G

Vector Processors

Revised by Krste Asanovic
Department of Electrical Engineering and Computer Science, MIT

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's always 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)

G.1 Why Vector Processors?

In Chapters 3 and 4 we saw how we could significantly increase the performance of a processor by issuing multiple instructions per clock cycle and by more deeply pipelining the execution units to allow greater exploitation of instruction-level parallelism. (This appendix assumes that you have read Chapters 3 and 4 completely; in addition, the discussion on vector memory systems assumes that you have read Chapter 5.) Unfortunately, we also saw that there are serious difficulties in exploiting ever larger degrees of ILP.

As we increase both the width of instruction issue and the depth of the machine pipelines, we also increase the number of independent instructions required to keep the processor busy with useful work. This means an increase in the number of partially executed instructions that can be in flight at one time. For a dynamically-scheduled machine, hardware structures, such as instruction windows, reorder buffers, and rename register files, must grow to have sufficient capacity to hold all in-flight instructions, and worse, the number of ports on each element of these structures must grow with the issue width. The logic to track dependencies between all in-flight instructions grows quadratically in the number of instructions. Even a statically scheduled VLIW machine, which shifts more of the scheduling burden to the compiler, requires more registers, more ports per register, and more hazard interlock logic (assuming a design where hardware manages interlocks after issue time) to support more in-flight instructions, which similarly cause quadratic increases in circuit size and complexity. This rapid increase in circuit complexity makes it difficult to build machines that can control large numbers of in-flight instructions, and hence limits practical issue widths and pipeline depths.

Vector processors were successfully commercialized long before instruction-level parallel machines and take an alternative approach to controlling multiple functional units with deep pipelines. 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:

- A single vector instruction specifies a great deal of work—it is equivalent to executing an entire loop. Each instruction represents tens or hundreds of operations, and so the instruction fetch and decode bandwidth needed to keep multiple deeply pipelined functional units busy is dramatically reduced.
- By using a vector instruction, the compiler or programmer indicates that the computation of each result in the vector is independent of the computation of other results in the same vector and so hardware does not have to check for data hazards within a vector instruction. The elements in the vector can be

computed using an array of parallel functional units, or a single very deeply pipelined functional unit, or any intermediate configuration of parallel and pipelined functional units.

- Hardware need only check for data hazards between two vector instructions once per vector operand, not once for every element within the vectors. That means the dependency checking logic required between two vector instructions is approximately the same as that required between two scalar instructions, but now many more elemental operations can be in flight for the same complexity of control logic.
- 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 5.8). 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 application domain can use them frequently.

As mentioned above, vector processors pipeline and parallelize the operations on the individual elements of a vector. The operations include 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 instructions to be in progress at the same time, creating further parallelism among the operations on different vectors.

Vector processors are particularly useful for large scientific and engineering applications, including car crash simulations and weather forecasting, for which a typical job might take dozens of hours of supercomputer time running over multigigabyte data sets. Multimedia applications can also benefit from vector processing, as they contain abundant data parallelism and process large data streams. 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.

G.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 Chapters 3 and 4, and commercial vector machines have included both out-of-order scalar units (NEC SX/5) and VLIW scalar units (Fujitsu VPP5000).

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, including the Cray Research processors (Cray-1, Cray-2, X-MP, Y-MP, C90, T90, and SV1), the Japanese supercomputers (NEC SX/2 through SX/5, Fujitsu VP200 through VPP5000, and the Hitachi S820 and S-8300), and the mini-supercomputers (Convex C-1 through C-4). 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 G.9) 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 G.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 VMIPS; its scalar portion is MIPS, and its vector portion is the logical vector extension of MIPS. The rest of this section examines how the basic architecture of VMIPS relates to other processors.

The primary components of the instruction set architecture of VMIPS are the following:

- *Vector registers*—Each vector register is a fixed-length bank holding a single vector. VMIPS 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 VMIPS. 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 8 write ports, are connected to the functional unit inputs or outputs by a pair of crossbars. (The description of the vector-register file design has been simplified here. Real machines make use of the regular access pattern within a vector instruction to reduce the costs of the vector-register file circuitry [Asanovic 1998]. For example, the Cray-1 manages to implement the register file with only a single port per register.)

- *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). VMIPS has five functional units, as shown in Figure G.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 VMIPS vector loads and stores are fully pipelined, so that words can be moved between the vector registers and memory

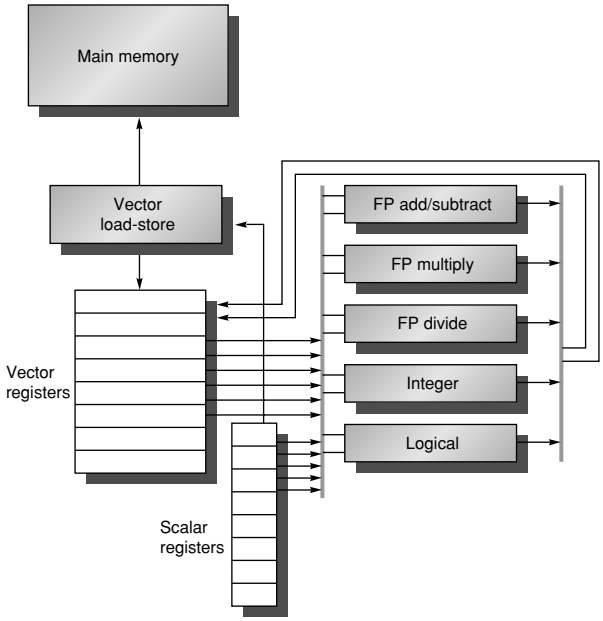

Figure G.1 The basic structure of a vector-register architecture, VMIPS. This processor has a scalar architecture just like MIPS. There are also eight 64-element vector registers, and all the functional units are vector functional units. Special vector instructions are defined both for arithmetic and for memory accesses. We show vector units for logical and integer operations. These are included so that VMIPS 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 G.4 we add chaining, which will require additional interconnect capability.

with a bandwidth of 1 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 MIPS. Scalar values are read out of the scalar register file, then latched at one input of the vector functional units.

Figure G.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. The last column in Figure G.2 shows the number of *lanes* in the machine, which is the number of parallel pipelines used to execute operations within each vector instruction. Lanes are described later in Section G.4; here we assume VMIPS has only a single pipeline per vector functional unit (one lane).

In VMIPS, vector operations use the same names as MIPS operations, but with the letter "V" appended. Thus, `ADDV.D` is an add of two double-precision vectors. The vector instructions take as their input either a pair of vector registers (`ADDV.D`) or a vector register and a scalar register, designated by appending "VS" (`ADDVS.D`). In the latter case, the value in the scalar register is used as the input for all operations—the operation `ADDVS.D` will add the contents of a scalar register to each element in a vector register. The scalar value is copied over to the vector functional unit at issue time. Most vector operations have a vector destination register, although 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 MIPS general-purpose register, is the starting address of the vector in memory. Figure G.3 lists the VMIPS 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 G.3 and G.4, respectively.

