B Vector Processors

I'm certainly not inventing vector processors. There are three kinds that I know of existing today. They are represented by the Illiac-IV, the (CDC) Star processor, and the TI (ASC) processor. Those three were all pioneering processors.... One of the problems of being a pioneer is you always make mistakes and I never, never want to be a pioneer. It'salways best to come second when you can look at the mistakes the pioneers made.

Seymour Cray
Public Lecture at Lawrence Livermore Laboratories on the Introduction of the CRAY-1 (1976)

B.1	Why Vector Processors?	B-1
B.2	Basic Vector Architecture	B-3
B.3	Two Real-World Issues: Vector Length and Stride	B-15
B.4	Effectiveness of Compiler Vectorization	B-22
B.5	Enhancing Vector Performance	B-23
B.6	Putting It All Together: Performance of Vector Processors	B-29
B.7	Fallacies and Pitfalls	B-35
B.8	Concluding Remarks	B-37
B.9	Historical Perspective and References	B-38
	Exercises	B-43

B.1 Why Vector Processors?

In Chapters 3 and 4 we looked at pipelining and exploitation of instruction-level parallelism in detail and saw that pipeline scheduling, issuing multiple instructions per clock cycle, and more deeply pipelining a processor could significantly improve the performance of a processor. (This appendix assumes that you have read Chapter 3 completely and at least skimmed Chapter 4; in addition, the discussion on vector memory systems assumes that you have read Chapter 5.) Yet there are limits on the performance improvement that pipelining can achieve. These limits are set by two primary factors:

- *Clock cycle time*—The clock cycle time can be decreased by making the pipelines deeper, but a deeper pipeline will increase the pipeline dependences and result in a higher CPI. At some point, each increase in pipeline depth has a corresponding increase in CPI. As we saw in Chapter 3's *Fallacies and Pitfalls,* very deep pipelining can slow down a processor.
- *Instruction fetch and decode rate*—This obstacle, sometimes called the *Flynn bottleneck* (based on Flynn [1966]), makes it difficult to fetch and issue many

instructions per clock. This obstacle is one reason that it has been difficult to build processors with high clock rates and very high issue rates.

The dual limitations imposed by deeper pipelines and issuing multiple instructions can be viewed from the standpoint of either clock rate or CPI: It is just as difficult to schedule a pipeline that is *n* times deeper as it is to schedule a processor that issues *n* instructions per clock cycle.

High-speed, pipelined processors are particularly useful for large scientific and engineering applications. A high-speed pipelined processor will usually use a cache to avoid forcing memory reference instructions to have very long latency. Unfortunately, big, long-running, scientific programs often have very large active data sets that are sometimes accessed with low locality, yielding poor performance from the memory hierarchy. This problem could be overcome by not caching these structures if it were possible to determine the memory-access patterns and pipeline the memory accesses efficiently. Novel cache architectures and compiler assistance through blocking and prefetching are decreasing these memory hierarchy problems, but they continue to be serious in some applications.

Vector processors provide high-level operations that work on *vectors*—linear arrays of numbers. A typical vector operation might add two 64-element, floating-point vectors to obtain a single 64-element vector result. The vector instruction is equivalent to an entire loop, with each iteration computing one of the 64 elements of the result, updating the indices, and branching back to the beginning.

Vector instructions have several important properties that solve most of the problems mentioned above:

- The computation of each result is independent of the computation of previous results, allowing a very deep pipeline *without* generating any data hazards. Essentially, the absence of data hazards was determined by the compiler or by the programmer when she decided that a vector instruction could be used.
- A single vector instruction specifies a great deal of work—it is equivalent to executing an entire loop. Thus, the instruction bandwidth requirement is reduced, and the Flynn bottleneck is considerably mitigated.
- Vector instructions that access memory have a known access pattern. If the vector's elements are all adjacent, then fetching the vector from a set of heavily interleaved memory banks works very well (as we saw in section 5.6). The high latency of initiating a main memory access versus accessing a cache is amortized, because a single access is initiated for the entire vector rather than to a single word. Thus, the cost of the latency to main memory is seen only once for the entire vector, rather than once for each word of the vector.
- Because an entire loop is replaced by a vector instruction whose behavior is predetermined, control hazards that would normally arise from the loop branch are nonexistent.

For these reasons, vector operations can be made faster than a sequence of scalar operations on the same number of data items, and designers are motivated to include vector units if the applications domain can use them frequently.

As mentioned above, vector processors pipeline the operations on the individual elements of a vector. The pipeline includes not only the arithmetic operations (multiplication, addition, and so on), but also memory accesses and effective address calculations. In addition, most high-end vector processors allow multiple vector operations to be done at the same time, creating parallelism among the operations on different elements. In this appendix, we focus on vector processors that gain performance by pipelining and instruction overlap.

B.2 Basic Vector Architecture

A vector processor typically consists of an ordinary pipelined scalar unit plus a vector unit. All functional units within the vector unit have a latency of several clock cycles. This allows a shorter clock cycle time and is compatible with long-running vector operations that can be deeply pipelined without generating hazards. Most vector processors allow the vectors to be dealt with as floating-point numbers, as integers, or as logical data. Here we will focus on floating point. The scalar unit is basically no different from the type of advanced pipelined CPU discussed in Chapter 3.

There are two primary types of architectures for vector processors: *vector-register processors* and *memory-memory vector processors*. In a vector-register processor, all vector operations—except load and store—are among the vector registers. These architectures are the vector counterpart of a load-store architecture. All major vector computers shipped since the late 1980s use a vector-register architecture; these include the Cray Research processors (CRAY-1, CRAY-2, X-MP, Y-MP, and C-90), the Japanese supercomputers (NEC SX/2 and SX/3, Fujitsu VP200 and VP400, and the Hitachi S820), as well as the mini-supercomputers (Convex C-1 and C-2). In a memory-memory vector processor, all vector operations are memory to memory. The first vector computers were of this type, as were CDC's vector computers. From this point on we will focus on vector-register architectures only; we will briefly return to memory-memory vector architectures at the end of the appendix (section B.7) to discuss why they have not been as successful as vector-register architectures.

We begin with a vector-register processor consisting of the primary components shown in Figure B.1. This processor, which is loosely based on the CRAY-1, is the foundation for discussion throughout most of this appendix. We will call it DLXV; its integer portion is DLX, and its vector portion is the logical vector extension of DLX. The rest of this section examines how the basic architecture of DLXV relates to other processors.

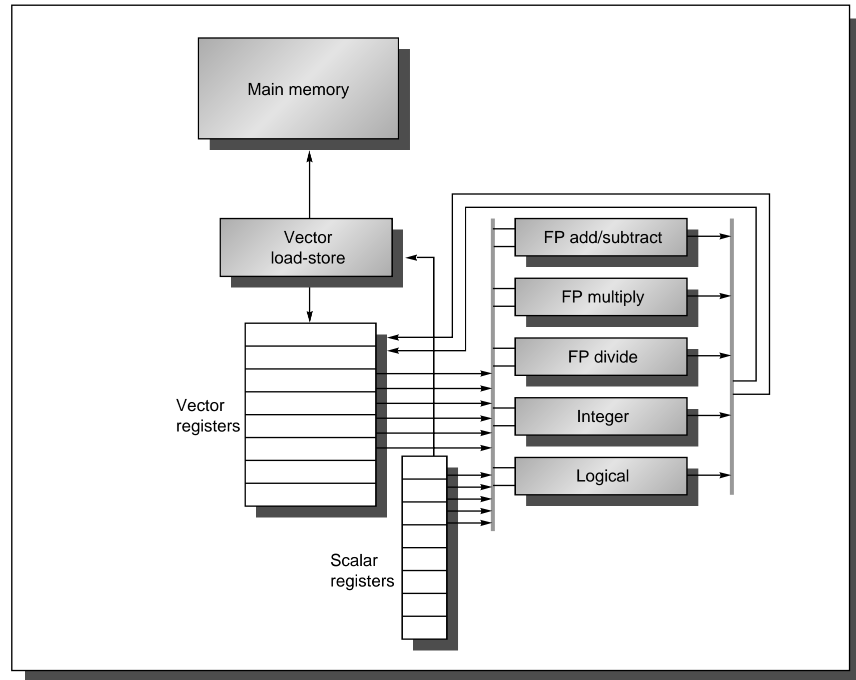

FIGURE B.1 The basic structure of a vector-register architecture, DLXV. This processor has a scalar architecture just like DLX. There are also eight 64-element vector registers, and all the functional units are vector functional units. Special vector instructions are defined both for arithmetic and for memory accesses. We show vector units for logical and integer operations. These are included so that DLXV looks like a standard vector processor, which usually includes these units. However, we will not be discussing these units except in the Exercises. The vector and scalar registers have a significant number of read and write ports to allow multiple simultaneous vector operations. These ports are connected to the inputs and outputs of the vector functional units by a set of crossbars (shown in thick gray lines). In section B.5 we add chaining, which will require additional interconnect capability.

The primary components of the instruction set architecture of DLXV are

- *Vector registers*—Each vector register is a fixed-length bank holding a single vector. DLXV has eight vector registers, and each vector register holds 64 elements. Each vector register must have at least two read ports and one write port in DLXV. This will allow a high degree of overlap among vector operations to different vector registers. (We do not consider the problem of a shortage of vector register ports. In real machines this would result in a structural hazard.) The read and write ports, which total at least 16 read ports and eight write ports, are connected to the functional unit inputs or outputs by a pair of crossbars. (The CRAY-1 manages to implement the register file with only a single port per register using some clever implementation techniques.)

- *Vector functional units*—Each unit is fully pipelined and can start a new operation on every clock cycle. A control unit is needed to detect hazards, both from conflicts for the functional units (structural hazards) and from conflicts for register accesses (data hazards). DLXV has five functional units, as shown in Figure B.1. For simplicity, we will focus exclusively on the floating-point functional units. Depending on the vector processor, scalar operations either use the vector functional units or use a dedicated set. We assume the functional units are shared, but again, for simplicity, we ignore potential conflicts.
- *Vector load-store unit*—This is a vector memory unit that loads or stores a vector to or from memory. The DLXV vector loads and stores are fully pipelined, so that words can be moved between the vector registers and memory with a bandwidth of one word per clock cycle, after an initial latency. This unit would also normally handle scalar loads and stores.
- *A set of scalar registers*—Scalar registers can also provide data as input to the vector functional units, as well as compute addresses to pass to the vector load-store unit. These are the normal 32 general-purpose registers and 32 floating-point registers of DLX, though more read and write ports are needed. The scalar registers are also connected to the functional units by the pair of crossbars.

Figure B.2 shows the characteristics of some typical vector processors, including the size and count of the registers, the number and types of functional units, and the number of load-store units.

In DLXV, vector operations use the same names as DLX operations, but with the letter "V" appended. These are double-precision, floating-point vector operations. (We have omitted single-precision FP operations and integer and logical operations for simplicity.) Thus, `ADDV` is an add of two double-precision vectors. The vector instructions take as their input either a pair of vector registers (`ADDV`) or a vector register and a scalar register, designated by appending "SV" (`ADDSV`). In the latter case, the value in the scalar register is used as the input for all operations—the operation `ADDSV` will add the contents of a scalar register to each element in a vector register. Most vector operations have a vector destination register, though a few (population count) produce a scalar value, which is stored to a scalar register. The names `LV` and `SV` denote vector load and vector store, and they load or store an entire vector of double-precision data. One operand is the vector register to be loaded or stored; the other operand, which is a DLX general-purpose register, is the starting address of the vector in memory. Figure B.3 lists the DLXV vector instructions. In addition to the vector registers, we need two additional special-purpose registers: the vector-length and vector-mask registers. We will discuss these registers and their purpose in sections B.3 and B.5, respectively.

Processor	Year announced	Clock rate (MHz)	Registers	Elements per register (64-bit elements)	Functional units	Load-store units
CRAY-1	1976	80	8	64	6: add, multiply, reciprocal, integer add, logical, shift	1
CRAY X-MP CRAY Y-MP	1983 1988	120 166	8	64	8: FP add, FP multiply, FP reciprocal, integer add, 2 logical, shift, population count/parity	2 loads 1 store
CRAY-2	1985	166	8	64	5: FP add, FP multiply, FP reciprocal/sqrt, integer (add shift, population count), logical	1
Fujitsu VP100/200	1982	133	8–256	32–1024	3: FP or integer add/logical, multiply, divide	2
Hitachi S810/820	1983	71	32	256	4: 2 integer add/logical, 1 multiply-add, and 1 multiply/divide–add unit	4
Convex C-1	1985	10	8	128	4: multiply, add, divide, integer/logical	1
NEC SX/2	1984	160	8 + 8192	256 variable	16: 4 integer add/logical, 4 FP multiply/divide, 4 FP add, 4 shift	8
DLXV	1990	200	8	64	5: multiply, divide, add, integer add, logical	1
Cray C-90	1991	240	8	128	8: FP add, FP multiply, FP reciprocal, integer add, 2 logical, shift, population count/parity	4
Convex C-4	1994	135	16	128	3: each is full integer, logical, and FP (including multiply-add)	
NEC SX/4	1995	400	8 + 8192	256 variable	16: 4 integer add/logical, 4 FP multiply/divide, 4 FP add, 4 shift	8
Cray J-90	1995	100	8	64	4: FP add, FP multiply, FP reciprocal, integer/logical	
Cray T-90	1996	~500	8	128	8: FP add, FP multiply, FP reciprocal, integer add, 2 logical, shift, population count/parity	4

FIGURE B.2 Characteristics of several vector-register architectures. The vector functional units include all operation units used by the vector instructions. The functional units are floating point unless stated otherwise. If the processor is a multiprocessor, the entries correspond to the characteristics of one processor. Each vector load-store unit represents the ability to do an independent, overlapped transfer to or from the vector registers. The Fujitsu VP200's vector registers are configurable: The size and count of the 8 K 64-bit entries may be varied inversely to one another (e.g., eight registers each 1 K elements long, or 128 registers each 64 elements long). The NEC SX/2 has eight fixed registers of length 256, plus 8 K of configurable 64-bit registers. The reciprocal unit on the CRAY processors is used to do division (and square root on the CRAY-2). Add pipelines perform floating-point add and subtract. The multiply/divide–add unit on the Hitachi S810/820 performs an FP multiply or divide followed by an add or subtract (while the multiply-add unit performs a multiply followed by an add or subtract). Note that most processors use the vector FP multiply and divide units for vector integer multiply and divide, just like DLX, and several of the processors use the same units for FP scalar and FP vector operations. Several of the machines have different clock rates in the vector and scalar units; the clock rates shown are for the vector units.

