Instruction type	Pipe	Stages						
Integer instruction	IF	ID	EX	MEM	WB			
FP instruction	IF	ID	EX	MEM	WB			
Integer instruction		IF	ID	EX	MEM	WB		
FP instruction		IF	ID	EX	MEM	WB		
Integer instruction			IF	ID	EX	MEM	WB	
FP instruction			IF	ID	EX	MEM	WB	
Integer instruction				IF	ID	EX	MEM	WB
FP instruction				IF	ID	EX	MEM	WB

FIGURE 6.46 Superscalar pipeline in operation. The integer and floating-point instructions are issued at the same time, and each executes at its own pace through the pipeline. This scheme will only improve the performance of programs with a fair amount of floating point.

By issuing an integer and a floating-point operation in parallel, the need for additional hardware is minimized—integer and floating-point operations use different register sets and different functional units. The only conflict arises when the integer instruction is a floating-point load, store, or move. This creates contention for the floating-point register ports and may also create a hazard if the floating-point operation uses the result of a floating-point load issued at the same time. Both problems could be solved by detecting this contention as a structural hazard and delaying the issue of the floating-point instruction. The contention could also be eliminated by providing two additional ports, a read and a write, on the floating-point register file. We would also need to add several additional bypass paths to avoid performance loss.

There is another difficulty that may limit the effectiveness of a superscalar pipeline. In our simple DLX pipeline, loads had a latency of one clock cycle; this prevented one instruction from using the result without stalling. In the superscalar pipeline, the result of a load instruction cannot be used on the same clock cycle or on the next clock cycle. This means that the next three instructions cannot use the load result without stalling; without extra ports, moves between the register sets are similarly affected. The branch delay also becomes three instructions. To effectively exploit the parallelism available in a superscalar machine, more ambitious compiler-scheduling techniques, as well as more complex instruction decoding, will need to be implemented. Loop unrolling helps generate larger straightline fragments for scheduling; more powerful compiler techniques are discussed near the end of this section.

Let's see how well loop unrolling and scheduling work on a superscalar version of DLX with the same delays in clock cycles.

Example

How would the unrolled loop on page 317 be scheduled on a superscalar pipeline for DLX? To schedule it without any delays, we will need to unroll it to make five copies of the body.

Answer

The resulting code is shown in Figure 6.47.

	Integer instruction		FP instruction	Clock cycle
Loop:	LD	F0,0(R1)		1
	LD	F6,-8(R1)		2
	LD	F10,-16(R1)	ADDD F4,F0,F2	3
	LD	F14,-24(R1)	ADDD F8,F6,F2	4
	LD	F18,-32(R1)	ADDD F12,F10,F2	5
	SD	0(R1),F4	ADDD F16,F14,F2	6
	SD	-8(R1),F8	ADDD F20,F18,F2	7
	SD	-16(R1),F12		8
	SD	-24(R1),F16		9
	SUB	R1,R1,#40		10
	BNEZ	R1,LOOP		11
	SD	8(R1),F20		12

FIGURE 6.47 The unrolled and scheduled code as it would look on a superscalar DLX.

This unrolled superscalar loop now runs in 12 clock cycles per iteration, or 2.4 clock cycles per element, versus 3.5 for the scheduled and unrolled loop on the ordinary DLX pipeline. In this example, the performance of the superscalar DLX is limited by the balance between integer and floating-point computation. Every floating-point instruction is issued together with an integer instruction, but there are not enough floating-point instructions to keep the floating-point pipeline full. When scheduled, the original loop ran in 6 clock cycles per iteration. We have improved on that by a factor of 2.5, more than half of which came from loop unrolling, which took us from 6 to 3.5, with the rest coming from issuing more than one instruction per clock cycle.

Ideally, our superscalar machine will pick up two instructions and issue them both if the first is an integer and the second is a floating-point instruction. If they do not fit this pattern, which can be quickly detected, then they are issued sequentially. This points to one of the major advantages of a general superscalar machine: There is little impact on code density, and even unscheduled programs can be run. The number of issues and classes of instructions that can be issued together are the major factors that differentiate superscalar processors.

Multiple Instruction Issue with Dynamic Scheduling

Multiple instruction issue can also be applied to dynamically scheduled machines. We could start with either the scoreboard scheme or Tomasulo's algorithm. Let's assume we want to extend Tomasulo's algorithm to support issuing two instructions per clock cycle, one integer and one floating point. We do not want to issue instructions in the queue out of order, since this makes the bookkeeping in the register file impossible. Rather, by employing data structures for the integer and floating-point registers, both types of instructions can be issued to their respective reservation stations, as long as the two instructions at the head of the instruction queue do not access the same register set. Unfortunately, this approach bars issuing two instructions with a dependence in the same clock cycle. This is, of course, true in the superscalar case, where it is clearly the compiler's problem. There are three approaches that can be used to achieve dual issue. First, we could use software scheduling to ensure that dependent instructions do not appear adjacent. However, this would require pipeline-scheduling software, thereby defeating one of the advantages of dynamically scheduled pipelines.