How Vector Processors Work: An Example

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

```
Y = a × 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 single-precision a × X plus Y; DAXPY for double-precision a × X plus Y.) Linpack is a collection of linear algebra routines, and the routines for performing Gaussian elimination constitute what is known as the

Processor (year)	Clock rate (MHz)	Vector registers	Elements per register (64-bit elements)	Vector arithmetic units	Vector load-store units	Lanes
Cray-1 (1976)	80	8	64	6: FP add, FP multiply, FP reciprocal, integer add, logical, shift	1	1
Cray X-MP (1983) Cray Y-MP (1988)	118 166	8	64	8: FP add, FP multiply, FP reciprocal, integer add, 2 logical, shift, population count/parity	2 loads 1 store	1
Cray-2 (1985)	244	8	64	5: FP add, FP multiply, FP reciprocal/ sqrt, integer add/shift/population count, logical	1	1
Fujitsu VP100/ VP200 (1982)	133	8–256	32–1024	3: FP or integer add/logical, multiply, divide	2	1 (VP100) 2 (VP200)
Hitachi S810/ S820 (1983)	71	32	256	4: FP multiply-add, FP multiply/ divide-add unit, 2 integer add/logical	3 loads 1 store	1 (S810) 2 (S820)
Convex C-1 (1985)	10	8	128	2: FP or integer multiply/divide, add/ logical	1	1 (64 bit) 2 (32 bit)
NEC SX/2 (1985)	167	8 + 32	256	4: FP multiply/divide, FP add, integer add/logical, shift	1	4
Cray C90 (1991) Cray T90 (1995)	240 460	8	128	8: FP add, FP multiply, FP reciprocal, integer add, 2 logical, shift, population count/parity	2 loads 1 store	2
NEC SX/5 (1998)	312	8 + 64	512	4: FP or integer add/shift, multiply, divide, logical	1	16
Fujitsu VPP5000 (1999)	300	8–256	128–4096	3: FP or integer multiply, add/logical, divide	1 load 1 store	16
Cray SV1 (1998) SV1ex (2001)	300 500	8	64	8: FP add, FP multiply, FP reciprocal, integer add, 2 logical, shift, population count/parity	1 load-store 1 load	2 8 (MSP)
VMIPS (2001)	500	8	64	5: FP multiply, FP divide, FP add, integer add/shift, logical	1 load-store	1

Figure G.2 Characteristics of several vector-register architectures. If the machine is a multiprocessor, the entries correspond to the characteristics of one processor. Several of the machines have different clock rates in the vector and scalar units; the clock rates shown are for the vector units. The Fujitsu machines' vector registers are configurable: The size and count of the 8K 64-bit entries may be varied inversely to one another (e.g., on the VP200, from eight registers each 1K elements long to 256 registers each 32 elements long). The NEC machines have eight foreground vector registers connected to the arithmetic units plus 32–64 background vector registers connected between the memory system and the foreground vector registers. The reciprocal unit on the Cray processors is used to do division (and square root on the Cray-2). Add pipelines perform 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, and several of the processors use the same units for FP scalar and FP vector operations. Each vector load-store unit represents the ability to do an independent, overlapped transfer to or from the vector registers. The number of lanes is the number of parallel pipelines in each of the functional units as described in Section G.4. For example, the NEC SX/5 can complete 16 multiplies per cycle in the multiply functional unit. The Convex C-1 can split its single 64-bit lane into two 32-bit lanes to increase performance for applications that require only reduced precision. The Cray SV1 can group four CPUs with two lanes each to act in unison as a single larger CPU with eight lanes, which Cray calls a Multi-Streaming Processor (MSP).

Instruction	Operands	Function
ADDV.D	V1,V2,V3	Add elements of V2 and V3, then put each result in V1.
ADDVS.D	V1,V2,F0	Add F0 to each element of V2, then put each result in V1.
SUBV.D	V1,V2,V3	Subtract elements of V3 from V2, then put each result in V1.
SUBVS.D	V1,V2,F0	Subtract F0 from elements of V2, then put each result in V1.
SUBSV.D	V1,F0,V2	Subtract elements of V2 from F0, then put each result in V1.
MULV.D	V1,V2,V3	Multiply elements of V2 and V3, then put each result in V1.
MULVS.D	V1,V2,F0	Multiply each element of V2 by F0, then put each result in V1.
DIVV.D	V1,V2,V3	Divide elements of V2 by V3, then put each result in V1.
DIVVS.D	V1,V2,F0	Divide elements of V2 by F0, then put each result in V1.
DIVSV.D	V1,F0,V2	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.D S--VS.D	V1,V2 V1,F0	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--VS.D 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.
MTC1 MFC1	VLR,R1 R1,VLR	Move contents of R1 to the vector-length register. Move the contents of the vector-length register to R1.
MVTM MVFM	VM,F0 F0,VM	Move contents of F0 to the vector-mask register. Move contents of vector-mask register to F0.

Figure G.3 The VMIPS 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 G.3) and VM (discussed in Section G.4). These special registers are assumed to live in the MIPS coprocessor 1 space along with the FPU registers. The operations with stride are explained in Section G.3, and the use of the index creation and indexed load-store operations are explained in Section G.4.

Linpack benchmark. The DAXPY routine, which implements the preceding 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 MIPS and VMIPS for the DAXPY loop. Assume that the starting addresses of X and Y are in Rx and Ry, respectively.

Answer Here is the MIPS code.

```
        L.D     F0,a        ;load scalar a
        DADDIU  R4,Rx,#512  ;last address to load
Loop:   L.D     F2,0(Rx)    ;load X(i)
        MUL.D   F2,F2,F0    ;a × X(i)
        L.D     F4,0(Ry)    ;load Y(i)
        ADD.D   F4,F4,F2    ;a × X(i) + Y(i)
        S.D     0(Ry),F4    ;store into Y(i)
        DADDIU  Rx,Rx,#8    ;increment index to X
        DADDIU  Ry,Ry,#8    ;increment index to Y
        DSUBU   R20,R4,Rx   ;compute bound
        BNEZ    R20,Loop    ;check if done
```

Here is the VMIPS code for DAXPY.

```
        L.D     F0,a        ;load scalar a
        LV      V1,Rx       ;load vector X
        MULVS.D V2,V1,F0    ;vector-scalar multiply
        LV      V3,Ry       ;load vector Y
        ADDV.D  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 MIPS. This reduction occurs both because the vector operations work on 64 elements and because the overhead instructions that constitute nearly half the loop on MIPS are not present in the VMIPS code.