Instruction	Operands	Function
ADDV ADDSV	V1,V2,V3 V1,F0,V2	Add elements of V2 and V3, then put each result in V1. Add F0 to each element of V2, then put each result in V1.
SUBV SUBVS SUBSV	V1,V2,V3 V1,V2,F0 V1,F0,V2	Subtract elements of V3 from V2, then put each result in V1. Subtract F0 from elements of V2, then put each result in V1. Subtract elements of V2 from F0, then put each result in V1.
MULTV MULTSV	V1,V2,V3 V1,F0,V2	Multiply elements of V2 and V3, then put each result in V1. Multiply F0 by each element of V2, then put each result in V1.
DIVV DIVVS DIVSV	V1,V2,V3 V1,V2,F0 V1,F0,V2	Divide elements of V2 by V3, then put each result in V1. Divide elements of V2 by F0, then put each result in V1. Divide F0 by elements of V2, then put each result in V1.
LV	V1,R1	Load vector register V1 from memory starting at address R1.
SV	R1,V1	Store vector register V1 into memory starting at address R1.
LVWS	V1,(R1,R2)	Load V1 from address at R1 with stride in R2, i.e., R1+i × R2.
SVWS	(R1,R2),V1	Store V1 from address at R1 with stride in R2, i.e., R1+i × R2.
LVI	V1,(R1+V2)	Load V1 with vector whose elements are at R1+V2(i), i.e., V2 is an index.
SVI	(R1+V2),V1	Store V1 to vector whose elements are at R1+V2(i), i.e., V2 is an index.
CVI	V1,R1	Create an index vector by storing the values 0, 1 × R1, 2 × R1,...,63 × R1 into V1.
S--V S--SV	V1,V2 F0,V1	Compare the elements (EQ, NE, GT, LT, GE, LE) in V1 and V2. If condition is true, put a 1 in the corresponding bit vector; otherwise put 0. Put resulting bit vector in vector-mask register (VM). The instruction S--SV performs the same compare but using a scalar value as one operand.
POP	R1,VM	Count the 1s in the vector-mask register and store count in R1.
CVM		Set the vector-mask register to all 1s.
MOVI2S MOVS2I	VLR,R1 R1,VLR	Move contents of R1 to the vector-length register. Move the contents of the vector-length register to R1.
MOVF2S MOVS2F	VM,F0 F0,VM	Move contents of F0 to the vector-mask register. Move contents of vector-mask register to F0.

FIGURE B.3 The DLXV vector instructions. Only the double-precision FP operations are shown. In addition to the vector registers, there are two special registers, VLR (discussed in section B.3) and VM (discussed in section B.5). The operations with stride are explained in section B.3, and the use of the index creation and indexed load-store operations are explained in section B.5.

A vector processor is best understood by looking at a vector loop on DLXV. Let's take a typical vector problem, which will be used throughout this appendix:

$$Y = a \times X + Y$$

X and Y are vectors, initially resident in memory, and a is a scalar. This is the so-called *SAXPY* or *DAXPY* loop that forms the inner loop of the Linpack benchmark. (SAXPY stands for <u>s</u>ingle-precision <u>a</u> × <u>X</u> <u>p</u>lus <u>Y</u>; DAXPY for <u>d</u>ouble-precision <u>a</u> × <u>X</u> <u>p</u>lus <u>Y</u>.) Linpack is a collection of linear algebra routines, and the

routines for performing Gaussian elimination constitute what is known as the Linpack benchmark. The DAXPY routine, which implements the above loop, represents a small fraction of the source code of the Linpack benchmark, but it accounts for most of the execution time for the benchmark.

For now, let us assume that the number of elements, or length, of a vector register (64) matches the length of the vector operation we are interested in. (This restriction will be lifted shortly.)

EXAMPLE Show the code for DLX and DLXV for the DAXPY loop. Assume that the starting addresses of X and Y are in Rx and Ry, respectively.

ANSWER Here is the DLX code.

```
       LD      F0,a
       ADDI    R4,Rx,#512    ;last address to load
Loop:  LD      F2,0(Rx)      ;load X(i)
       MULTD   F2,F0,F2      ;a × X(i)
       LD      F4,0(Ry)      ;load Y(i)
       ADDD    F4,F2,F4      ;a × X(i) + Y(i)
       SD      0(Ry),F4      ;store into Y(i)
       ADDI    Rx,Rx,#8      ;increment index to X
       ADDI    Ry,Ry,#8      ;increment index to Y
       SUB     R20,R4,Rx     ;compute bound
       BNEZ    R20,Loop      ;check if done
```

Here is the code for DLXV for DAXPY.

```
LD       F0,a          ;load scalar a
LV       V1,Rx         ;load vector X
MULTSV   V2,F0,V1      ;vector-scalar multiply
LV       V3,Ry         ;load vector Y
ADDV     V4,V2,V3      ;add
SV       Ry,V4         ;store the result
```

There are some interesting comparisons between the two code segments in this Example. The most dramatic is that the vector processor greatly reduces the dynamic instruction bandwidth, executing only six instructions versus almost 600 for DLX. This reduction occurs both because the vector operations work on 64 elements and because the overhead instructions that constitute nearly half the loop on DLX are not present in the DLXV code. ■

Another important difference is the frequency of pipeline interlocks. In the straightforward DLX code every ADDD must wait for a MULTD, and every SD must wait for the ADDD. On the vector processor, each vector instruction operates on all the vector elements independently. Thus, pipeline stalls are required only once per vector operation, rather than once per vector element. In this example, the pipeline-stall frequency on DLX will be about 64 times higher than it is on DLXV. The pipeline stalls can be eliminated on DLX by using software pipelining or loop unrolling (as we saw in Chapter 4). However, the large difference in instruction bandwidth cannot be reduced.

Vector Execution Time

The execution time of a sequence of vector operations primarily depends on three factors: the length of the vectors being operated on, structural hazards among the operations, and the data dependences. Given the vector length and the *initiation rate*, which is the rate at which a vector unit consumes new operands and produces new results, we can compute the time for a single vector instruction. The initiation rate is usually one per clock cycle for individual operations. However, some supercomputers have vector instructions that can produce two or more results per clock, and others have units that may not be fully pipelined. For simplicity, we assume that initiation rates are one throughout this appendix. Thus, the execution time for a single vector instruction is approximately the vector length.

To simplify the discussion of vector execution and its timing, we will use the notion of a *convoy*, which is the set of vector instructions that could potentially begin execution together in one clock period. (Although the concept of a convoy is used in vector compilers, no standard terminology exists. Hence, we created the term *convoy*.) The instructions in a convoy *must not* contain any structural or data hazards (though we will relax this later); if such hazards were present, the instructions in the potential convoy would need to be serialized and initiated in different convoys. To keep the analysis simple, we assume that a convoy of instructions must complete execution before any other instructions (scalar or vector) can begin execution. We will relax this in section B.6 by using a less restrictive, but more complex, method for issuing instructions.

Accompanying the notion of a convoy is a timing metric, called a *chime*, that can be used for estimating the performance of a vector sequence consisting of convoys. A chime is an approximate measure of execution time for a vector sequence; a chime measurement is independent of vector length. Thus, a vector sequence that consists of m convoys executes in m chimes, and for a vector length of n, this is approximately $m \times n$ clock cycles. A chime approximation ignores some processor-specific overheads, many of which are dependent on vector length. Hence, measuring time in chimes is a better approximation for long vectors. We will use the chime measurement, rather than clock cycles per result, to explicitly indicate that certain overheads are being ignored.

If we know the number of convoys in a vector sequence, we know the execution time in chimes. One source of overhead ignored in measuring chimes is any limitation on initiating multiple vector instructions in a clock cycle. If only one vector instruction can be initiated in a clock cycle (the reality in most vector processors), the chime count will underestimate the actual execution time of a convoy. Because the vector length is typically much greater than the number of instructions in the convoy, we will simply assume that the convoy executes in one chime.

EXAMPLE Show how the following code sequence lays out in convoys, assuming a single copy of each vector functional unit:

```
LV        V1,Rx        ;load vector X
MULTSV    V2,F0,V1     ;vector-scalar multiply
LV        V3,Ry        ;load vector Y
ADDV      V4,V2,V3     ;add
SV        Ry,V4        ;store the result
```

How many chimes will this vector sequence take? How many chimes per FLOP (floating-point operation) are needed?

ANSWER The first convoy is occupied by the first `LV` instruction. The `MULTSV` is dependent on the first `LV`, so it cannot be in the same convoy. The second `LV` instruction can be in the same convoy as the `MULTSV`. The `ADDV` is dependent on the second `LV`, so it must come in yet a third convoy, and finally the `SV` depends on the `ADDV`, so it must go in a following convoy. This leads to the following layout of vector instructions into convoys:

1. `LV`
2. `MULTSV` `LV`
3. `ADDV`
4. `SV`

The sequence requires four convoys and hence takes four chimes. Note that although we allow the `MULTSV` and the `LV` both to execute in convoy 2, most vector machines will take two clock cycles to initiate the instructions. Since the sequence takes a total of four chimes and there are two floating-point operations per result, the number of chimes per FLOP is two. ■

The chime approximation is reasonably accurate for long vectors. For example, for 64-element vectors, the time in chimes is four, so the sequence would take about 256 clock cycles. The overhead of issuing convoy 2 in two separate clocks would be small.

Another source of overhead is far more significant than the issue limitation. The most important source of overhead ignored by the chime model is vector *start-up time*. The start-up time comes from the pipelining latency of the vector operation and is principally determined by how deep the pipeline is for the functional unit used. The start-up time increases the effective time to execute a convoy to more than one chime. Because of our assumption that convoys do not overlap in time, the start-up time delays the execution of subsequent convoys. Of course the instructions in successive convoys have either structural conflicts for some functional unit or are data dependent, so the assumption of no overlap is reasonable. The actual time to complete a convoy is determined by the sum of the vector length and the start-up time. If vector lengths were infinite, this start-up overhead would be amortized, but finite vector lengths expose it, as the following Example shows.

EXAMPLE Assume the start-up overhead for functional units is shown in Figure B.4.

Unit	Start-up overhead
Load and store unit	12 cycles
Multiply unit	7 cycles
Add unit	6 cycles

FIGURE B.4 Start-up overhead.

Show the time that each convoy can begin and the total number of cycles needed. How does the time compare to the chime approximation for a vector of length 64?

ANSWER Figure B.5 provides the answer in convoys, assuming that the vector length is *n*:

Convoy	Starting time	First-result time	Last-result time
1. `LV`	0	12	$11 + n$
2. `MULTSV LV`	$12 + n$	$12 + n + 12$	$23 + 2n$
3. `ADDV`	$24 + 2n$	$24 + 2n + 6$	$29 + 3n$
4. `SV`	$30 + 3n$	$30 + 3n + 12$	$41 + 4n$

FIGURE B.5 Starting times and first- and last-result times for convoys 1 through 4. The vector length is *n*.

One tricky question is when we assume the vector sequence is done; this determines whether the start-up time of the `SV` is visible or not. We assume that the instructions following cannot fit in the same convoy, and we

have already assumed that convoys do not overlap. Thus the total time is given by the time until the last vector instruction in the last convoy completes. This is an approximation, and the start-up time of the last vector instruction may be seen in some sequences and not in others. For simplicity, we always include it.

The time per result for a vector of length 64 is 4 + (42/64) = 4.65 clock cycles, while the chime approximation would be 4. The execution time with start-up overhead is 1.16 times higher. ■

For simplicity, we will use the chime approximation for running time, incorporating start-up time effects only when we want more detailed performance or to illustrate the benefits of some enhancement. For long vectors, a typical situation, the overhead effect is not that large. Later in the appendix we will explore ways to reduce start-up overhead.

