %i0,%i1,%i0
dpstd   %f2,[%i0]
add     %i0,%i1,%i0
dpstd   %f4,[%i0]
add     %i0,%i1,%i0
dpstd   %f6,[%i0]
add     %i0,%i1,%i0
dpstd   %f8,[%i0]
add     %i0,%i1,%i0
dpstd   %f10,[%i0]
add     %i0,%i1,%i0
dpstd   %f12,[%i0]
add     %i0,%i1,%i0
dpstd   %f14,[%i0]
add     %i0,%i1,%i0
dpstd   %f16,[%i0]
add     %i0,%i1,%i0
dpstd   %f18,[%i0]
add     %i0,%i1,%i0
dpstd   %f20,[%i0]
add     %i0,%i1,%i0
dpstd   %f22,[%i0]
add     %i0,%i1,%i0
dpstd   %f24,[%i0]
add     %i0,%i1,%i0
dpstd   %f26,[%i0]
add     %i0,%i1,%i0
dpstd   %f28,[%i0]
add     %i0,%i1,%i0
dpstd   %f30,[%i0]
sub     %i5,16,%i5       ! decrmement count by 16 elements
```

```
        cmp     %i5,16              ! band of 16 words left to do?
        bg      loop                ! branch to do more if so
        add   %i0,%i1,%i0        ! (last dest ptr inc moved here)

    ! Do tail values (one at a time). Note that one could imagine
    ! a faster method, such as branching into arbitrary spots
    ! in the code above. (Not that this is recommended, mind you!)

    tail:
        cmp     %i5,0
        ble     done
        nop
        duloadv*=%i5 [%i2],V2    ! Load remaining values
                                 ! into V2 (all VU's)
        dpsync
        mov     %l3,%o4             ! copy address of V2
                                 ! on source chip, 1st VU
    tail_loop:
        dpldd   [%o4],%f0           ! pick up value from V2
        dpstd   %f0,[%i0]           ! store it out
        add     %o4,8,%o4           ! advance source ptr
        deccc   %i5                 ! dec counter
        bnz     tail_loop           ! do more
        add   %i0,%i1,%i0        ! advance dest ptr
    done:
```

Disregarding the effect of the tail, this algorithm moves data in 18 Mbus cycles per element, 10 to read from the register (3 IU plus 7 Mbus), plus 8 for the store to memory (4 IU plus 4 Mbus). This will yield a peak bandwidth of about 14 Mbytes/sec. For single precision, an additional 3 cycles per element are required in the store for a total of 21 cycles or 12 Mbytes/sec.

10.4 Single Precision Memory Data Issues

The accelerator chip operates using 64–bit wide memory data buses to its memory. As a result, keep the following in mind when programming using single precision (32–bit) data, i.e. data of types **`i`**, **`u`**, or **`f`**:

First, due to the 64–bit wide memory, store operations of single precision data, such as **`fstorev`**, **`istores`**, etc., must be done by the hardware using a read–modify–write operation. This is three times slower than the simple write used for double precision (64–bit) data.

As a result, the DPEAC programmer may wish to store single precision data from the registers to memory in pairs, i.e. two values at a time. Accordingly, the real and imaginary halves of single precision complex numbers may best be handled together in aligned pairs. It may be advantageous to pad single precision data out to 64 bits, wasting memory, but allowing the numbers to be handled singly without loss of performance. Note that in the latter cases (complex and padded), the values stored in the registers will be stored at strides of 2 in the register file, which can restrict the choice of instruction formats.

Second, the **dpas** assembler must generate slightly less efficient code for long format DPEAC instructions that perform memory operations on single precision data that may not be double–word aligned. The programmer has two ways of preventing this. First, on such instructions, the **align** modifier may be added to indicate that the memory address used is guaranteed to be doubleword–aligned. For example:

```
faddv v1,0f1.0,v3; floadv [%i3],v1; align
```

tells **dpas** that the address in **%i3** is doubleword aligned (the use of the immediate value forces a long format (SI) instruction).

If all such addresses will be doubleword–aligned within the assembly of a file, **dpas** can be given the **-a** switch, which causes it to assume such alignment as the default. In this case, the **noalign** modifier can be used on instructions with memory addresses that may not be doubleword–aligned.

10.5 Tips, Tricks, And Algorithms

10.5.1 Avoiding dpset

The performance of control register writes is, frankly, awful (up to 30 cycles). Thus, it is highly recommended that the **dpset** instruction be avoided in favor of setting the operating modes of the accelerator via the "sticky–setting" modifiers of the instruction set. For example, Memory Stride mode is much more quickly set by an instruction in Memory Stride format than it is using **dpset**. Even if it is impossible to stick the setting of the mode on an existing instruction, it is always better to stick it onto a **NOP** instruction than to use **dpset**.

The following DPEAC macros illustrate the techniques for setting the common modes in this fashion. (These macros are defined for you in the include file **<cmsys/dpeac.h>**.)