Another important difference is the frequency of pipeline interlocks. In the straightforward MIPS code every `ADD.D` must wait for a `MUL.D`, and every `S.D` must wait for the `ADD.D`. On the vector processor, each vector instruction will only stall for the first element in each vector, and then subsequent elements will flow smoothly down the pipeline. Thus, pipeline stalls are required only once per vector operation, rather than once per vector element. In this example, the pipeline stall frequency on MIPS will be about 64 times higher than it is on VMIPS. The pipeline stalls can be eliminated on MIPS 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 operand vectors, 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. All modern supercomputers have vector functional units with multiple parallel pipelines (or *lanes*) that can produce two or more results per clock cycle, but may also have some functional units that are not fully pipelined. For simplicity, our VMIPS implementation has one lane with an initiation rate of one element per clock cycle for individual operations. 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. Placing vector instructions into a convoy is analogous to placing scalar operations into a VLIW instruction. 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 G.4 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 the unit of time taken to execute one convoy. 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
MULVS.D   V2,V1,F0    ;vector-scalar multiply
LV        V3,Ry       ;load vector Y
ADDV.D    V4,V2,V3    ;add
SV        Ry,V4       ;store the result
```

How many chimes will this vector sequence take? How many cycles per FLOP (floating-point operation) are needed ignoring vector instruction issue overhead?

Answer The first convoy is occupied by the first `LV` instruction. The `MULVS.D` 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 `MULVS.D`. The `ADDV.D` is dependent on the second `LV`, so it must come in yet a third convoy, and finally the `SV` depends on the `ADDV.D`, so it must go in a following convoy. This leads to the following layout of vector instructions into convoys:

1. `LV`
2. `MULVS.D` `LV`
3. `ADDV.D`
4. `SV`

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

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 G.4.

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 G.5 provides the answer in convoys, assuming that 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.

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

Figure G.4 Start-up overhead.

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

Figure G.5 Starting times and first- and last-result times for convoys 1 through 4. The vector length is n.

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 1 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 2 to 20 stages is not uncommon—although the most heavily used units have pipeline depths of 4–8 clock cycles.

For VMIPS, we will use the same pipeline depths as the Cray-1, although latencies in more modern processors have tended to increase, especially for loads. All functional units are fully pipelined. As shown in Figure G.6, pipeline depths are 6 clock cycles for floating-point add and 7 clock cycles for floating-point multiply. On VMIPS, as on most vector processors, independent vector operations using different functional units can issue in the same convoy.

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 1 clock cycle because memory bank stalls can reduce effective throughput.

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

Figure G.6 Start-up penalties on VMIPS. These are the start-up penalties in clock cycles for VMIPS vector operations.

Typically, penalties for start-ups on load-store units are higher than those for arithmetic functional units—over 100 clock cycles on some processors. For VMIPS we will assume a start-up time of 12 clock cycles, the same as the Cray-1. Figure G.6 summarizes the start-up penalties for VMIPS vector operations.

To maintain an initiation rate of 1 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 5.8. 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 three primary reasons:

1. Many vector computers support multiple loads or stores per clock, and the memory bank cycle time is often several times larger than the CPU cycle time. 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.
3. Many vector computers support multiple processors sharing the same memory system, and so each processor will be generating its own independent stream of addresses.

In combination, these features lead to a large number of independent memory banks, as shown by the following example.

Example The Cray T90 has a CPU clock cycle of 2.167 ns and in its largest configuration (Cray T932) has 32 processors each capable of generating four loads and two stores per CPU clock cycle. The CPU clock cycle is 2.167 ns, while the cycle time of the SRAMs used in the memory system is 15 ns. Calculate the minimum number of memory banks required to allow all CPUs to run at full memory bandwidth.

Answer The maximum number of memory references each cycle is 192 (32 CPUs times 6 references per CPU). Each SRAM bank is busy for 15/2.167 = 6.92 clock cycles, which we round up to 7 CPU clock cycles. Therefore we require a minimum of $192 \times 7 = 1344$ memory banks!

The Cray T932 actually has 1024 memory banks, and so the early models could not sustain full bandwidth to all CPUs simultaneously. A subsequent memory upgrade replaced the 15 ns asynchronous SRAMs with pipelined synchronous SRAMs that more than halved the memory cycle time, thereby providing sufficient bandwidth.

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 6 clocks. How many memory banks must we have to support one fetch per clock cycle? 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 G.7 shows the timing for the first few sets of accesses for an eight-bank system with a 6-clock-cycle access latency.

	Bank							
Cycle no.	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0		136						
1		busy	144					
2		busy	busy	152				
3		busy	busy	busy	160			
4		busy	busy	busy	busy	168		
5		busy	busy	busy	busy	busy	176	
6			busy	busy	busy	busy	busy	184
7	192			busy	busy	busy	busy	busy
8	busy	200			busy	busy	busy	busy
9	busy	busy	208			busy	busy	busy
10	busy	busy	busy	216			busy	busy
11	busy	busy	busy	busy	224			busy
12	busy	busy	busy	busy	busy	232		
13		busy	busy	busy	busy	busy	240	
14			busy	busy	busy	busy	busy	248
15	256			busy	busy	busy	busy	busy
16	busy	264			busy	busy	busy	busy

Figure G.7 Memory addresses (in bytes) by bank number and time slot at which access begins. Each memory bank latches the element address at the start of an access and is then busy for 6 clock cycles before returning a value to the CPU. Note that the CPU cannot keep all eight banks busy all the time because it is limited to supplying one new address and receiving one data item each cycle.

The timing of real memory banks is usually split into two different components, the access latency and the bank cycle time (or *bank busy time*). The access latency is the time from when the address arrives at the bank until the bank returns a data value, while the busy time is the time the bank is occupied with one request. The access latency adds to the start-up cost of fetching a vector from memory (the total memory latency also includes time to traverse the pipelined interconnection networks that transfer addresses and data between the CPU and memory banks). The bank busy time governs the effective bandwidth of a memory system because a processor cannot issue a second request to the same bank until the bank busy time has elapsed.

For simple unpipelined SRAM banks as used in the previous examples, the access latency and busy time are approximately the same. For a pipelined SRAM bank, however, the access latency is larger than the busy time because each element access only occupies one stage in the memory bank pipeline. For a DRAM bank, the access latency is usually shorter than the busy time because a DRAM needs extra time to restore the read value after the destructive read operation. 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.

G.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 VMIPS, 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 run time! 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 or equal to 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 that 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 G.8.

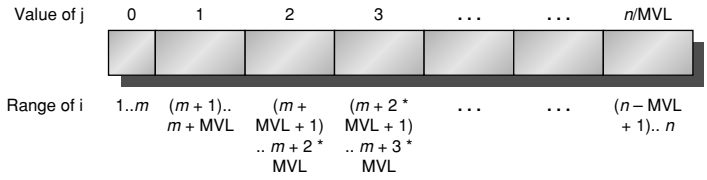

Figure G.8 A vector of arbitrary length processed with strip mining. All blocks but the first are of length MVL, utilizing the full power of the vector processor. In this figure, the variable *m* is used for the expression `(n mod MVL)`.

The inner loop of the preceding code 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 to the vector functional unit when a vector operation issues, in case VLR is changed for a subsequent vector operation.

Several vector ISAs have been developed that allow implementations to have different maximum vector-register lengths. For example, the IBM vector extension for the IBM 370 series mainframes supports an MVL of anywhere between 8 and 512 elements. A "load vector count and update" (VLVCU) instruction is provided to control strip-mined loops. The VLVCU instruction has a single scalar register operand that specifies the desired vector length. The vector-length register is set to the minimum of the desired length and the maximum available vector length, and this value is also subtracted from the scalar register, setting the condition codes to indicate if the loop should be terminated. In this way, object code can be moved unchanged between two different implementations while making full use of the available vector-register length within each strip-mined loop iteration.

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:

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

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 VMIPS. 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 VMIPS for the vector operation A = B × 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 MIPS (and VMIPS) R0 always holds 0. Since (200 mod 64) = 8, the first iteration of the strip-mined loop will execute for a vector length of 8 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 8 or 64, we increment the address registers by 8 × 8 = 64 after the first segment and 8 × 64 = 512 for later segments. The total number of bytes in the vector is 8 × 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:

```
        DADDUI   R2,R0,#1600  ;total # bytes in vector
        DADDU    R2,R2,Ra     ;address of the end of A vector
        DADDUI   R1,R0,#8     ;loads length of 1st segment
        MTC1     VLR,R1       ;load vector length in VLR
        DADDUI   R1,R0,#64    ;length in bytes of 1st segment
        DADDUI   R3,R0,#64    ;vector length of other segments
Loop:   LV       V1,Rb        ;load B
        MULVS.D  V2,V1,Fs     ;vector * scalar
        SV       Ra,V2        ;store A
        DADDU    Ra,Ra,R1     ;address of next segment of A
        DADDU    Rb,Rb,R1     ;address of next segment of B
        DADDUI   R1,R0,#512   ;load byte offset next segment
        MTC1     VLR,R3       ;set length to 64 elements
        DSUBU    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 7-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 G.4, we will be more ambitious—allowing overlapping of separate convoys.

Figure G.9 shows the overhead and effective rates per element for the previous example (A = B × s) with various vector lengths. A chime counting model would lead to 3 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.

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.

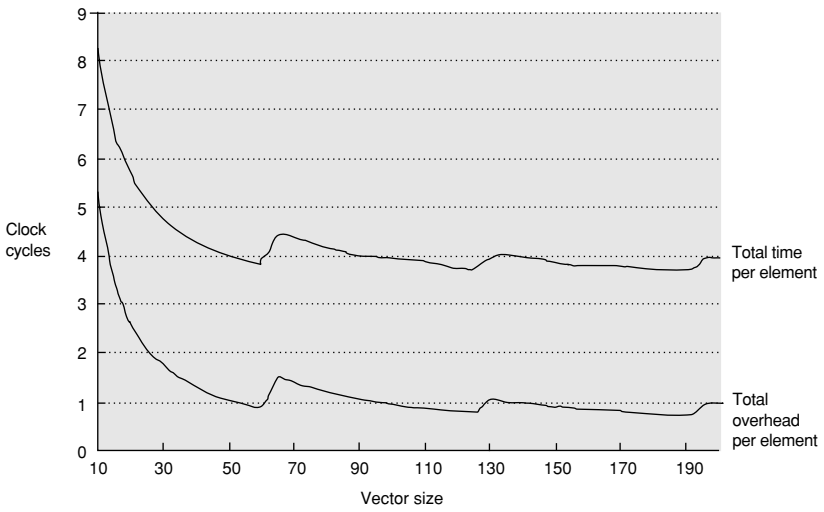

Figure G.9 The total execution time per element and the total overhead time per element versus the vector length for the example on page G-19. For short vectors the total start-up time is more than one-half of the total time, while for long vectors it reduces to about one-third of the total time. The 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 $T_{loop} + T_{start}$.

To do so, we must consider how adjacent elements in B and adjacent elements in C are addressed. As we discussed in Section 5.5, 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 preceding 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. For vector processors without caches, 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 with unit stride, 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 5.5), the ability to efficiently access data that is not contiguous remains an advantage for vector processors on certain problems.

On VMIPS, 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 VMIPS 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. Unit strides occur much more frequently than other strides and can benefit from special case handling in the memory system, and so are often separated from nonunit stride operations as in VMIPS.

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 bank busy time in clock cycles. Once nonunit strides are introduced, however, it becomes possible to request accesses from the same bank more frequently than the bank busy time allows. 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

$$\frac{\text{Number of banks}}{\text{Least common multiple (Stride, Number of banks)}} < \text{Bank busy time}$$

Example Suppose we have 8 memory banks with a bank busy time of 6 clocks and a total memory latency of 12 cycles. 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 bank busy time, 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 8 memory banks. Every access to memory (after the first one) will collide with the previous access and will have to wait for the 6-clock-cycle bank busy time. The total time will be 12 + 1 + 6 * 63 = 391 clock cycles, or 6.1 clocks per element.

Memory bank conflicts will not occur within a single vector memory instruction 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; with 64 banks, a stride of 8 will stall on every eighth access. If we have multiple memory pipelines and/or multiple processors sharing the same memory system, we will also need more banks to prevent conflicts. Even machines with a single memory pipeline can experience memory bank conflicts on unit stride accesses between the last few elements of one instruction and the first few elements of the next instruction, and increasing the number of banks will reduce the probability of these interinstruction conflicts. In 2001, 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, programmers favor unit stride accesses whenever possible.

A modern supercomputer may have dozens of CPUs, each with multiple memory pipelines connected to thousands of memory banks. It would be impractical to provide a dedicated path between each memory pipeline and each memory bank, and so typically a multistage switching network is used to connect memory pipelines to memory banks. Congestion can arise in this switching network as different vector accesses contend for the same circuit paths, causing additional stalls in the memory system.

G.4 Enhancing Vector Performance

In this section we present five techniques for improving the performance of a vector processor. The first, *chaining,* deals with making a sequence of dependent vector operations run faster, and originated in the Cray-1 but is now supported on most vector processors. The next two deal with expanding the class of loops that can be run in vector mode by combating the effects of conditional execution and sparse matrices with new types of vector instruction. The fourth technique increases the peak performance of a vector machine by adding more parallel execution units in the form of additional *lanes*. The fifth technique reduces start-up overhead by pipelining and overlapping instruction start-up.

Chaining—the Concept of Forwarding Extended to Vector Registers

Consider the simple vector sequence

```
MULV.D      V1,V2,V3
ADDV.D      V4,V1,V5
```

In VMIPS, 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.D` 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 simultaneous access to the same vector register by different vector instructions, which can be implemented either by adding more read and write ports or by organizing the vector-register file storage into interleaved banks in a similar way to the memory system. 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 previous sequence, 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

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

Figure G.10 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 G.10, 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 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 appli-

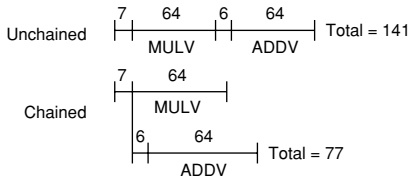

Figure G.10 Timings for a sequence of dependent vector operations ADDV and MULV, both unchained and chained. The 6- and 7-clock-cycle delays are the latency of the adder and multiplier.

cation. 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 VMIPS with only one vector load-store unit.

We will see in Section G.6 that chaining plays a major role in boosting vector performance. In fact, chaining is so important that every modern vector processor supports flexible chaining.