Start-up time for an instruction comes from the pipeline depth for the functional unit implementing that instruction. If the initiation rate is to be kept at one clock cycle per result, then

$$\text{Pipeline depth} = \left\lceil \frac{\text{Total functional unit time}}{\text{Clock cycle time}} \right\rceil$$

For example, if an operation takes 10 clock cycles, it must be pipelined 10 deep to achieve an initiation rate of one per clock cycle. Pipeline depth, then, is determined by the complexity of the operation and the clock cycle time of the processor. The pipeline depths of functional units vary widely—from two to 20 stages is not uncommon—though the most heavily used units have pipeline depths of four to eight clock cycles.

For DLXV, we will use the same pipeline depths as the CRAY-1, though more modern processors might have units with lower latency. All functional units are fully pipelined. As shown in Figure B.6, pipeline depths are six clock cycles for floating-point add and seven clock cycles for floating-point multiply. On DLXV, as on most vector processors, independent vector operations using different functional units can issue in the same convoy.

Operation	Start-up penalty
Vector add	6
Vector multiply	7
Vector divide	20
Vector load	12

FIGURE B.6 Start-up penalties on DLXV. These are the start-up penalties in clock cycles for DLXV vector operations.

Vector Load-Store Units and Vector Memory Systems

The behavior of the load-store vector unit is significantly more complicated than that of the arithmetic functional units. The start-up time for a load is the time to get the first word from memory into a register. If the rest of the vector can be supplied without stalling, then the vector initiation rate is equal to the rate at which new words are fetched or stored. Unlike simpler functional units, the initiation rate may not necessarily be one clock cycle.

Typically, penalties for start-ups on load-store units are higher than those for arithmetic functional units—up to 50 clock cycles on some processors. For DLXV we will assume a start-up time of 12 clock cycles; by comparison, the CRAY-1 and CRAY X-MP have load-store start-up times of between nine and 17 clock cycles. Figure B.6 summarizes the start-up penalties for DLXV vector operations.

To maintain an initiation rate of one word fetched or stored per clock, the memory system must be capable of producing or accepting this much data. This is usually done by creating multiple memory banks, as discussed in section 5.6. As we will see in the next section, having significant numbers of banks is useful for dealing with vector loads or stores that access rows or columns of data.

Most vector processors use memory banks rather than simple interleaving for two primary reasons:

1. Many vector computers support multiple loads or stores per clock. To support multiple simultaneous accesses, the memory system needs to have multiple banks and be able to control the addresses to the banks independently.
2. As we will see in the next section, many vector processors support the ability to load or store data words that are not sequential. In such cases, independent bank addressing, rather than interleaving, is required.

In Chapter 5 we saw that the desired access rate and the bank access time determined how many banks were needed to access a memory without a stall. The next Example shows how these timings work out in a vector processor.

EXAMPLE Suppose we want to fetch a vector of 64 elements starting at byte address 136, and a memory access takes six clocks. How many memory banks must we have? With what addresses are the banks accessed? When will the various elements arrive at the CPU?

ANSWER Six clocks per access require at least six banks, but because we want the number of banks to be a power of two, we choose to have eight banks. Figure B.7 shows what byte addresses each bank accesses within each time period. Remember that a bank begins a new access as soon as it has completed the old access.

Beginning at clock no.	Bank 0	1	2	3	4	5	6	7
0	192	136	144	152	160	168	176	184
6	256	200	208	216	224	232	240	248
14	320	264	272	280	288	296	304	312
22	384	328	336	344	352	360	368	376

FIGURE B.7 Memory addresses (in bytes) by bank number and time slot at which access begins. The exact time when a bank transmits its data is given by the address it accesses minus the starting address, divided by eight, plus the memory latency (six clocks). It is important to observe that bank 0 accesses a word in the next block (i.e., it accesses 192 rather than 128 and then 256 rather than 192, and so on). If bank 0 were to start at the lower address, we would require an extra cycle to transmit the data, and we would transmit one value unnecessarily. While this problem is not severe for this example, if we had 64 banks, up to 63 unnecessary clock cycles and transfers could occur. The fact that bank 0 does not access a word in the same block of eight distinguishes this type of memory system from interleaved memory. Normally, interleaved memory systems combine the bank address and the base starting address by concatenation rather than addition. Also, interleaved memories are almost always implemented with synchronized access. Memory banks require address latches for each bank, which are not normally needed in a system with only interleaving. This timing diagram is drawn as if all banks access in clock 0, clock 16, etc. In practice, since the bus allocations needed to return the words are staggered, the actual accesses are often staggered.

Figure B.8 shows the timing for the first few sets of accesses for an eight-bank system with a six-clock-cycle access latency. There are two important observations about Figures B.7 and B.8: First, notice that the exact address fetched by a bank is largely determined by the lower-order bits in the bank number; however, the initial access to a bank is always within eight double words of the starting address. Second, notice that once the initial latency is overcome (six clocks in this case), the pattern is to access a bank every *n* clock cycles, where *n* is the total number of banks (*n* = 8 in this case).

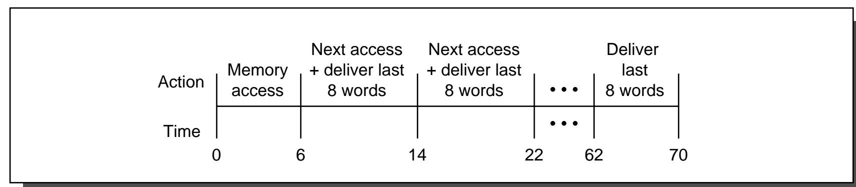

FIGURE B.8 Access timing for the first 64 double-precision words of the load. After the six-clock-cycle initial latency, eight double-precision words are returned every eight clock cycles.

■

The number of banks in the memory system and the pipeline depth in the functional units are essentially counterparts, since they determine the initiation rates for operations using these units. The processor cannot access a memory bank faster than the memory cycle time. Thus, if memory is built from DRAM, where the memory cycle time is about twice the access time, the processor needs twice as many banks as the above Example shows. For memory systems that support multiple simultaneous vector accesses or allow nonsequential accesses in vector loads or stores, the number of memory banks should be larger than the minimum, otherwise, memory bank conflicts will exist. We explore this in more detail in the next section.

B.3 Two Real-World Issues: Vector Length and Stride

This section deals with two issues that arise in real programs: What do you do when the vector length in a program is not exactly 64? How do you deal with nonadjacent elements in vectors that reside in memory? First, let's consider the issue of vector length.

Vector-Length Control

A vector-register processor has a natural vector length determined by the number of elements in each vector register. This length, which is 64 for DLXV, is unlikely to match the real vector length in a program. Moreover, in a real program the length of a particular vector operation is often unknown at compile time. In fact, a single piece of code may require different vector lengths. For example, consider this code:

```
        do 10 i = 1,n
10      Y(i) = a * X(i) + Y(i)
```

The size of all the vector operations depends on `n`, which may not even be known until runtime! The value of `n` might also be a parameter to a procedure containing the above loop and therefore be subject to change during execution.

The solution to these problems is to create a *vector-length register* (VLR). The VLR controls the length of any vector operation, including a vector load or store. The value in the VLR, however, cannot be greater than the length of the vector registers. This solves our problem as long as the real length is less than the *maximum vector length* (MVL) defined by the processor.

What if the value of `n` is not known at compile time, and thus may be greater than MVL? To tackle the second problem where the vector is longer than the maximum length, a technique called *strip mining* is used. Strip mining is the generation of code such that each vector operation is done for a size less than or

equal to the MVL. We could strip-mine the loop in the same manner that we unrolled loops in Chapter 4: Create one loop that handles any number of iterations that is a multiple of MVL and another loop that handles any remaining iterations, which must be less than MVL. In practice, compilers usually create a single strip-mined loop that is parameterized to handle both portions by changing the length. The strip-mined version of the DAXPY loop written in FORTRAN, the major language used for scientific applications, is shown with C-style comments:

```
          low = 1
          VL = (n mod MVL) /*find the odd size piece*/
          do 1 j = 0,(n / MVL) /*outer loop*/
               do 10 i = low, low+VL-1 /*runs for length VL*/
                    Y(i) = a*X(i) + Y(i) /*main operation*/
10             continue
               low = low+VL /*start of next vector*/
               VL = MVL /*reset the length to max*/
1         continue
```

The term n/MVL represents truncating integer division (which is what FORTRAN does) and is used throughout this section. The effect of this loop is to block the vector into segments which are then processed by the inner loop. The length of the first segment is (n mod MVL) and all subsequent segments are of length MVL. This is depicted in Figure B.9.

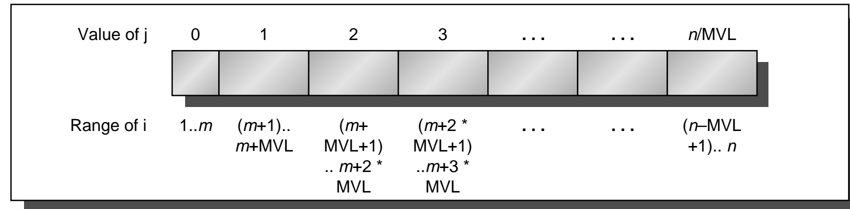

FIGURE B.9 A vector of arbitrary length processed with strip mining. All blocks but the first are of length MVL, utilizing the full power of the vector processor. In this figure, the variable *m* is used for the expression (n mod MVL).

The inner loop of the code above is vectorizable with length VL, which is equal to either (n mod MVL) or MVL. The VLR register must be set twice—once at each place where the variable VL in the code is assigned. With multiple vector operations executing in parallel, the hardware must copy the value of VLR when a vector operation issues, in case VLR is changed for a subsequent vector operation.

In addition to the start-up overhead, we need to account for the overhead of executing the strip-mined loop. This strip-mining overhead, which arises from the need to reinitiate the vector sequence and set the VLR, effectively adds to the vector start-up time, assuming that a convoy does not overlap with other instructions. If that overhead for a convoy is 10 cycles, then the effective overhead per 64 elements increases by 10 cycles, or 0.15 cycles per element.

There are two key factors that contribute to the running time of a strip-mined loop consisting of a sequence of convoys:

1. The number of convoys in the loop, which determines the number of chimes. We use the notation T_{chime} for the execution time in chimes.
2. The overhead for each strip-mined sequence of convoys. This overhead consists of the cost of executing the scalar code for strip mining each block, T_{loop}, plus the vector start-up cost for each convoy, T_{start}.

There may also be a fixed overhead associated with setting up the vector sequence the first time. In recent vector processors this overhead has become quite small, so we ignore it.

The components can be used to state the total running time for a vector sequence operating on a vector of length n, which we will call T_n:

$$T_n = \left\lceil \frac{n}{\text{MVL}} \right\rceil \times (T_{\text{loop}} + T_{\text{start}}) + n \times T_{\text{chime}}$$

The values of T_{start}, T_{loop}, and T_{chime} are compiler and processor dependent. The register allocation and scheduling of the instructions affect both what goes in a convoy and the start-up overhead of each convoy.

For simplicity, we will use a constant value for T_{loop} on DLXV. Based on a variety of measurements of CRAY-1 vector execution, the value chosen is 15 for T_{loop}. At first glance, you might think that this value is too small. The overhead in each loop requires setting up the vector starting addresses and the strides, incrementing counters, and executing a loop branch. In practice, these scalar instructions can be totally or partially overlapped with the vector instructions, minimizing the time spent on these overhead functions. The value of T_{loop} of course depends on the loop structure, but the dependence is slight compared with the connection between the vector code and the values of T_{chime} and T_{start}.

EXAMPLE What is the execution time on DLXV for the vector operation A = B $\times$ s, where s is a scalar and the length of the vectors A and B is 200?

ANSWER Assume the addresses of A and B are initially in `Ra` and `Rb`, s is in `Fs`, and recall that for DLX (and DLXV) R0 always holds 0. Since (200 mod 64) = 8, the first iteration of the strip-mined loop will execute for a vector length

of eight elements, and the following iterations will execute for a vector length of 64 elements. The starting byte addresses of the next segment of each vector is eight times the vector length. Since the vector length is either eight or 64, we increment the address registers by $8 \times 8 = 64$ after the first segment and $8 \times 64 = 512$ for latter segments. The total number of bytes in the vector is $8 \times 200 = 1600$, and we test for completion by comparing the address of the next vector segment to the initial address plus 1600. Here is the actual code:

```
        ADDI     R2,R0,#1600   ;total # bytes in vector
        ADD      R2,R2,Ra      ;address of the end of A vector
        ADDI     R1,R0,#8      ;loads length of 1st segment
        MOVI2S   VLR,R1        ;load vector length in VLR
        ADDI     R1,R0,#64     ;length in bytes of 1st segment
        ADDI     R3,R0,#64     ;vector length other segments
  Loop: LV       V1,Rb         ;load B
        MULTSV   V2,Fs,V1      ;vector * scalar
        SV       Ra,V2         ;store A
        ADD      Ra,Ra,R1      ;address of next segment of A
        ADD      Rb,Rb,R1      ;address of next segment of B
        ADDI     R1,R0,#512    ;load byte offset next segment
        MOVI2S   VLR,R3        ;set length to 64 element
        SUB      R4,R2,Ra      ;at the end of A?
        BNEZ     R4,Loop       ;if not, go back
```

The three vector instructions in the loop are dependent and must go into three convoys, hence $T_{chime} = 3$. Let's use our basic formula:

$$T_n = \left\lceil \frac{n}{MVL} \right\rceil \times (T_{loop} + T_{start}) + n \times T_{chime}$$

$$T_{200} = 4 \times (15 + T_{start}) + 200 \times 3$$

$$T_{200} = 60 + (4 \times T_{start}) + 600 = 660 + (4 \times T_{start})$$

The value of T_{start} is the sum of

- The vector load start-up of 12 clock cycles
- A seven-clock-cycle start-up for the multiply
- A 12-clock-cycle start-up for the store.