```
/* Macros for setting DASH control registers
   the fast way (not via DPSET!)
   In these macros, args are as follows:
     vlen: Vector len as constant or IU reg
           (with "<" appended if decremented and
            shifted to bit 19);
     rs1_stride: a constant (expression)
     vmmode:  one of "always", "condmem",
              "condalu" or "cond"
     mem_stride:
              constant (expression) or IU reg. */

#macro set_mem_stride mem_stride
    memnop []:={mem_stride}
#endmacro

#macro set_vmmode vmmode
    memnop; vmmode:={vmmode}
#endmacro

#macro set_vector_length_and_vmmode vlen, vmmode
    memnop={vlen}; vmmode:={vmmode}
#endmacro

#macro set_vector_length_and_rs1_stride vlen, rs1_stride
    uors={vlen S2={rs1_stride},S2,S2
#endmacro

#macro set_vector_length_and_rs1_stride_and_vmmode \
                             vlen, rs1_stride, vmmode
    uors={vlen S2={rs1_stride},S2,S2; vmmode:={vmmode}
#endmacro

#macro set_vector_length vlen
    memnop={vlen}
#endmacro
```

10.5.2 Using Unpacked Masks

In the data parallel runtime systems, such as used by CM FORTRAN, user masks are often given in unpacked form in memory. This format represents the flag for each element as a full word of all one bits (**TRUE**) or all zero bits (**FALSE**). The technique for directly loading an unpacked mask into the low N bits of the vector mask (*N* = vector length) follows (**%i0** points to the packed mask in memory):

```
uloadv  [%i0],V2; ueqv V2,0; vmcurrent
```

The technique for converting the low *N* bits of the Vector Mask Buffer into an unpacked mask takes advantage of the **iaddc** operation which performs an add of two operands and a carry bit. The carry is taken from the vector mask buffer, reulsting in the following instruction sequence (the 16–bit packed mask is assumed to be in register S2):

```
imoves  –1,S4         ! Put a –1 in S4
                     !   (this can be outside the loop)
unots   S2,S2         ! invert the mask
ldvm    S2            ! load the mask
iaddcv  S4,0,V2       ! flag = 0+-1+(~vm bit)
```

10.5.3 Reduction

An interesting technique that takes advantage of the 2–cycle latency of the 2nd ALU operand allows for an optimal reduction method. Consider the problem of adding the 16 values in **V2** (**R16** through **R31**). Because the 2nd ALU operand is read just two cycles before the result is stored, the following will reduce the value to two sums:

```
faddv V2,R8:64,R8:64
fadds R72,R8,R8
```

Notice that the second operand is strided by half of the register file size. As a result, the vector of second operands is taken from **R2**, **R66**, **R2**, **R66**, etc. And these are sequentially replaced by the two running totals. Following this, it is only necessary to add the two totals into the aggregate reduced total. Refer to Section 9.1.1 for a list of operands that have this two–cycle latency. All other arithmetic operands have the standard 4–cycle latency. For them, the same trick can be used, but using four running reduction results and register strides of 1/4 of the register file size (32).

10.5.4 List Compression Algorithm

The following algorithm reads 16 data values from a vector in memory, and using a 16–bit mask, stores only the elements for which the coresponding bit in the mask is set. The resulting destination vector contains only the selected elements packed together. i.e. with the deselected elements compressed out:

```
! %i0 = pointer to 16 memory elements

! 16-bit mask selecting valid elements
!   is in the Vector Mask Buffer

! S2 contains stride between elements
!   in the destination vector

    uloadv  [%i0],V2; vmcount=V14:1
              !add scan the context bits

    umsbiv  S2:0,S2:0,V14
              !(indices = counts * stride) - stride

    ustorev [%i1](V14),V2
```