Conditionally Executed Statements

From Amdahl's Law, we know that the speedup on programs with low to moderate levels of vectorization 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. 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 $A(i) \neq 0$, 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 previous 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
L.D         F0,#0         ;load FP zero into F0
SNEVS.D     V1,F0         ;sets VM(i) to 1 if V1(i)!=F0
SUBV.D      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 0s 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 previous 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 nearly all modern vector processors. The instructions LVI (load vector indexed) and SVI (store vector indexed) provide these operations in VMIPS. For example, assuming that Ra, Rc, Rk, and Rm contain the starting addresses of the vectors in the previous 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.D      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. A simple vectorizing compiler could not automatically vectorize the source code above because the compiler would not 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.

More sophisticated vectorizing compilers can vectorize the loop automatically without programmer annotations by inserting run time checks for data dependences. These run time checks are implemented with a vectorized software version of the advanced load address table (ALAT) hardware described in Chapter 4 for the Itanium processor. The associative ALAT hardware is replaced with a software hash table that detects if two element accesses within the same strip-mine iteration

are to the same address. If no dependences are detected, the strip-mine iteration can complete using the maximum vector length. If a dependence is detected, the vector length is reset to a smaller value that avoids all dependency violations, leaving the remaining elements to be handled on the next iteration of the strip-mined loop. Although this scheme adds considerable software overhead to the loop, the overhead is mostly vectorized for the common case where there are no dependences, and as a result the loop still runs considerably faster than scalar code (although much slower than if a programmer directive was provided).

A scatter-gather capability is included on many of the recent supercomputers. These operations often run more slowly than strided accesses because they are more complex to implement and are more susceptible to bank conflicts, 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 VMIPS 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 VMIPS, we define the `CVI` instruction to always create a compressed index vector using the vector mask. When the vector mask is all 1s, 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 G-25:

```
LV          V1,Ra         ;load vector A into V1
L.D         F0,#0         ;load FP zero into F0
SNEVS.D     V1,F0         ;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
MTC1        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.D      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 G-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 $4n + 4fn + c_2$, where f is the fraction of elements for which the condition is true (i.e., $A(i) \neq 0$). 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.

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

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

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

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.

Multiple Lanes

One of the greatest advantages of a vector instruction set is that it allows software to pass a large amount of parallel work to hardware using only a single short instruction. A single vector instruction can include tens to hundreds of independent operations yet be encoded in the same number of bits as a conventional scalar instruction. The parallel semantics of a vector instruction allows an implementation to execute these elemental operations using either a deeply pipelined functional unit, as in the VMIPS implementation we've studied so far, or by using an array of parallel functional units, or a combination of parallel and pipelined functional units. Figure G.11 illustrates how vector performance can be improved by using parallel pipelines to execute a vector add instruction.

The VMIPS instruction set has been designed with the property that all vector arithmetic instructions only allow element *N* of one vector register to take part in operations with element *N* from other vector registers. This dramatically simplifies the construction of a highly parallel vector unit, which can be structured as multiple parallel *lanes*. As with a traffic highway, we can increase the peak throughput of a vector unit by adding more lanes. The structure of a four-lane vector unit is shown in Figure G.12.

Each lane contains one portion of the vector-register file and one execution pipeline from each vector functional unit. Each vector functional unit executes vector instructions at the rate of one element group per cycle using multiple pipelines, one per lane. The first lane holds the first element (element 0) for all vector registers, and so the first element in any vector instruction will have its source and destination operands located in the first lane. This allows the arithmetic pipeline local to the lane to complete the operation without communicating with other lanes. Interlane wiring is only required to access main memory. This lack of interlane communication reduces the wiring cost and register file ports required to build a highly parallel execution unit, and helps explains why current vector supercomputers can complete up to 64 operations per cycle (2 arithmetic units and 2 load-store units across 16 lanes).

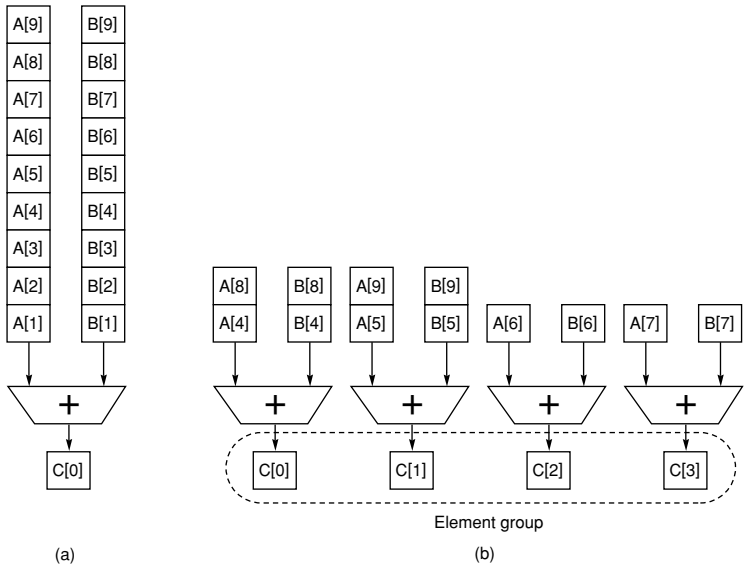

Figure G.11 Using multiple functional units to improve the performance of a single vector add instruction, C = A + B. The machine shown in (a) has a single add pipeline and can complete one addition per cycle. The machine shown in (b) has four add pipelines and can complete four additions per cycle. The elements within a single vector add instruction are interleaved across the four pipelines. The set of elements that move through the pipelines together is termed an *element group*. (Reproduced with permission from Asanovic [1998].)

Adding multiple lanes is a popular technique to improve vector performance as it requires little increase in control complexity and does not require changes to existing machine code. Several vector supercomputers are sold as a range of models that vary in the number of lanes installed, allowing users to trade price against peak vector performance. The Cray SV1 allows four two-lane CPUs to be ganged together using operating system software to form a single larger eight-lane CPU.

Pipelined Instruction Start-Up

Adding multiple lanes increases peak performance, but does not change start-up latency, and so it becomes critical to reduce start-up overhead by allowing the start of one vector instruction to be overlapped with the completion of preceding vector instructions. The simplest case to consider is when two vector instructions access a different set of vector registers. For example, in the code sequence