Thus, the value of T_{start} is given by

$$T_{start} = 12 + 7 + 12 = 31$$

So, the overall value becomes

$$T_{200} = 660 + 4 \times 31 = 784$$

The execution time per element with all start-up costs is then 784/200 = 3.9, compared with a chime approximation of three. In section B.6, we will be more ambitious—allowing overlapping of separate convoys. ■

Figure B.10 shows the overhead and effective rates per element for the above example (A = B × s) with various vector lengths. A chime counting model would lead to three clock cycles per element, while the two sources of overhead add 0.9 clock cycles per element in the limit.

The next few sections introduce enhancements that reduce this time. We will see how to reduce the number of convoys and hence the number of chimes using a technique called *chaining*. The loop overhead can be reduced by further overlapping the execution of vector and scalar instructions, allowing the scalar loop overhead in one iteration to be executed while the vector instructions in the previous instruction are completing. Finally, the vector start-up overhead can also be eliminated, using a technique that allows overlap of vector instructions in separate convoys.

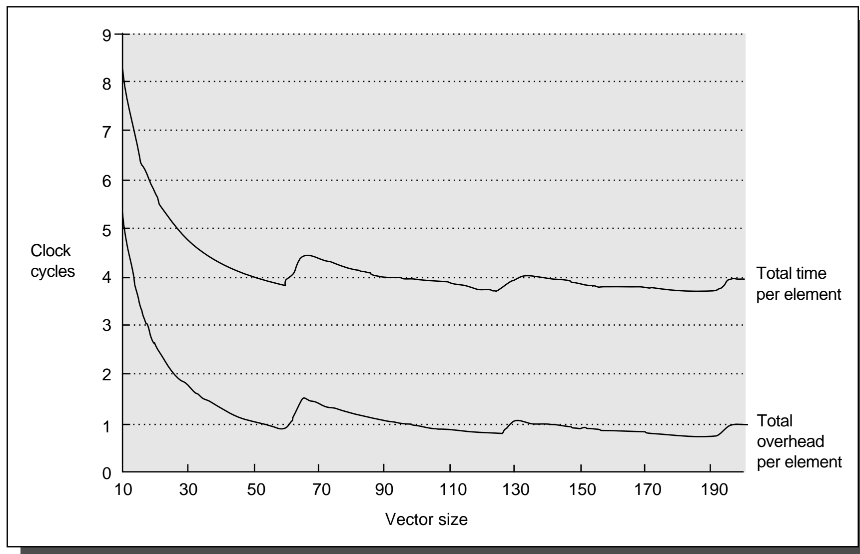

FIGURE B.10 This shows the total execution time per element and the total overhead time per element, versus the vector length for the Example on page B-17. For short vectors the total start-up time is more than one-half of the total time, while for long vectors it reduces to about one-third of the total time. The sudden jumps occur when the vector length crosses a multiple of 64, forcing another iteration of the strip-mining code and execution of a set of vector instructions. These operations increase T_n by $T_{loop} + T_{start}$.

Vector Stride

The second problem this section addresses is that the position in memory of adjacent elements in a vector may not be sequential. Consider the straightforward code for matrix multiply:

```
          do 10 i = 1,100
                 do 10 j = 1,100
                        A(i,j) = 0.0
                        do 10 k = 1,100
10                      A(i,j) = A(i,j)+B(i,k)*C(k,j)
```

At the statement labeled 10 we could vectorize the multiplication of each row of `B` with each column of `C` and strip-mine the inner loop with `k` as the index variable.

To do so, we must consider how adjacent elements in `B` and adjacent elements in `C` are addressed. As we discussed in section 5.3, when an array is allocated memory it is linearized and must be laid out in either row-major or column-major order. This linearization means that either the elements in the row or the elements in the column are not adjacent in memory. For example, if the above loop were written in FORTRAN, which allocates column-major order, the elements of `B` that are accessed by iterations in the inner loop are separated by the row size times 8 (the number of bytes per entry) for a total of 800 bytes. In Chapter 5, we saw that blocking could be used to improve the locality in cache-based systems. In vector processors we do not have caches, so we need another technique to fetch elements of a vector that are not adjacent in memory.

This distance separating elements that are to be gathered into a single register is called the *stride*. In the current example, using column-major layout for the matrices means that matrix `C` has a stride of 1, or 1 double word (8 bytes), separating successive elements, and matrix `B` has a stride of 100, or 100 double words (800 bytes).

Once a vector is loaded into a vector register it acts as if it had logically adjacent elements. Thus a vector-register processor can handle strides greater than one, called *nonunit strides*, using only vector-load and vector-store operations with stride capability. This ability to access nonsequential memory locations and to reshape them into a dense structure is one of the major advantages of a vector processor over a cache-based processor. Caches inherently deal with unit stride data, so that while increasing block size can help reduce miss rates for large scientific data sets, increasing block size can have a negative effect for data that is accessed with nonunit stride. While blocking techniques can solve some of these problems (see section 5.3), the ability to efficiently access data that is not contiguous remains an advantage for vector processors on certain problems.

On DLXV, where the addressable unit is a byte, the stride for our example would be 800. The value must be computed dynamically, since the size of the matrix may not be known at compile time, or—just like vector length—may

change for different executions of the same statement. The vector stride, like the vector starting address, can be put in a general-purpose register. Then the DLXV instruction LVWS (load vector with stride) can be used to fetch the vector into a vector register. Likewise, when a nonunit stride vector is being stored, SVWS (store vector with stride) can be used. In some vector processors the loads and stores always have a stride value stored in a register, so that only a single load and a single store instruction are required.

Complications in the memory system can occur from supporting strides greater than one. In Chapter 5 we saw that memory accesses could proceed at full speed if the number of memory banks was at least as large as the memory-access time in clock cycles. Once nonunit strides are introduced, however, it becomes possible to request accesses from the same bank at a higher rate than the memory-access time. When multiple accesses contend for a bank, a memory bank conflict occurs and one access must be stalled. A bank conflict, and hence a stall, will occur if

$$\frac{\text{Least common multiple (Stride, Number of banks)}}{\text{Stride}} < \text{Memory-access latency}$$

EXAMPLE Suppose we have 16 memory banks with a read latency of 12 clocks. How long will it take to complete a 64-element vector load with a stride of 1? With a stride of 32?

ANSWER Since the number of banks is larger than the read latency, for a stride of 1, the load will take 12 + 64 = 76 clock cycles, or 1.2 clocks per element. The worst possible stride is a value that is a multiple of the number of memory banks, as in this case with a stride of 32 and 16 memory banks. Every access to memory will collide with the previous one. This leads to a read latency of 12 clock cycles per element and a total time for the vector load of 768 clock cycles. ■

Memory bank conflicts will not occur if the stride and number of banks are relatively prime with respect to each other and there are enough banks to avoid conflicts in the unit-stride case. When there are no bank conflicts, multiword and unit strides run at the same rates. Increasing the number of memory banks to a number greater than the minimum to prevent stalls with a stride of length 1 will decrease the stall frequency for some other strides. For example, with 64 banks, a stride of 32 will stall on every other access, rather than every access. If we originally had a stride of 8 and 16 banks, every other access would stall; while with 64 banks, a stride of 8 will stall on every eighth access. If we have multiple memory pipelines, we will also need more banks to prevent conflicts. In 1995, most vector supercomputers have at least 64 banks, and some have as many as 1024 in the maximum memory configuration. Because bank conflicts can still occur in nonunit stride cases, many programmers favor unit stride accesses whenever possible.

B.4 Effectiveness of Compiler Vectorization

Two factors affect the success with which a program can be run in vector mode. The first factor is the structure of the program itself: Do the loops have true data dependences, or can they be restructured so as not to have such dependences? This factor is influenced by the algorithms chosen and, to some extent, by how they are coded. The second factor is the capability of the compiler. While no compiler can vectorize a loop where no parallelism among the loop iterations exists, there is tremendous variation in the ability of compilers to determine whether a loop can be vectorized. The techniques used to vectorize programs are the same as those discussed in Chapter 4 for uncovering ILP; here we simply review how well these techniques work.

As an indication of the level of vectorization that can be achieved in scientific programs, let's look at the vectorization levels observed for the Perfect Club benchmarks, mentioned in Chapter 1. These benchmarks are large, real scientific applications. Figure B.11 shows the percentage of floating-point operations in

Benchmark name	FP operations	FP operations executed in vector mode
ADM	23%	68%
DYFESM	26%	95%
FLO52	41%	100%
MDG	28%	27%
MG3D	31%	86%
OCEAN	28%	58%
QCD	14%	1%
SPICE	16%	7%
TRACK	9%	23%
TRFD	22%	10%

FIGURE B.11 Level of vectorization among the Perfect Club benchmarks when executed on the CRAY X-MP. The first column contains the percentage of operations that are floating point, while the second contains the percentage of FP operations executed in vector instructions.

each benchmark and the percentage executed in vector mode on the CRAY X-MP. The wide variation in level of vectorization has been observed by several studies of the performance of applications on vector processors. While better compilers might improve the level of vectorization in some of these programs, most will

require rewriting to achieve significant increases in vectorization. For example, a new program or a significant rewrite will be needed to obtain the benefits of a vector processor on SPICE.

There is also tremendous variation in how well compilers do in vectorizing programs. As a summary of the state of vectorizing compilers, consider the data in Figure B.12, which shows the extent of vectorization for different processors using a test suite of 100 hand-written FORTRAN kernels. The kernels were designed to test vectorization capability and can all be vectorized by hand; we will see several examples of these loops in the Exercises.

Processor	Compiler	Completely vectorized	Partially vectorized	Not vectorized
CDC CYBER-205	VAST-2 V2.21	62	5	33
Convex C-series	FC5.0	69	5	26
CRAY X-MP	CFT77 V3.0	69	3	28
CRAY X-MP	CFT V1.15	50	1	49
CRAY-2	CFT2 V3.1a	27	1	72
ETA-10	FTN 77 V1.0	62	7	31
Hitachi S810/820	FORT77/HAP V20-2B	67	4	29
IBM 3090/VF	VS FORTRAN V2.4	52	4	44
NEC SX/2	FORTRAN77 / SX V.040	66	5	29

FIGURE B.12 Result of applying vectorizing compilers to the 100 FORTRAN test kernels. For each processor we indicate how many loops were completely vectorized, partially vectorized, and unvectorized. These loops were collected by Callahan, Dongarra, and Levine [1988]. Two different compilers for the CRAY X-MP show the large dependence on compiler technology.

B.5 Enhancing Vector Performance

Three techniques for improving the performance of vector processors are discussed in this section. The first deals with making a sequence of dependent vector operations run faster. The other two deal with expanding the class of loops that can be run in vector mode. The first technique, *chaining,* originated in the CRAY-1, but is now supported on most vector processors. The techniques discussed in the second and third parts of this section combat the effects of conditional execution and sparse matrices. The extensions are taken from a variety of processors including the most recent supercomputers.

Chaining—The Concept of Forwarding Extended to Vector Registers

Consider the simple vector sequence

```
MULTV  V1,V2,V3
ADDV   V4,V1,V5
```

In DLXV, as it currently stands, these two instructions must be put into two separate convoys, since the instructions are dependent. On the other hand, if the vector register, `V1` in this case, is treated not as a single entity but as a group of individual registers, then the ideas of forwarding can be conceptually extended to work on individual elements of a vector. This insight, which will allow the `ADDV` to start earlier in this example, is called *chaining*. Chaining allows a vector operation to start as soon as the individual elements of its vector source operand become available: The results from the first functional unit in the chain are "forwarded" to the second functional unit. In practice, chaining is often implemented by allowing the processor to read and write a particular register at the same time, albeit to different elements. Early implementations of chaining worked like forwarding, but this restricted the timing of the source and destination instructions in the chain. Recent implementations use *flexible chaining*, which allows a vector instruction to chain to essentially any other active vector instruction, assuming that no structural hazard is generated. Flexible chaining requires more read and write ports for the vector register file, but it is the form of chaining used in most recent machines. We assume this type of chaining throughout the rest of this appendix.

Even though a pair of operations depend on one another, chaining allows the operations to proceed in parallel on separate elements of the vector. This permits the operations to be scheduled in the same convoy and reduces the number of chimes required. For the sequence above, a sustained rate (ignoring start-up) of two floating-point operations per clock cycle, or one chime, can be achieved, even though the operations are dependent! The total running time for the above sequence becomes

$$\text{Vector length} + \text{Start-up time}_{\text{ADDV}} + \text{Start-up time}_{\text{MULTV}}$$

Figure B.13 shows the timing of a chained and an unchained version of the above pair of vector instructions with a vector length of 64. This convoy requires one chime; however, because it uses chaining, the start-up overhead will be seen in the actual timing of the convoy. In Figure B.13, the total time for chained operation is 77 clock cycles, or 1.2 cycles per result. With 128 floating-point operations done in that time, 1.7 FLOPs per clock cycle are obtained. For the unchained version, there are 141 clock cycles or 0.9 FLOPs per clock cycle.