```
ADDV.D V1,V2,V3
ADDV.D V4,V5,V6
```

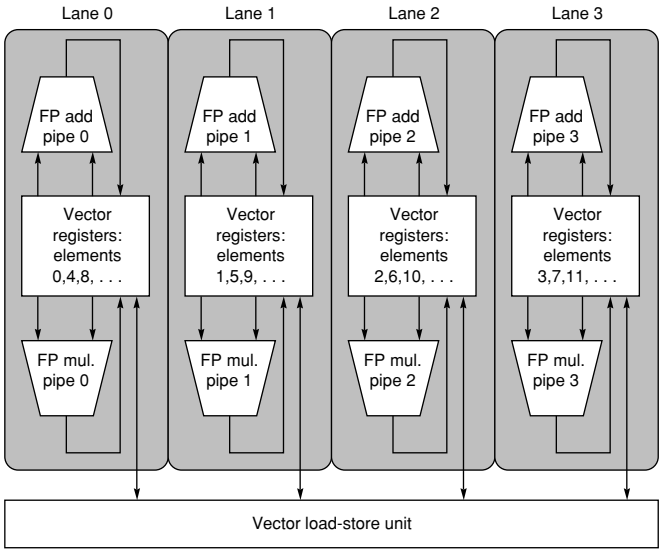

Figure G.12 Structure of a vector unit containing four lanes. The vector-register storage is divided across the lanes, with each lane holding every fourth element of each vector register. There are three vector functional units shown, an FP add, an FP multiply, and a load-store unit. Each of the vector arithmetic units contains four execution pipelines, one per lane, that act in concert to complete a single vector instruction. Note how each section of the vector-register file only needs to provide enough ports for pipelines local to its lane; this dramatically reduces the cost of providing multiple ports to the vector registers. The path to provide the scalar operand for vector-scalar instructions is not shown in this figure, but the scalar value must be broadcast to all lanes.

an implementation can allow the first element of the second vector instruction to immediately follow the last element of the first vector instruction down the FP adder pipeline. To reduce the complexity of control logic, some vector machines require some *recovery time* or *dead time* in between two vector instructions dispatched to the same vector unit. Figure G.13 is a pipeline diagram that shows both start-up latency and dead time for a single vector pipeline.

The following example illustrates the impact of this dead time on achievable vector performance.

Example The Cray C90 has two lanes but requires 4 clock cycles of dead time between any two vector instructions to the same functional unit, even if they have no data dependences. For the maximum vector length of 128 elements, what is the reduction in achievable peak performance caused by the dead time? What would be the reduction if the number of lanes were increased to 16?

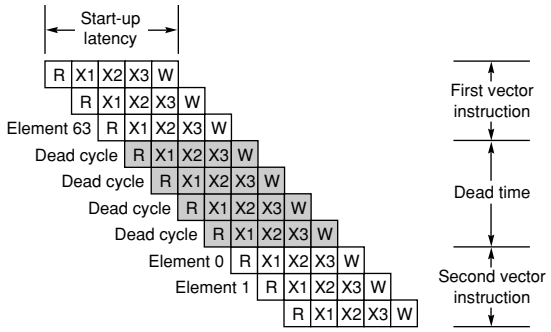

Figure G.13 Start-up latency and dead time for a single vector pipeline. Each element has a 5-cycle latency: 1 cycle to read the vector-register file, 3 cycles in execution, then 1 cycle to write the vector-register file. Elements from the same vector instruction can follow each other down the pipeline, but this machine inserts 4 cycles of dead time between two different vector instructions. The dead time can be eliminated with more complex control logic. (Reproduced with permission from Asanovic [1998].)

Answer A maximum length vector of 128 elements is divided over the two lanes and occupies a vector functional unit for 64 clock cycles. The dead time adds another 4 cycles of occupancy, reducing the peak performance to 64/(64 + 4) = 94.1% of the value without dead time. If the number of lanes is increased to 16, maximum length vector instructions will occupy a functional unit for only 128/16 = 8 cycles, and the dead time will reduce peak performance to 8/(8 + 4) = 66.6% of the value without dead time. In this second case, the vector units can never be more than 2/3 busy!

Pipelining instruction start-up becomes more complicated when multiple instructions can be reading and writing the same vector register, and when some instructions may stall unpredictably, for example, a vector load encountering memory bank conflicts. However, as both the number of lanes and pipeline latencies increase, it becomes increasingly important to allow fully pipelined instruction start-up.

G.5 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. These benchmarks are large, real scientific applications. Figure G.14 shows the percentage of operations executed in vector mode for two versions of the code running on the Cray Y-MP. The first version is that obtained with just compiler optimization on the original code, while the second version has been extensively hand-optimized by a team of Cray Research programmers. The wide variation in the level of compiler vectorization has been observed by several studies of the performance of applications on vector processors.

The hand-optimized versions generally show significant gains in vectorization level for codes the compiler could not vectorize well by itself, with all codes now above 50% vectorization. It is interesting to note that for MG3D, FLO52, and DYFESM, the faster code produced by the Cray programmers had *lower* levels of vectorization. The level of vectorization is not sufficient by itself to determine performance. Alternative vectorization techniques might execute fewer

Benchmark name	Operations executed in vector mode, compiler-optimized	Operations executed in vector mode, hand-optimized	Speedup from hand optimization
BDNA	96.1%	97.2%	1.52
MG3D	95.1%	94.5%	1.00
FLO52	91.5%	88.7%	N/A
ARC3D	91.1%	92.0%	1.01
SPEC77	90.3%	90.4%	1.07
MDG	87.7%	94.2%	1.49
TRFD	69.8%	73.7%	1.67
DYFESM	68.8%	65.6%	N/A
ADM	42.9%	59.6%	3.60
OCEAN	42.8%	91.2%	3.92
TRACK	14.4%	54.6%	2.52
SPICE	11.5%	79.9%	4.06
QCD	4.2%	75.1%	2.15

Figure G.14 Level of vectorization among the Perfect Club benchmarks when executed on the Cray Y-MP [Vajapeyam 1991]. The first column shows the vectorization level obtained with the compiler, while the second column shows the results after the codes have been hand-optimized by a team of Cray Research programmers. Speedup numbers are not available for FLO52 and DYFESM as the hand-optimized runs used larger data sets than the compiler-optimized runs.

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 G.15 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.

instructions, or keep more values in vector registers, or allow greater chaining and overlap among vector operations, and therefore improve performance even if the vectorization level remains the same or drops. For example, BDNA has almost the same level of vectorization in the two versions, but the hand-optimized code is over 50% faster.

There is also tremendous variation in how well different compilers do in vectorizing programs. As a summary of the state of vectorizing compilers, consider the data in Figure G.15, which shows the extent of vectorization for different processors using a test suite of 100 handwritten 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.

G.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 VMIPS 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 G.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 VMIPS. When chained, the inner loop of the DAXPY code in convoys looks like Figure G.16 (assuming that `Rx` and `Ry` hold starting addresses).

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

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

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 an MFLOPS rate of $2/3 \times$ clock rate (since there are 2 FLOPS per iteration). Thus, based only on the chime count, a 500 MHz VMIPS would run this loop at 333 MFLOPS assuming no strip-mining or start-up overhead. There are several ways to improve the performance: add additional vector

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

Figure G.16 The inner loop of the DAXPY code in chained convoys.

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, VMIPS's cousin, to boost the performance by 50%. Reducing the number of loads requires an interprocedural optimization; we examine this transformation in Exercise G.6. Before we examine the first two extensions, let's see what the real performance, including overhead, is.

The Peak Performance of VMIPS on DAXPY

First, we should determine what the peak performance, R_∞, really is, since we know it must differ from the ideal 333 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

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

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 500 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

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

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

$$R_\infty = \frac{2 \times 500 \text{ MHz}}{4} = 250 \text{ MFLOPS}$$

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 VMIPS 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.

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

$$R_{66} = \frac{2 \times 66 \times 500}{326} \text{ MFLOPS} = 202 \text{ MFLOPS}$$

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 VMIPS with a 500 MHz clock?

Answer Using R_∞ as the peak rate, we want to know the vector length that will achieve about 125 MFLOPS. We start with the formula for MFLOPS assuming that the measurement is made for $N_{1/2}$ elements:

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

$$125 = \frac{2 \times N_{1/2}}{T_{N_{1/2}}} \times 500$$

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

$$\begin{aligned} T_{N_{1/2}} &= 8 \times N_{1/2} \\ 1 \times 64 + 3 \times N_{1/2} &= 8 \times N_{1/2} \\ 5 \times N_{1/2} &= 64 \\ N_{1/2} &= 12.8 \end{aligned}$$

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

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 $T_{n \leq 64} = 64 + 3 \times n$ clock cycles. Therefore,

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

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 VMIPS

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 VMIPS 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, which 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 VMIPS if we added two more memory pipelines and allowed the strip-mining and start-up overheads 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,

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

$$R_\infty = \frac{2 \times 500 \text{ MHz}}{1} = 1000 \text{ MFLOPS}$$

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.

G.7 Fallacies and Pitfalls

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

Early memory-memory 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! On the CYBER 205, derived from the STAR-100, 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-lane CYBER 205 (the maximum-size processor) with the Cray X-MP that was delivered shortly after the 205, the 205 has a peak rate of 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 a 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 G.17 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 as measured by the harmonic mean. The next fallacy is closely related.

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

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

Figure G.17 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.

the Cray-1 became sufficient for a loop that formerly required more bandwidth. This ability to reuse values from vector registers is another advantage of vector-register architectures compared with memory-memory vector architectures, which have to fetch all vector operands from memory, requiring even greater memory bandwidth.

G.8 Concluding Remarks

During the 1980s and 1990s, rapid performance increases in pipelined scalar processors led to a dramatic closing of the gap between traditional vector supercomputers and fast, pipelined, superscalar VLSI microprocessors. In 2002, it is possible to buy a complete desktop computer system for under $1000 that has a higher CPU clock rate than any available vector supercomputer, even those costing tens of millions of dollars. Although the vector supercomputers have lower clock rates, they support greater parallelism through the use of multiple lanes (up to 16 in the Japanese designs) versus the limited multiple issue of the superscalar microprocessors. Nevertheless, the peak floating-point performance of the low-cost microprocessors is within a factor of 4 of the leading vector supercomputer CPUs. Of course, high clock rates and high peak performance do not necessarily translate into sustained application performance. Main memory bandwidth is the key distinguishing feature between vector supercomputers and superscalar microprocessor systems. The fastest microprocessors in 2002 can sustain around 1 GB/sec of main memory bandwidth, while the fastest vector supercomputers can sustain around 50 GB/sec per CPU. For nonunit stride accesses the bandwidth discrepancy is even greater. For certain scientific and engineering applications, performance correlates directly with nonunit stride main memory bandwidth, and these are the applications for which vector supercomputers remain popular.

Providing this large nonunit stride memory bandwidth is one of the major expenses in a vector supercomputer, and traditionally SRAM was used as main memory to reduce the number of memory banks needed and to reduce vector start-up penalties. While SRAM has an access time several times lower than that of DRAM, it costs roughly 10 times as much per bit! To reduce main memory costs and to allow larger capacities, all modern vector supercomputers now use DRAM for main memory, taking advantage of new higher-bandwidth DRAM interfaces such as synchronous DRAM.

This adoption of DRAM for main memory (pioneered by Seymour Cray in the Cray-2) is one example of how vector supercomputers are adapting commodity technology to improve their price-performance. Another example is that vector supercomputers are now including vector data caches. Caches are not effective for all vector codes, however, and so these vector caches are designed to allow high main memory bandwidth even in the presence of many cache misses. For example, the cache on the Cray SV1 can support 384 outstanding cache misses per CPU, while for microprocessors 8–16 outstanding misses is a more typical maximum number.

Another example is the demise of bipolar ECL or gallium arsenide as technologies of choice for supercomputer CPU logic. Because of the huge investment in CMOS technology made possible by the success of the desktop computer, CMOS now offers competitive transistor performance with much greater transistor density and much reduced power dissipation compared with these more exotic technologies. As a result, all leading vector supercomputers are now built with the same CMOS technology as superscalar microprocessors. The primary reason that vector supercomputers now have lower clock rates than commodity microprocessors is that they are developed using standard cell ASIC techniques rather than full custom circuit design to reduce the engineering design cost. While a microprocessor design may sell tens of millions of copies and can amortize the design cost over this large number of units, a vector supercomputer is considered a success if over a hundred units are sold!

Conversely, superscalar microprocessor designs have begun to absorb some of the techniques made popular in earlier vector computer systems. Many multimedia applications contain code that can be vectorized, and as discussed in Chapter 2, most commercial microprocessor ISAs have added multimedia extensions that resemble short vector instructions. A common technique is to allow a wide 64-bit register to be split into smaller subwords that are operated on in parallel. This idea was used in the early TI ASC and CDC STAR-100 vector machines, where a 64-bit lane could be split into two 32-bit lanes to give higher performance on lower-precision data. Although the initial microprocessor multimedia extensions were very limited in scope, newer extensions such as AltiVec for the IBM/Motorola PowerPC and SSE2 for the Intel x86 processors have both increased the vector length to 128 bits (still small compared with the 4096 bits in a VMIPS vector register) and added better support for vector compilers. Vector instructions are particularly appealing for embedded processors because they support high degrees of parallelism at low cost and with low power dissipation, and have been used in several game machines such as the Nintendo-64 and the Sony Playstation 2 to boost graphics performance. We expect that microprocessors will continue to extend their support for vector operations, as this represents a much simpler approach to boosting performance for an important class of applications compared with the hardware complexity of increasing scalar instruction issue width, or the software complexity of managing multiple parallel processors.

G.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 and to reduce memory bandwidth requirements. 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 lanes, 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 floating-point operation), 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 ns 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 256M word (64-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 the uniprocessor C-1 vector processor and then offered a series of small multiprocessors ending with the C-4 announced in 1994. The keys to the success of Convex over this period were their emphasis on Cray software capability, the effectiveness of their compiler (see Figure G.15), and the quality of their UNIX OS implementation. The C-4 was the last vector machine Convex sold; they switched to making large-scale multiprocessors using Hewlett-Packard RISC microprocessors and were bought by HP in 1995. 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 the early 1980s, CDC spun out a group, called ETA, to build a new supercomputer, the ETA-10, capable of 10 GFLOPS. The ETA processor was 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.

In 1983, processor vendors from Japan entered the supercomputer marketplace, starting with the Fujitsu VP100 and VP200 (see 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 1.32 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. A single Cray-3 prototype was delivered to the National Center for Atmospheric Research (NCAR) for evaluation purposes in 1993, but no paying customers were found for the design. The Cray-4 prototype, which was to have been the first processor to run at 1 GHz, was close to completion when the company filed for bankruptcy. Shortly before his tragic death in a car accident in 1996, Seymour Cray started yet another company, SRC Computers, to develop high-performance systems but this time using commodity components. In 2000, SRC announced the SRC-6 system that combines 512 Intel microprocessors, 5 billion gates of reconfigurable logic, and a high-performance vector-style memory system.

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, employing up to 2048 Digital Alpha 21064 microprocessors. 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 (460 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 $2.5 million. The T90 was the last bipolar ECL vector machine built by Cray. 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, Cray Research was acquired by Silicon Graphics, and in 1998 released the SV1 system, which grafted considerably faster CMOS processors onto the J90 memory system, and which also added a data cache for vectors to each CPU to help meet the increased memory bandwidth demands. Silicon Graphics sold Cray Research to Tera Computer in 2000, and the joint company was renamed Cray Inc. Cray Inc. plans to release the SV2 in 2002, which will be based on a completely new vector ISA.

The Japanese supercomputer makers have continued to evolve their designs and have generally placed greater emphasis on increasing the number of lanes in their vector units. In 2001, the NEC SX/5 was generally held to be the fastest available vector supercomputer, with 16 lanes clocking at 312 MHz and with up to 16 processors sharing the same memory. The Fujitsu VPP5000 was announced in 2001 and also had 16 lanes and clocked at 300 MHz, but connected up to 128 processors in a distributed-memory cluster. In 2001, Cray Inc. announced that they would be marketing the NEC SX/5 machine in the United States, after many years in which Japanese supercomputers were unavailable to U.S. customers after the U.S. Commerce Department found NEC and Fujitsu guilty of bidding below cost for a 1996 NCAR supercomputer contract and imposed heavy import duties on their products.

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] give 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 released a set of benchmarks, SPEChpc96, aimed at high-end scientific processing in 1996. The NAS parallel benchmarks developed at the NASA Ames Research Center [Bailey et al. 1991] have become a popular set of kernels and applications used for supercomputer evaluation.

In less than 30 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. However, the enormous price-performance advantages of microprocessor technology are bringing this era to an end. Advanced superscalar microprocessors are approaching the peak performance of the fastest vector processors, and in 2001, most of the highest-performance machines in the world were large-scale multiprocessors based on these microprocessors. Vector supercomputers remain popular for certain applications including car crash simulation and weather prediction that rely heavily on scatter-gather performance over large data sets and for which effective massively parallel programs have yet to be written. Over time, we expect that microprocessors will support higher-bandwidth memory systems, and that more applications will be parallelized and/or tuned for cached multiprocessor systems. As the set of applications best suited for vector supercomputers shrinks, they will become less viable as commercial products and will eventually disappear. But vector processing techniques will likely survive as an integral part of future microprocessor architectures, with the currently popular SIMD multimedia extensions representing the first step in this direction.

References

Alliant Computer Systems Corp. [1987]. *Alliant FX/Series: Product Summary* (June), Acton, Mass.

Asanovic, K. [1998]. "Vector microprocessors," Ph.D. thesis, Computer Science Division, Univ. of California at Berkeley (May).

Bailey, D. H., E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga [1991]. "The NAS parallel benchmarks," *Int'l. J. Supercomputing Applications* 5, 63–73.

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.

Brandt, M., J. Brooks, M. Cahir, T. Hewitt, E. Lopez-Pineda, and D. Sandness [2000]. *The Benchmarker's Guide for Cray SV1 Systems*. Cray Inc., Seattle, Wash.

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.

Vajapeyam, S. [1991]. "Instruction-level characterization of the Cray Y-MP processor," Ph.D. thesis, Computer Sciences Department, University of Wisconsin-Madison.

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 VMIPS has a clock rate of 500 MHz and that T_{loop} = 15. Use the start-up times from Figure G.4, and assume that the store latency is always included in the running time.

G.1 [10] <G.1, G.2> Write a VMIPS vector sequence that achieves the peak MFLOPS performance of the processor (use the functional unit and instruction description in Section G.2). Assuming a 500-MHz clock rate, what is the peak MFLOPS?

G.2 [20/15/15] <G.1–G.6> Consider the following vector code run on a 500-MHz version of VMIPS for a fixed vector length of 64:

```
LV       V1,Ra
MULV.D   V2,V1,V3
ADDV.D   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] <G.1–G.4> 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] <G.1–G.4> If the vector sequence is chained, how many clock cycles per result does this sequence require, including overhead?

c. [15] <G.1–G.6> Suppose VMIPS 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?

G.3 [20/20/15/15/20/20/20] <G.2–G.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 G.6 to estimate performance throughout this exercise, assuming a 500-MHz version of VMIPS.

a. [20] <G.2–G.6> Write the best VMIPS 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] <G.2–G.6> Find the total time for this loop on VMIPS (T_{100}). What is the MFLOPS rating for the loop (R_{100})?

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

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

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

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

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

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

G.5 [12/12] <G.5–G.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] <G.5–G.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] <G.5–G.6> Write the VMIPS vector code for the loop without a dependence.

G.6 [20/15/20/20] <G.5–G.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] <G.5–G.6> Write the *straightforward* code sequence for just the inner loop in VMIPS vector instructions.

b. [15] <G.5–G.6> Using the techniques of Section G.6, estimate the performance of this code on VMIPS 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] <G.5–G.6> Rewrite the VMIPS code to reduce the performance limitation; show the resulting inner loop in VMIPS vector instructions. (*Hint:* Think about what establishes T_{chime}; can you affect it?) Find the total time for the resulting sequence.

d. [20] <G.5–G.6> Estimate the performance of your new version, using the techniques of Section G.6 and finding T_{64}.

G.7 [15/15/25] <G.4> 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] <G.4> Write the VMIPS code for this loop using the vector-mask capability.

b. [15] <G.4> Write the VMIPS code for this loop using scatter-gather.

c. [25] <G.4> 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?

G.8 [15/20/15/15] <G.1–G.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 VMIPS 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 VMIPS (call it VMIPS-II) that has two copies of every floating-point functional unit with full chaining among them. Assume that both VMIPS and VMIPS-II have two load-store units. Because of the extra func-

tional units and the increased complexity of assigning operations to units, all the overheads (T_{loop} and T_{start}) are doubled.

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

b. [20] <G.1–G.6> Find the number of clock cycles on code sequence 1 for VMIPS-II. How does this compare to VMIPS?

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

d. [15] <G.1–G.6> Find the number of clock cycles on code sequence 2 for VMIPS-II. How does this compare to VMIPS?

G.9 [20] <G.5> 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
```

G.10 [12/15] <G.5> Consider the following loop:

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

a. [12] <G.5> Show there is a loop-carried dependence in this code fragment.

b. [15] <G.5> Rewrite the code in FORTRAN so that it can be vectorized as two separate vector sequences.

G.11 [15/25/25] <G.5> As we saw in Section G.5, 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] <G.5> 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 preceding code fragment using recursive doubling.

b. [25] <G.5> 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 VMIPS 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 VMIPS code sequence that reduces the contents of V1 to eight partial sums. It can be done with one vector operation.

c. Discuss how adding the extension in part (b) would affect a machine that had multiple lanes.

G.12 [40] <G.2–G.4> Extend the MIPS simulator to be a VMIPS simulator, including the ability to count clock cycles. Write some short benchmark programs in MIPS and VMIPS assembly language. Measure the speedup on VMIPS, the percentage of vectorization, and usage of the functional units.

G.13 [50] <G.5> Modify the MIPS compiler to include a dependence checker. Run some scientific code and loops through it and measure what percentage of the statements could be vectorized.

G.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?

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