Although chaining allows us to reduce the chime component of the execution time by putting two dependent instructions in the same convoy, it does not

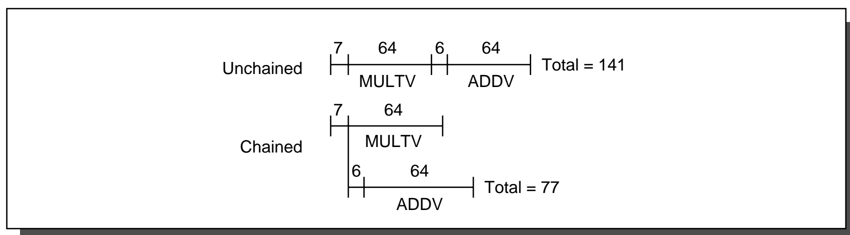

FIGURE B.13 Timings for a sequence of dependent vector operations ADDV and MULTV, both unchained and chained. The 6- and 7-clock-cycle delays are the latency of the adder and multiplier.

eliminate the start-up overhead. If we want an accurate running time estimate, we must count the start-up time both within and across convoys. With chaining the number of chimes for a sequence is determined by the number of different vector functional units available in the processor and the number required by the application. In particular, no convoy can contain a structural hazard. This means, for example, that a sequence containing two vector memory instructions must take at least two convoys, and hence two chimes, on a processor like DLXV with only one vector load-store unit.

We will see in section B.6 that chaining plays a major role in boosting vector performance. In fact, chaining is so important that virtually every vector processor now supports flexible chaining.

Conditionally Executed Statements

In the last section, we saw that many programs only achieved low to moderate levels of vectorization. Because of Amdahl's Law, the speedup on such programs will be very limited. Two reasons why higher levels of vectorization are not achieved are the presence of conditionals (if statements) inside loops and the use of sparse matrices. Programs that contain if statements in loops cannot be run in vector mode using the techniques we have discussed so far because the if statements introduce control dependences into a loop. Likewise, sparse matrices cannot be efficiently implemented using any of the capabilities we have seen so far; this is one factor in the lack of vectorization for SPICE. We discuss strategies for dealing with conditional execution here, leaving the discussion of sparse matrices to the following subsection.

Consider the following loop:

```
      do 100 i = 1, 64
            if (A(i).ne. 0) then
                  A(i) = A(i) – B(i)
            endif
100   continue
```

This loop cannot normally be vectorized because of the conditional execution of the body; however, if the inner loop could be run for the iterations for which A(i) ≠ 0, then the subtraction could be vectorized. In Chapter 4, we saw that the conditionally executed instructions could turn such control dependences into data dependences, enhancing the ability to parallelize the loop. Vector processors can benefit from an equivalent capability for vectors.

The extension that is commonly used for this capability is *vector-mask control*. The vector-mask control uses a Boolean vector of length MVL to control the execution of a vector instruction just as conditionally executed instructions use a Boolean condition to determine whether an instruction is executed. When the *vector-mask register* is enabled, any vector instructions executed operate only on the vector elements whose corresponding entries in the vector-mask register are 1. The entries in the destination vector register that correspond to a 0 in the mask register are unaffected by the vector operation. If the vector-mask register is set by the result of a condition, only elements satisfying the condition will be affected. Clearing the vector-mask register sets it to all 1s, making subsequent vector instructions operate on all vector elements. The following code can now be used for the above loop, assuming that the starting addresses of A and B are in `Ra` and `Rb`, respectively:

```
LV      V1,Ra         ;load vector A into V1
LV      V2,Rb         ;load vector B
LD      F0,#0         ;load FP zero into F0
SNESV   F0,V1         ;sets VM(i) to 1 if V1(i)≠F0
SUBV    V1,V1,V2      ;subtract under vector mask
CVM                   ;set the vector mask to all 1s
SV      Ra,V1         ;store the result in A
```

Most recent vector processors provide vector-mask control. The vector-mask capability described here is available on some processors, but others allow the use of the vector mask with only a subset of the vector instructions.

Using a vector-mask register does, however, have disadvantages. When we examined conditionally executed instructions, we saw that such instructions still require execution time when the condition is not satisfied. Nonetheless, the elimination of a branch and the associated control dependences can make a conditional instruction faster even if it sometimes does useless work. Similarly, vector instructions executed with a vector mask still take execution time, even for the elements where the mask is 0. Likewise, even with a significant number of zeros in the mask, using vector-mask control may still be significantly faster than using scalar mode. In fact, the large difference in potential performance between vector and scalar mode makes the inclusion of vector-mask instructions critical.

Second, in some vector processors the vector mask serves only to disable the storing of the result into the destination register, and the actual operation still occurs. Thus, if the operation in the above example were a divide rather than a

subtract and the test was on B rather than A, false floating-point exceptions might result since a division by 0 would occur. Processors that mask the operation as well as the storing of the result avoid this problem.

Sparse Matrices

There are techniques for allowing programs with sparse matrices to execute in vector mode. In a sparse matrix, the elements of a vector are usually stored in some compacted form and then accessed indirectly. Assuming a simplified sparse structure, we might see code that looks like this:

```
          do 100 i = 1,n
100          A(K(i)) = A(K(i)) + C(M(i))
```

This code implements a sparse vector sum on the arrays `A` and `C`, using index vectors `K` and `M` to designate the nonzero elements of `A` and `C`. (`A` and `C` must have the same number of nonzero elements—`n` of them.) Another common representation for sparse matrices uses a bit vector to say which elements exist and a dense vector for the nonzero elements. Often both representations exist in the same program. Sparse matrices are found in many codes, and there are many ways to implement them, depending on the data structure used in the program.

A primary mechanism for supporting sparse matrices is *scatter-gather operations* using index vectors. The goal of such operations is to support moving between a dense representation (i.e., zeros are not included) and normal representation (i.e., the zeros are included) of a sparse matrix. A *gather* operation takes an *index vector* and fetches the vector whose elements are at the addresses given by adding a base address to the offsets given in the index vector. The result is a nonsparse vector in a vector register. After these elements are operated on in dense form, the sparse vector can be stored in expanded form by a *scatter* store, using the same index vector. Hardware support for such operations is called *scatter-gather* and appears on several processors. The instructions `LVI` (load vector indexed) and `SVI` (store vector indexed) provide these operations in DLXV. For example, assuming that `Ra`, `Rc`, `Rk`, and `Rm` contain the starting addresses of the vectors in the above sequence, the inner loop of the sequence can be coded with vector instructions such as

```
LV      Vk,Rk           ;load K
LVI     Va,(Ra+Vk)      ;load A(K(I))
LV      Vm,Rm           ;load M
LVI     Vc,(Rc+Vm)      ;load C(M(I))
ADDV    Va,Va,Vc        ;add them
SVI     (Ra+Vk),Va      ;store A(K(I))
```

This technique allows code with sparse matrices to be run in vector mode. The source code above would *never* be automatically vectorized by a compiler because the compiler cannot know that the elements of `K` are distinct values, and thus that no dependences exist. Instead, a programmer directive would tell the compiler that it could run the loop in vector mode; without such directives, programs such as SPICE will not be vectorized even if the hardware support exists.

A scatter-gather capability is included on many of the recent supercomputers. Such operations rarely run at one element per clock, but they are still much faster than the alternative, which may be a scalar loop. If the sparsity properties of a matrix change, a new index vector must be computed. Many processors provide support for computing the index vector quickly. The `CVI` (create vector index) instruction in DLXV creates an index vector given a stride (m), where the values in the index vector are 0, m, $2 \times m$, ..., $63 \times m$. Some processors provide an instruction to create a compressed index vector whose entries correspond to the positions with a 1 in the mask register. Other vector architectures provide a method to compress a vector. In DLXV, we define the `CVI` instruction to always create a compressed index vector using the vector mask. When the vector mask is all ones, a standard index vector will be created.

The indexed loads-stores and the `CVI` instruction provide an alternative method to support conditional vector execution. Here is a vector sequence that implements the loop we saw on page B-25:

```
LV        V1,Ra         ;load vector A into V1
LD        F0,#0         ;load FP zero into F0
SNESV     F0,V1         ;sets the VM to 1 if V1(i)≠F0
CVI       V2,#8         ;generates indices in V2
POP       R1,VM         ;find the number of 1's in VM
MOVI2S    VLR,R1        ;load vector length register
CVM                     ;clears the mask
LVI       V3,(Ra+V2)    ;load the nonzero A elements
LVI       V4,(Rb+V2)    ;load corresponding B elements
SUBV      V3,V3,V4      ;do the subtract
SVI       (Ra+V2),V3    ;store A back
```

Whether the implementation using scatter-gather is better than the conditionally executed version depends on the frequency with which the condition holds and the cost of the operations. Ignoring chaining, the running time of the first version (on page B-25) is $5n + c_1$. The running time of the second version, using indexed loads and stores with a running time of one element per clock, is $4n + 4 \times f \times n + c_2$, where f is the fraction of elements for which the condition is true (i.e., A $\neq 0$). If we assume that the values of c_1 and c_2 are comparable, or that they are much smaller than n, we can find when this second technique is better.

$$\text{Time}_1 = 5(n)$$
$$\text{Time}_2 = 4n + 4 \times f \times n$$

We want $\text{Time}_1 \geq \text{Time}_2$, so

$$5n \geq 4n + 4 \times f \times n$$
$$\frac{1}{4} \geq f$$

That is, the second method is faster if less than one-quarter of the elements are nonzero. In many cases the frequency of execution is much lower. If the index vector can be reused, or if the number of vector statements within the if statement grows, the advantage of the scatter-gather approach will increase sharply.

B.6 Putting It All Together: Performance of Vector Processors

In this section we look at different measures of performance for vector processors and what they tell us about the processor. To determine the performance of a processor on a vector problem we must look at the start-up cost and the sustained rate. The simplest and best way to report the performance of a vector processor on a loop is to give the execution time of the vector loop. For vector loops people often give the MFLOPS (millions of floating-point operations per second) rating rather than execution time. We use the notation R_n for the MFLOPS rating on a vector of length n. Using the measurements T_n (time) or R_n (rate) is equivalent if the number of FLOPs is agreed upon (see Chapter 1 for a longer discussion on MFLOPS). In any event, either measurement should include the overhead.

In this section we examine the performance of DLXV on our DAXPY loop by looking at performance from different viewpoints. We will continue to compute the execution time of a vector loop using the equation developed in section B.3. At the same time, we will look at different ways to measure performance using the computed time. The constant values for T_{loop} used in this section introduce some small amount of error, which will be ignored.

Measures of Vector Performance

Because vector length is so important in establishing the performance of a processor, length-related measures are often applied in addition to time and MFLOPS. These length-related measures tend to vary dramatically across different processors and are interesting to compare. (Remember, though, that *time* is always the measure of interest when comparing the relative speed of two processors.) Three of the most important length-related measures are

- R_∞—The MFLOPS rate on an infinite-length vector. Although this measure may be of interest when estimating peak performance, real problems do not have unlimited vector lengths, and the overhead penalties encountered in real problems will be larger.
- $N_{1/2}$—The vector length needed to reach one-half of R_∞. This is a good measure of the impact of overhead.
- N_v—The vector length needed to make vector mode faster than scalar mode. This measures both overhead and the speed of scalars relative to vectors.

Let's look at these measures for our DAXPY problem running on DLXV. When chained, the inner loop of the DAXPY code in convoys looks like Figure B.14 (assuming that `Rx` and `Ry` hold starting addresses).

`LV V1,Rx`	`MULTSV V2,F0,V1`	`Convoy 1: chained load and multiply`
`LV V3,Ry`	`ADDV V4,V2,V3`	`Convoy 2: second load and ADD, chained`
`SV Ry,V4`		`Convoy 3: store the result`

FIGURE B.14 The chained inner loop of the DAXPY code in convoys.

Recall our performance equation for the execution time of a vector loop with n elements, T_n:

$$T_n = T_{base} + \left\lceil \frac{n}{MVL} \right\rceil \times (T_{loop} + T_{start}) + n \times T_{chime}$$

Chaining allows the loop to run in three chimes (and no less, since there is one memory pipeline); thus $T_{chime} = 3$. If T_{chime} were a complete indication of performance, the loop would run at a MFLOPS rate of 2/3 × clock rate (since there are 2 FLOPs per iteration). Thus, based only on the chime count, a 200-MHz DLXV would run this loop at 133 MFLOPS assuming no strip-mining or start-up overhead. There are several ways to improve the performance: add additional vector load-store units, allow convoys to overlap to reduce the impact of start-up overheads, and decrease the number of loads required by vector register allocation. We will examine the first two extensions in this section. The last optimization is actually used for the Cray-1, DLXV's cousin, to boost the performance by 50%. Reducing the number of loads requires an interprocedural optimization; we examine this transformation in Exercise B.6. Before we examine the first two extensions, let's see what the real performance, including overhead, is.

The Peak Performance of DLXV on DAXPY

First, we should determine what the peak performance, R_∞, really is, since we know it must differ from the ideal 133-MFLOPS rate. For now, we continue to use the simplifying assumption that a convoy cannot start until all the instructions in an earlier convoy have completed; later we will remove this restriction. Using

this simplification, the start-up overhead for the vector sequence is simply the sum of the start-up times of the instructions:

$$T_{start} = 12 + 7 + 12 + 6 + 12 = 49$$

Using MVL = 64, $T_{loop} = 15$, $T_{start} = 49$, and $T_{chime} = 3$ in the performance equation, and assuming that n is not an exact multiple of 64, the time for an n-element operation is

$$\begin{aligned} T_n &= \left\lceil \frac{n}{64} \right\rceil \times (15 + 49) + 3n \\ &= (n + 64) + 3n \\ &= 4n + 64 \end{aligned}$$

The sustained rate is actually over 4 clock cycles per iteration, rather than the theoretical rate of 3 chimes, which ignores overhead. The major part of the difference is the cost of the start-up overhead for each block of 64 elements (49 cycles versus 15 for the loop overhead).

We can now compute R_∞ for a 200-MHz clock as

$$R_\infty = \lim_{n \to \infty} \left(\frac{\text{Operations per iteration} \times \text{Clock rate}}{\text{Clock cycles per iteration}} \right)$$

The numerator is independent of n, hence

$$R_\infty = \frac{\text{Operations per iteration} \times \text{Clock rate}}{\lim_{n \to \infty} (\text{Clock cycles per iteration})}$$

$$\lim_{n \to \infty} (\text{Clock cycles per iteration}) = \lim_{n \to \infty} \left(\frac{T_n}{n} \right) = \lim_{n \to \infty} \left(\frac{4n + 64}{n} \right) = 4$$

$$R_\infty = \frac{2 \times 200 \text{ MHz}}{4} = 100 \text{ MFLOPS}$$

The performance without the start-up overhead, which is the peak performance given the vector functional unit structure, is now 1.33 times higher. In actuality the gap between peak and sustained performance for this benchmark is even larger!

Sustained Performance of DLXV on the Linpack Benchmark

The Linpack benchmark is a Gaussian elimination on a 100 × 100 matrix. Thus, the vector element lengths range from 99 down to 1. A vector of length k is used k times. Thus, the average vector length is given by

$$\frac{\sum_{i=1}^{99} i^2}{\sum_{i=1}^{99} i} = 66.3$$

Now we can obtain an accurate estimate of the performance of DAXPY using a vector length of 66.

$$T_{66} = 2 \times (15 + 49) + 66 \times 3 = 128 + 198 = 326$$

$$R_{66} = \frac{2 \times 66 \times 200}{326} \text{ MFLOPS} = 81 \text{ MFLOPS}$$

The peak number, ignoring start-up overhead, is 1.64 times higher than this estimate of sustained performance on the real vector lengths. In actual practice, the Linpack benchmark contains a nontrivial fraction of code that cannot be vectorized. Although this code accounts for less than 20% of the time before vectorization, it runs at less than one-tenth of the performance when counted as FLOPs. Thus, Amdahl's Law tells us that the overall performance will be significantly lower than the performance estimated from analyzing the inner loop.

Since vector length has a significant impact on performance, the $N_{1/2}$ and N_v measures are often used in comparing vector machines.

EXAMPLE What is $N_{1/2}$ for just the inner loop of DAXPY for DLXV with a 200-MHz clock?

ANSWER Using R_∞ as the peak rate, we want to know the vector length that will achieve about 50 MFLOPS. We start with the formula for MFLOPS assuming that the measurement is made for $N_{1/2}$ elements:

$$\text{MFLOPS} = \frac{\text{FLOPs executed in } N_{1/2} \text{ iterations}}{\text{Clock cycles to execute } N_{1/2} \text{ iterations}} \times \frac{\text{Clock cycles}}{\text{Second}} \times 10^{-6}$$

$$50 = \frac{2 \times N_{1/2}}{T_{N_{1/2}}} \times 200$$

Simplifying this and then assuming $N_{1/2} \le 64$, so that $T_{n \le 64} = 1 \times 64 + 3 \times n$, yields

$$T_{N_{1/2}} = 8 \times N_{1/2}$$

$$1 \times 64 + 3 \times N_{1/2} = 8 \times N_{1/2}$$

$$5 \times N_{1/2} = 64$$

$$N_{1/2} = 12.8$$

So $N_{1/2} = 13$; that is, a vector of length 13 gives approximately one-half the peak performance for the DAXPY loop on DLXV. ■

EXAMPLE What is the vector length, N_v, such that the vector operation runs faster than the scalar?

ANSWER Again, we know that $N_v < 64$. The time to do one iteration in scalar mode can be estimated as $10 + 12 + 12 + 7 + 6 + 12 = 59$ clocks, where 10 is the estimate of the loop overhead, known to be somewhat less than the strip-mining loop overhead. In the last problem, we showed that this vector loop runs in vector mode in time $T_{n \le 64} = 64 + 3 \times n$ clock cycles. Therefore,

$$64 + 3N_v = 59N_v$$
$$N_v = \left\lceil \frac{64}{56} \right\rceil$$
$$N_v = 2$$

For the DAXPY loop, vector mode is faster than scalar as long as the vector has at least two elements. This number is surprisingly small, as we will see in the next section (*Fallacies and Pitfalls*). ■

DAXPY Performance on an Enhanced DLXV

DAXPY, like many vector problems, is memory limited. Consequently, performance could be improved by adding more memory-access pipelines. This is the major architectural difference between the CRAY X-MP (and later processors) and the CRAY-1. The CRAY X-MP has three memory pipelines, compared with the CRAY-1's single memory pipeline, and the X-MP has more flexible chaining. How does this affect performance?

EXAMPLE What would be the value of T_{66} for DAXPY on DLXV if we added two more memory pipelines?

ANSWER With three memory pipelines all the instructions fit in one convoy and take one chime. The start-up overheads are the same, so

$$T_{66} = \left\lceil \frac{66}{64} \right\rceil \times (T_{loop} + T_{start}) + 66 \times T_{chime}$$
$$T_{66} = 2 \times (15 + 49) + 66 \times 1 = 194$$

With three memory pipelines, we have reduced the clock-cycle count for sustained performance from 326 to 194, a factor of 1.7. Note the effect of Amdahl's Law: We improved the theoretical peak rate, as measured by the number of chimes by a factor of 3, but only achieved an overall improvement of a factor of 1.7 in sustained performance. ■

Another improvement could come from allowing different convoys to overlap and also allowing the scalar loop overhead to overlap with the vector instructions. This requires that one vector operation be allowed to begin using a functional unit before another operation has completed and complicates the instruction issue logic. Allowing this overlap eliminates the separate start-up overhead for every convoy except the first and hides the loop overhead as well.

To achieve the maximum hiding of strip-mining overhead, we need to be able to overlap strip-mined instances of the loop, allowing two instances of a convoy as well as possibly two instances of the scalar code to be in execution simultaneously. This requires the same techniques we looked at in Chapter 4 to avoid WAR hazards, although because no overlapped read and write of a single vector element is possible, copying can be avoided. This technique, called *tailgating*, was used in the Cray-2. Alternatively, we could unroll the outer loop to create several instances of the vector sequence using different register sets (assuming sufficient registers), just as we did in Chapter 4. By allowing maximum overlap of the convoys and the scalar loop overhead, the start-up and loop overheads will only be seen *once* per vector sequence, independent of the number of convoys and the instructions in each convoy. In this way a processor with vector registers can have both low start-up overhead for short vectors and high peak performance for very long vectors.

EXAMPLE What would be the values of R_∞ and T_{66} for DAXPY on DLXV if we added two more memory pipelines and allowed the strip-mining and start-up overhead to be fully overlapped?

ANSWER

$$R_\infty = \lim_{n \to \infty}\left(\frac{\text{Operations per iteration} \times \text{Clock rate}}{\text{Clock cycles per iteration}}\right)$$

$$\lim_{n \to \infty}(\text{Clock cycles per iteration}) = \lim_{n \to \infty}\left(\frac{T_n}{n}\right)$$

Since the overhead is only seen once, $T_n = n + 49 + 15 = n + 64$. Thus,

$$\lim_{n \to \infty}\left(\frac{T_n}{n}\right) = \lim_{n \to \infty}\left(\frac{n+64}{n}\right) = 1$$

$$R_\infty = \frac{2 \times 200\text{ MHz}}{1} = 400\text{ MFLOPS}$$

Adding the extra memory pipelines and more flexible issue logic yields an improvement in peak performance of a factor of 4. However, $T_{66} = 130$, so for shorter vectors, the sustained performance improvement is about 326/130 = 2.5 times. ■

In summary, we have examined several measures of vector performance. Theoretical peak performance can be calculated based purely on the value of T_{chime} as

$$\frac{\text{Number of FLOPs per iteration} \times \text{Clock rate}}{T_{chime}}$$

By including the loop overhead, we can calculate values for peak performance for an infinite-length vector (R_∞) and also for sustained performance, R_n for a vector of length n, which is computed as

$$R_n = \frac{\text{Number of FLOPs per iteration} \times n \times \text{Clock rate}}{T_n}$$

Using these measures we also can find $N_{1/2}$ and N_V, which give us another way of looking at the start-up overhead for vectors and the ratio of vector to scalar speed. A wide variety of measures of performance of vector processors are useful in understanding the range of performance that applications may see on a vector processor.

B.7 Fallacies and Pitfalls

Pitfall: Concentrating on peak performance and ignoring start-up overhead.

Early vector processors such as the TI ASC and the CDC STAR-100 had long start-up times. For some vector problems, N_V could be greater than 100! Today, the supercomputers from Japan often have higher sustained rates than the Cray Research processors. But with start-up overheads that are 50–100% higher, the faster sustained rates often provide no real advantage. On the CYBER-205 the start-up overhead for DAXPY is 158 clock cycles, substantially increasing the break-even point. With a single vector unit, which contains 2 memory pipelines,

the CYBER-205 can sustain a rate of 2 clocks per iteration. The time for DAXPY for a vector of length n is therefore roughly $158 + 2n$. If the clock rates of the CRAY-1 and the CYBER-205 were identical, the CRAY-1 would be faster until $n > 64$. Because the CRAY-1 clock is also faster (even though the 205 is newer), the crossover point is over 100. Comparing a four-vector-pipeline CYBER-205 (the maximum-size processor) with the CRAY X-MP that was delivered shortly after the 205, the 205 completes two results per clock cycle—twice as fast as the X-MP. However, vectors must be longer than about 200 for the CYBER-205 to be faster. The problem of start-up overhead has been the major difficulty for the memory-memory vector architectures, hence their lack of popularity.

Pitfall: Increasing vector performance, without comparable increases in scalar performance.

This was a problem on many early vector processors, and a place where Seymour Cray rewrote the rules. Many of the early vector processors had comparatively slow scalar units (as well as large start-up overheads). Even today, processors with higher peak vector performance can be outperformed by a processor with lower vector performance but better scalar performance. Good scalar performance keeps down overhead costs (strip mining, for example) and reduces the impact of Amdahl's Law. A good example of this comes from comparing a fast scalar processor and a vector processor with lower scalar performance. The Livermore FORTRAN kernels are a collection of 24 scientific kernels with varying degrees of vectorization. Figure B.15 shows the performance of two different processors on this benchmark. Despite the vector processor's higher peak performance, its low scalar performance makes it slower than a fast scalar processor. The next fallacy is closely related.

Processor	Minimum rate for any loop	Maximum rate for any loop	Harmonic mean of all 24 loops
MIPS M/120-5	0.80 MFLOPS	3.89 MFLOPS	1.85 MFLOPS
Stardent-1500	0.41 MFLOPS	10.08 MFLOPS	1.72 MFLOPS

FIGURE B.15 Performance measurements for the Livermore FORTRAN kernels on two different processors. Both the MIPS M/120-5 and the Stardent-1500 (formerly the Ardent Titan-1) use a 16.7-MHz MIPS R2000 chip for the main CPU. The Stardent-1500 uses its vector unit for scalar FP and has about half the scalar performance (as measured by the minimum rate) of the MIPS M/120, which uses the MIPS R2010 FP chip. The vector processor is more than a factor of 2.5 times faster for a highly vectorizable loop (maximum rate). However, the lower scalar performance of the Stardent-1500 negates the higher vector performance when total performance is measured by the harmonic mean on all 24 loops.

Fallacy: You can get vector performance without providing memory bandwidth.

As we saw with the DAXPY loop, memory bandwidth is quite important. DAXPY requires 1.5 memory references per floating-point operation, and this ratio is typical of many scientific codes. Even if the floating-point operations took no time, a CRAY-1 could not increase the performance of the vector sequence used, since it is memory limited. The CRAY-1 performance on Linpack jumped when the compiler used clever transformations to change the computation so that values could be kept in the vector registers. This lowered the number of memory references per FLOP and improved the performance by nearly a factor of 2! Thus, the memory bandwidth on the CRAY-1 became sufficient for a loop that formerly required more bandwidth.

B.8 Concluding Remarks

In the late 1980s rapid performance increases in efficiently pipelined scalar processors led to a dramatic closing of the gap between vector supercomputers, costing millions of dollars, and fast, pipelined, VLSI microprocessors costing less than tens of thousands of dollars. In Chapter 1, we saw that a desk-side processor offered nearly the performance of a vector supercomputer introduced five years earlier for less than a tenth of the price. Comparing that processor against its contemporary, a Cray C-90, would show a reduced price-performance advantage, but still exceeding a factor of three times. While the price advantage comes from the use of microprocessor technology, the high performance comes from the exploitation of instruction-level parallelism in the microprocessor, which allows CPIs to be under 1.

For scientific programs, an interesting counterpart to CPI is clock cycles per FLOP, or CPF. We saw in this chapter that for vector processors this number was typically in the range of 2 (for a CRAY X-MP style processor) to 4 (for a CRAY-1 style processor); a C-90 might reduce this number further but probably not below 1 to 1.5. In Chapter 4, we saw that the pipelined processor varied from about 6 (for DLX) down to about 2.5 (for a superscalar DLX with no memory system losses running a DAXPY-type loop). For processors like an IBM Power-2 or MIPS R8000 with multiple memory pipelines and a multiply-add instruction, this number could be as low as 1.

In addition to the use of vectors rather than multiple issue, the other major distinction between vector machines and advanced scalar machines is the use of vector memory systems versus caches. As we saw earlier in this appendix, vector memory systems can have significant advantages when accesses do not have unit stride. This performance advantage, however, comes at a significant price disadvantage. To keep the start-up penalties of vector loads small and to keep the number of required memory banks reasonable, many high-end vector machines use SRAM for the main memory. While SRAM has an access time several times lower than that of DRAM, it costs roughly 10 times as much per bit!

Recent trends in vector processor design have focused on high peak-vector performance and multiprocessing. Meanwhile, high-speed scalar processors concentrate on keeping the ratio of peak to sustained performance near 1. Thus, if the

peak rates advance comparably, the sustained rates of the scalar processors will advance more quickly, and the scalar processors will continue to close the CPF gap. These multiple-issue scalar processors can rival or exceed the performance of vector processors with comparable clock speeds, especially for levels of vectorization below 70%.

In 1994, we saw two dramatic demonstrations that the gap between vector processors and superscalars may disappear in the future. First, microprocessors with clock rates exceeding those of the high-end Cray C-90 appeared. Second, microprocessors such as the MIPS R8000 (TFP) and the IBM Power-2 delivered CPF numbers competitive with vector processors by issuing multiple memory references and FP operations per cycle. In the near future, it is likely that designers will be able to use the advances in silicon technology to achieve low CPF performance while also achieving a high clock rate. At that point it may be primarily the memory systems that distinguish vector processors from microprocessor-based superscalars. Advances in compiler technology for cache-based systems, such as blocking and prefetching, are closing the performance gap in the memory system, while cache-based systems continue to have large cost advantages. New cache organizations, such as that used in the R8000 (a large pipelined cache for all FP data), are also helping to close the performance gap. New advances are likely to further narrow the advantages of vector-oriented memory systems both by reducing the performance gap and by narrowing the range of applications where a vector memory system is better than a cache-based system. Overall, a Cray C-90 processor has a SPECfp rating that is about 1.8 times higher than an R8000 processor and a price almost 20 times higher. On some benchmarks, however, the C-90 is over five times faster; while on others it is about half the speed of the R8000. Whether the range of applications for which the C-90 has a substantial performance advantage will remain large enough to justify the premium price for vector computers remains to be seen.

B.9 Historical Perspective and References

The first vector processors were the CDC STAR-100 (see Hintz and Tate [1972]) and the TI ASC (see Watson [1972]), both announced in 1972. Both were memory-memory vector processors. They had relatively slow scalar units—the STAR used the same units for scalars and vectors—making the scalar pipeline extremely deep. Both processors had high start-up overhead and worked on vectors of several hundred to several thousand elements. The crossover between scalar and vector could be over 50 elements. It appears that not enough attention was paid to the role of Amdahl's Law on these two processors.

Cray, who worked on the 6600 and the 7600 at CDC, founded Cray Research and introduced the CRAY-1 in 1976 (see Russell [1978]). The CRAY-1 used a vector-register architecture to significantly lower start-up overhead. He also had efficient support for nonunit stride and invented chaining. Most importantly, the

CRAY-1 was the fastest scalar processor in the world at that time. This matching of good scalar and vector performance was probably the most significant factor in making the CRAY-1 a success. Some customers bought the processor primarily for its outstanding scalar performance. Many subsequent vector processors are based on the architecture of this first commercially successful vector processor. Baskett and Keller [1977] provide a good evaluation of the CRAY-1.

In 1981, CDC started shipping the CYBER-205 (see Lincoln [1982]). The 205 had the same basic architecture as the STAR, but offered improved performance all around as well as expandability of the vector unit with up to four vector pipelines, each with multiple functional units and a wide load-store pipe that provided multiple words per clock. The peak performance of the CYBER-205 greatly exceeded the performance of the CRAY-1. However, on real programs, the performance difference was much smaller.

The CDC STAR processor and its descendant, the CYBER-205, were memory-memory vector processors. To keep the hardware simple and support the high bandwidth requirements (up to three memory references per FLOP), these processors did not efficiently handle nonunit stride. While most loops have unit stride, a nonunit stride loop had poor performance on these processors because memory-to-memory data movements were required to gather together (and scatter back) the nonadjacent vector elements; these operations used special scatter-gather instructions. In addition, there was special support for sparse vectors that used a bit vector to represent the zeros and nonzeros and a dense vector of nonzero values. These more complex vector operations were slow because of the long memory latency, and it was often faster to use scalar mode for sparse or nonunit stride operations. Schneck [1987] described several of the early pipelined processors (e.g., Stretch) through the first vector processors, including the 205 and CRAY-1. Dongarra [1986] did another good survey, focusing on more recent processors.

In 1983, Cray Research shipped the first CRAY X-MP (see Chen [1983]). With an improved clock rate (9.5 ns versus 12.5 on the CRAY-1), better chaining support, and multiple memory pipelines, this processor maintained the Cray Research lead in supercomputers. The CRAY-2, a completely new design configurable with up to four processors, was introduced later. A major feature of the CRAY-2 was the use of DRAM, which made it possible to have very large memories. The first CRAY-2 with its 256 M word (60-bit words) memory contained more memory than the total of all the Cray machines shipped to that point! The CRAY-2 had a much faster clock than the X-MP, but also much deeper pipelines; however, it lacked chaining, had an enormous memory latency, and had only one memory pipe per processor. In general, the CRAY-2 is only faster than the CRAY X-MP on problems that require its very large main memory.

The 1980s also saw the arrival of smaller-scale vector processors, called mini-supercomputers. Priced at roughly one-tenth the cost of a supercomputer ($0.5 to $1 million versus $5 to $10 million), these processors caught on quickly. Although many companies joined the market, the two companies that were most

successful were Convex and Alliant. Convex started with a uniprocessor vector processor (C-1) and now offers a small multiprocessor (C-2); they emphasize Cray software capability. One of the keys to the success of Convex has been the effectiveness of their compiler (see Figure B.12 on page B-23) and the quality of their Unix OS implementation. The Convex example illustrates the increasing importance of software—even in the supercomputer business. Alliant [1987] concentrated more on the multiprocessor aspects; they built an eight-processor computer, with each processor offering vector capability. Alliant ceased operation in the early 1990s.

In 1983, processor vendors from Japan entered the supercomputer marketplace, starting with the Fujitsu VP100 and VP200 (Miura and Uchida [1983]), and later expanding to include the Hitachi S810 and the NEC SX/2 (see Watanabe [1987]). These processors have proved to be close to the CRAY X-MP in performance. In general, these three processors have much higher peak performance than the CRAY X-MP. However, because of large start-up overhead, their typical performance is often lower than the CRAY X-MP (see Figure 1.18 in Chapter 1). The CRAY X-MP favored a multiple-processor approach, first offering a two-processor version and later a four-processor. In contrast, the three Japanese processors had expandable vector capabilities.

In 1988, Cray Research introduced the CRAY Y-MP—a bigger and faster version of the X-MP. The Y-MP allows up to eight processors and lowers the cycle time to 6 ns. With a full complement of eight processors, the Y-MP was generally the fastest supercomputer, though the single-processor Japanese supercomputers may be faster than a one-processor Y-MP. In late 1989 Cray Research was split into two companies, both aimed at building high-end processors available in the early 1990s. Seymour Cray headed the spin-off, Cray Computer Corporation, until its demise in 1995. Their initial processor, the CRAY-3, was to be implemented in gallium arsenide, but they were unable to develop a reliable and cost-effective implementation technology. The CRAY-3 was cancelled and efforts were aimed at the CRAY-4, scheduled for delivery in 1995–96.

Cray Research focused on the C90, a new high-end processor with up to 16 processors and a clock rate of 240 MHz. This processor was delivered in 1991. Typical configurations are about $15 million. In 1993, Cray Research introduced their first highly parallel processor, the T3D. In 1995, they announced the availability of both a new low-end vector machine, the J90, and a high-end machine, the T90. The T90 is much like the C90, but offers a clock that is twice as fast (500 MHz), using three-dimensional packaging and optical clock distribution. Like the C90, the T90 costs in the tens of millions, though a single CPU is available for $2,500,000. The J90 is a CMOS-based vector machine using DRAM memory starting at $250,000, but with typical configurations running about $1 million. In mid 1995, Silicon Graphics acquired Cray Research, Inc.

In the early 1980s, CDC spun out a group, called ETA, to build a new supercomputer, the ETA-10, capable of 10 gigaFLOPS. The ETA processor delivered in the late 1980s (see Fazio [1987]) and used low-temperature CMOS in a

configuration with up to 10 processors. Each processor retained the memory-memory architecture based on the CYBER-205. Although the ETA-10 achieved enormous peak performance, its scalar speed was not comparable. In 1989 CDC, the first supercomputer vendor, closed ETA and left the supercomputer design business.

In 1986, IBM introduced the System/370 vector architecture (see Moore et al. [1987]) and its first implementation in the 3090 Vector Facility. The architecture extends the System/370 architecture with 171 vector instructions. The 3090/VF is integrated into the 3090 CPU. Unlike most other vector processors, the 3090/VF routes its vectors through the cache.

The basis for modern vectorizing compiler technology and the notion of data dependence was developed by Kuck and his colleagues [1974] at the University of Illinois. Banerjee [1979] developed the test named after him. Padua and Wolfe [1986] gave a good overview of vectorizing compiler technology.

Benchmark studies of various supercomputers, including attempts to understand the performance differences, have been undertaken by Lubeck, Moore, and Mendez [1985], Bucher [1983], and Jordan [1987]. In Chapter 1, we discussed several benchmark suites aimed at scientific usage and often employed for supercomputer benchmarking, including Linpack and the Lawrence Livermore Laboratories FORTRAN kernels. The University of Illinois coordinated the collection of a set of benchmarks for supercomputers, called the Perfect Club. In 1993, the Perfect Club was integrated into SPEC, which will release a set of benchmarks aimed at high-end scientific processing sometime in 1995.

In less than 20 years vector processors have gone from unproven, new architectures to playing a significant role in the goal to provide engineers and scientists with ever-larger amounts of computing power. The enormous price-performance advantages of microprocessor technology may bring this era to an end. Recently, Cray, NEC, Fujitsu, and Convex announced and delivered large-scale multiprocessors based on microprocessors. By using advanced superscalar microprocessors, designers can build processors that exceed the peak performance of the fastest vector processors. The challenge, as we saw in Chapter 8, lies in programming these processors. As progress is made on this front, the role of vector processors in science and engineering may continue to decrease.

References

ALLIANT COMPUTER SYSTEMS CORP. [1987]. *Alliant FX/Series: Product Summary* (June), Acton, Mass.

BANERJEE, U. [1979]. *Speedup of Ordinary Programs*, Ph.D. Thesis, Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign (October).

BASKETT, F. AND T. W. KELLER [1977]. "An Evaluation of the CRAY-1 Processor," in *High Speed Computer and Algorithm Organization*, D. J. Kuck, D. H. Lawrie, and A. H. Sameh, eds., Academic Press, San Diego, 71–84.

BUCHER, I. Y. [1983]. “The computational speed of supercomputers,” *Proc. SIGMETRICS Conf. on Measuring and Modeling of Computer Systems,* ACM (August), 151–165.

CALLAHAN, D., J. DONGARRA, AND D. LEVINE [1988]. “Vectorizing compilers: A test suite and results,” *Supercomputing ‘88,* ACM/IEEE (November), Orlando, Fla., 98–105.

CHEN, S. [1983]. “Large-scale and high-speed multiprocessor system for scientific applications,” *Proc. NATO Advanced Research Work on High Speed Computing* (June); also in K. Hwang, ed., “Superprocessors: Design and applications,” *IEEE* (August), 1984.

DONGARRA, J. J. [1986]. “A survey of high performance processors,” *COMPCON, IEEE* (March), 8–11.

FAZIO, D. [1987]. “It’s really much more fun building a supercomputer than it is simply inventing one,” *COMPCON, IEEE* (February), 102–105.

FLYNN, M. J. [1966]. “Very high-speed computing systems,” *Proc. IEEE* 54:12 (December), 1901–1909.

HINTZ, R. G. AND D. P. TATE [1972]. “Control data STAR-100 processor design,” *COMPCON, IEEE* (September), 1–4.

JORDAN, K. E. [1987]. “Performance comparison of large-scale scientific processors: Scalar mainframes, mainframes with vector facilities, and supercomputers,” *Computer* 20:3 (March), 10–23.

KUCK, D., P. P. BUDNIK, S.-C. CHEN, D. H. LAWRIE, R. A. TOWLE, R. E. STREBENDT, E. W. DAVIS, JR., J. HAN, P. W. KRASKA, AND Y. MURAOKA [1974]. “Measurements of parallelism in ordinary FORTRAN programs,” *Computer* 7:1 (January), 37–46.

LINCOLN, N. R. [1982]. “Technology and design trade offs in the creation of a modern supercomputer,” *IEEE Trans. on Computers* C-31:5 (May), 363–376.

LUBECK, O., J. MOORE, AND R. MENDEZ [1985]. “A benchmark comparison of three supercomputers: Fujitsu VP-200, Hitachi S810/20, and CRAY X-MP/2,” *Computer* 18:1 (January), 10–29.

MIRANKER, G. S., J. RUBENSTEIN, AND J. SANGUINETTI [1988]. “Squeezing a Cray-class supercomputer into a single-user package,” *COMPCON, IEEE* (March), 452–456.

MIURA, K. AND K. UCHIDA [1983]. “FACOM vector processing system: VP100/200,” *Proc. NATO Advanced Research Work on High Speed Computing* (June); also in K. Hwang, ed., “Superprocessors: Design and applications,” *IEEE* (August 1984), 59–73.

MOORE, B., A. PADEGS, R. SMITH, AND W. BUCHOLZ [1987]. “Concepts of the System/370 vector architecture,” *Proc. 14th Symposium on Computer Architecture* (June), ACM/IEEE, Pittsburgh, 282–292.

PADUA, D. AND M. WOLFE [1986]. “Advanced compiler optimizations for supercomputers,” *Comm. ACM* 29:12 (December), 1184–1201.

RUSSELL, R. M. [1978]. “The CRAY-1 processor system,” *Comm. of the ACM* 21:1 (January), 63–72.

SCHNECK, P. B. [1987]. *Superprocessor Architecture,* Kluwer Academic Publishers, Norwell, Mass.

SMITH, B. J. [1981]. “Architecture and applications of the HEP multiprocessor system,” *Real-Time Signal Processing IV* 298 (August), 241–248.

SPORER, M., F. H. MOSS, AND C. J. MATHAIS [1988]. “An introduction to the architecture of the Stellar Graphics supercomputer,” *COMPCON, IEEE* (March), 464.

WATANABE, T. [1987]. “Architecture and performance of the NEC supercomputer SX system,” *Parallel Computing* 5, 247–255.

WATSON, W. J. [1972]. “The TI ASC—A highly modular and flexible super processor architecture,” *Proc. AFIPS Fall Joint Computer Conf.,* 221–228.

EXERCISES

In these Exercises assume DLXV has a clock rate of 200 MHz and that $T_{loop} = 15$. Use the start-up times from the appendix, and assume that the store latency is always included in the running time.

B.1 [10] <B.1,B.2> Write a DLXV vector sequence that achieves the peak MFLOPS performance of the processor (use the functional unit and instruction description in section B.2). Assuming a 200-MHz clock rate, what is the peak MFLOPS?

B.2 [20/15/15] <B.1–B.6> Consider the following vector code run on a 200-MHz version of DLXV for a fixed vector length of 64:

```
LV      V1,Ra
MULTV   V2,V1,V3
ADDV    V4,V1,V3
SV      Rb,V2
SV      Rc,V4
```

Ignore all strip-mining overhead, but assume that the store latency must be included in the time to perform the loop. The entire sequence produces 64 results.

a. [20] <B.1–B.5> Assuming no chaining and a single memory pipeline, how many chimes are required? How many clock cycles per result (including both stores as one result) does this vector sequence require, including start-up overhead?

b. [15] <B.1–B.5> If the vector sequence is chained, how many clock cycles per result does this sequence require, including overhead?

c. [15] <B.1–B.6> Suppose DLXV had three memory pipelines and chaining. If there were no bank conflicts in the accesses for the above loop, how many clock cycles are required per result for this sequence?

B.3 [20/20/15/15/20/20/20] <B.2–B.6> Consider the following FORTRAN code:

```
        do 10 i=1,n
               A(i) = A(i) + B(i)
               B(i) = x * B(i)
10      continue
```

Use the techniques of section B.6 to estimate performance throughout this Exercise, assuming a 200-MHz version of DLXV.

a. [20] <B.2–B.6> Write the best DLXV vector code for the inner portion of the loop. Assume `x` is in F0 and the addresses of `A` and `B` are in `Ra` and `Rb`, respectively.

b. [20] <B.2–B.6> Find the total time for this loop on DLXV (T_{100}). What is the MFLOP rating for the loop (R_{100})?

c. [15] <B.2–B.6> Find R_{∞} for this loop.

d. [15] <B.2–B.6> Find $N_{1/2}$ for this loop.

e. [20] <B.2–B.6> Find N_V for this loop. Assume the scalar code has been pipeline scheduled so that each memory reference takes six cycles and each FP operation takes three cycles. Assume the scalar overhead is also T_{loop}.

f. [20] <B.2–B.6> Assume DLXV has two memory pipelines. Write vector code that takes advantage of the second memory pipeline. Show the layout in convoys.

g. [20] <B.2–B.6> Compute T_{100} and R_{100} for DLXV with two memory pipelines.

B.4 [20/10] <B.3> Suppose we have a version of DLXV with eight memory banks (each a double word wide) and a memory-access time of eight cycles.

a. [20] <B.3> If a load vector of length 64 is executed with a stride of 20 double words, how many cycles will the load take to complete?

b. [10] <B.3> What percentage of the memory bandwidth do you achieve on a 64-element load at stride 20 versus stride 1?

B.5 [12/12] <B.4–B.6> Consider the following loop:

```
           C = 0.0
           do 10 i=1,64
                  A(i) = A(i) + B(i)
                  C = C + A(i)
    10     continue
```

a. [12] <B.4–B.6> Split the loop into two loops: one with no dependence and one with a dependence. Write these loops in FORTRAN—as a source-to-source transformation. This optimization is called *loop fission.*

b. [12] <B.4–B.6> Write the DLXV vector code for the loop without a dependence.

B.6 [20/15/20/20] <B.4–B.6> The compiled Linpack performance of the CRAY-1 (designed in 1976) was almost doubled by a better compiler in 1989. Let's look at a simple example of how this might occur. Consider the DAXPY-like loop (where *k* is a parameter to the procedure containing the loop):

```
           do 10 i=1,64
                  do 10 j=1,64
                  Y(k,j) = a*X(i,j) + Y(k,j)
    10     continue
```

a. [20] <B.4–B.6> Write the *straightforward* code sequence for just the inner loop in DLXV vector instructions.

b. [15] <B.4–B.6> Using the techniques of section B.6, estimate the performance of this code on DLXV by finding T_{64} in clock cycles. You may assume that T_{loop} of overhead is incurred for each iteration of the outer loop. What limits the performance?

c. [20] <B.4–B.6> Rewrite the DLXV code to reduce the performance limitation; show the resulting inner loop in DLXV vector instructions. (*Hint*: Think about what establishes T_{chime}; can you affect it?) Find the total time for the resulting sequence.

d. [20] <B.4–B.6> Estimate the performance of your new version, using the techniques of section B.6 and finding T_{64}.

B.7 [15/15/25] <B.5> Consider the following code.

```
           do 10 i=1,64
                  if (B(i) .ne. 0) then
                         A(i) = A(i) / B(i)
    10     continue
```

Assume that the addresses of A and B are in Ra and Rb, respectively, and that F0 contains 0.

a. [15] <B.5> Write the DLXV code for this loop using the vector-mask capability.

b. [15] <B.5> Write the DLXV code for this loop using scatter-gather.

c. [25] <B.5> Estimate the performance (T_{100} in clock cycles) of these two vector loops, assuming a divide latency of 20 cycles. Assume that all vector instructions run at one result per clock, independent of the setting of the vector-mask register. Assume that 50% of the entries of B are 0. Considering hardware costs, which would you build if the above loop were typical?

B.8 [15/20/15/15] <B.1–B.6> In *Fallacies and Pitfalls* of Chapter 1, we saw that the difference between peak and sustained performance could be large: For one problem, a Hitachi S810 had a peak speed twice as high as that of the CRAY X-MP, while for another more realistic problem, the CRAY X-MP was twice as fast as the Hitachi processor. Let's examine why this might occur using two versions of DLXV and the following code sequences:

```
C               Code sequence 1
        do 10 i=1,10000
                A(i) = x * A(i) + y * A(i)
10      continue


C               Code sequence 2
        do 10 i=1,100
                A(i) = x * A(i)
10      continue
```

Assume there is a version of DLXV (call it DLXVII) that has two copies of every floating-point functional unit with full chaining among them. Assume that both DLXV and DLXVII have two load-store units. Because of the extra functional units and the increased complexity of assigning operations to units, all the overheads (T_{loop} and T_{start}) are doubled.

a. [15] <B.1–B.6> Find the number of clock cycles for code sequence 1 on DLXV.

b. [20] <B.1–B.6> Find the number of clock cycles on code sequence 1 for DLXVII. How does this compare to DLXV?

c. [15] <B.1–B.6> Find the number of clock cycles on code sequence 2 for DLXV.

d. [15] <B.1–B.6> Find the number of clock cycles on code sequence 2 for DLXVII. How does this compare to DLXV?

B.9 [20] <B.4> Here is a tricky piece of code with two-dimensional arrays. Does this loop have dependences? Can these loops be written so they are parallel? If so, how? Rewrite the *source* code so that it is clear that the loop can be vectorized, if possible.

```
        do 290 j = 2,n
                do 290 i = 2,j
                        aa(i,j)= aa(i-1,j)*aa(i-1,j)+bb(i,j)
290     continue
```

B.10 [12/15] <B.4> Consider the following loop:

```
            do 10 i = 2,n
               A(i) = B
10             C(i) = A(i-1)
```

a. [12] <B.4> Show there is a loop-carried dependence in this code fragment.

b. [15] <B.4> Rewrite the code in FORTRAN so that it can be vectorized as two separate vector sequences.

B.11 [15/25] <B.4> As we saw in Chapter 4 and in section B.4, some loop structures are not easily vectorized. One common structure is a *reduction*—a loop that reduces an array to a single value by repeated application of an operation. This is a special case of a recurrence. A common example occurs in dot product:

```
            dot = 0.0
            do 10 i=1,64
10                 dot = dot + A(i) * B(i)
```

This loop has an obvious loop-carried dependence (on `dot`) and cannot be vectorized in a straightforward fashion. The first thing a good vectorizing compiler would do is split the loop to separate out the vectorizable portion and the recurrence and perhaps rewrite the loop as

```
            do 10 i=1,64
10                 dot(i) = A(i) * B(i)
            do 20 i=2,64
20                 dot(1) = dot(1) + dot(i)
```

The variable `dot` has been expanded into a vector; this transformation is called *scalar expansion*. We can try to vectorize the second loop either relying strictly on the compiler (part (a), or with hardware support as well, part (b)). There is an important caveat in the use of vector techniques for reduction. To make reduction work, we are relying on the associativity of the operator being used for the reduction. Because of rounding and finite range, however, floating-point arithmetic is not strictly associative. For this reason, most compilers require the programmer to indicate whether associativity can be used to more efficiently compile reductions.

a. [15] <B.4> One simple scheme for compiling the loop with the recurrence is to add sequences of progressively shorter vectors—two 32-element vectors, then two 16-element vectors, and so on. This technique has been called *recursive doubling*. It is faster than doing all the operations in scalar mode. Show how the FORTRAN code would look for execution of the second loop in the code fragment above using recursive doubling.

b. [25] <B.4> In some vector processors, the vector registers are addressable, and the operands to a vector operation may be two different parts of the same vector register. This allows another solution for the reduction, called *partial sums*. The key idea in partial sums is to reduce the vector to *m* sums where *m* is the total latency through the vector functional unit, including the operand read and write times. Assume that the DLXV vector registers are addressable (e.g., you can initiate a vector operation with the operand V1(16), indicating that the input operand began with element 16). Also, assume that the total latency for adds, including operand read and write, is eight cycles. Write a DLXV code sequence that reduces the contents of V1 to eight partial sums. It can be done with one vector operation.

B.12 [40] <B.2–B.5> Extend the DLX simulator to be a DLXV simulator, including the ability to count clock cycles. Write some short benchmark programs in DLX and DLXV assembly language. Measure the speedup on DLXV, the percentage of vectorization, and usage of the functional units.

B.13 [50] <B.4> Modify the DLX compiler to include a dependence checker. Run some scientific code and loops through it and measure what percentage of the statements could be vectorized.

B.14 [Discussion] Some proponents of vector processors might argue that the vector processors have provided the best path to ever-increasing amounts of processor power by focusing their attention on boosting peak vector performance. Others would argue that the emphasis on peak performance is misplaced because an increasing percentage of the programs are dominated by nonvector performance. (Remember Amdahl's Law?) The proponents would respond that programmers should work to make their programs vectorizable. What do you think about this argument?

B.15 [Discussion] Consider the points raised in *Concluding Remarks* (section B.8). This topic—the relative advantages of pipelined scalar processors versus FP vector processors—is the source of much debate in the 1990s. What advantages do you see for each side? What would you do in this situation?