A second approach is to pipeline the instruction-issue stage so that it runs twice as fast as the basic clock rate. This permits updating the tables before processing the next instruction; then the two instructions can begin execution at once.

The third approach is based on the observation that if multiple instructions are not being issued to the same functional unit, then it will only be loads and stores that will create dependences among instructions that we wish to issue together. The need for reservation tables for loads and stores can be eliminated by using queues for the result of a load and for the source operand of a store. Since dynamic scheduling is most effective for loads and stores, while static scheduling is highly effective in register–register code sequences, we could use static scheduling to eliminate reservation stations completely and rely only on the queues for loads and stores. This style of machine organization has been called a *decoupled architecture*.

For simplicity, let us assume that we have pipelined the instruction issue logic so that we can issue two operations that are dependent but use different functional units. Let's see how this would work with our example.

Example

Consider the execution of our simple loop on a DLX pipeline extended with Tomasulo's algorithm and with multiple issue. Assume that both a floating-point and an integer operation can be issued on every clock cycle, even if they are related. The number of cycles of latency per instruction is the same. Assume that issue and write results take one cycle each, and that there is dynamic branch-prediction hardware. Create a table showing when each instruction issues, begins execution, and writes its result, for the first two iterations of the loop. Here is the original loop:

```
Loop:  LD    F0,0(R1)
       ADDD  F4,F0,F2
       SD    0(R1),F4
       SUB   R1,R1,#8
       BNEZ  R1,LOOP
```

Answer The loop will be dynamically unwound and, whenever possible, instructions will be issued in pairs. The result is shown in Figure 6.48. The loop runs in $4 + \frac{7}{n}$ clock cycles per result for *n* iterations. For large *n* this approaches 4 clock cycles per result.

Iteration number	Instructions	Issues at clock-cycle number	Executes at clock-cycle number	Writes result at clock-cycle number
1	`LD F0,0(R1)`	1	2	4
1	`ADDD F4,F0,F2`	1	5	8
1	`SD 0(R1),F4`	2	9	
1	`SUB R1,R1,#8`	3	4	5
1	`BNEZ R1,LOOP`	4	5	
2	`LD F0,0(R1)`	5	6	8
2	`ADDD F4,F0,F2`	5	9	12
2	`SD 0(R1),F4`	6	13	
2	`SUB R1,R1,#8`	7	8	9
2	`BNEZ R1,LOOP`	8	9	

FIGURE 6.48 The time of issue, execution, and writing result for a dual-issue version of our Tomasulo pipeline. The write-result stage does not apply to either stores or branches, since they do not write any registers.

The number of dual issues is small because there is only one floating-point operation per iteration. The relative number of dual-issued instructions would be helped by the compiler partially unwinding the loop to reduce the instruction count by eliminating loop overhead. With that transformation, the loop would run as fast as on a superscalar machine. We will return to this transformation in Exercises 6.16 and 6.17.

The VLIW Approach

Our superscalar DLX machine can issue two instructions per clock cycle. That could perhaps be extended to three or at most four, but it becomes difficult to determine whether three or four instructions can all issue simultaneously without knowing what order the instructions could be in when fetched and what dependencies might exist among them. An alternative is an LIW (*Long Instruction Word*) or VLIW (*Very Long Instruction Word*) architecture. VLIWs use multiple, independent functional units. Rather than attempting to issue multiple, independent instructions to the units, a VLIW packages the multiple operations into one very long instruction, hence the name. A VLIW instruction might include two integer operations, two floating-point operations, two memory references, and a branch. An instruction would have a set of fields for each functional unit—perhaps 16 to 24 bits per unit, yielding an instruction length of between 112 and 168 bits. To keep the functional units busy there must be enough work in a straightline code sequence to keep the instructions scheduled. This is accomplished by unrolling loops and scheduling code across basic blocks using a technique called *trace scheduling*. In addition to eliminating branches by unrolling loops, trace scheduling provides a method to move instructions across branch points. We will discuss trace scheduling more in the next section. For now, let's assume we have a technique to generate long, straightline code sequences for building up VLIW instructions.

Example

Suppose we have a VLIW that could issue two memory references, two FP operations, and one integer operation or branch in every clock cycle. Show an unrolled version of the vector sum loop for such a machine. Unroll as many times as necessary to eliminate any stalls. Ignore the branch-delay slot.

Answer

The code is shown in Figure 6.49. The loop has been unrolled 6 times, which eliminates stalls, and runs in 9 cycles. This yields a running rate of 7 results in 9 cycles, or 1.28 cycles per result.

Memory reference 1	Memory reference 2	FP operation 1	FP operation 2	Integer operation / branch
`LD F0,0(R1)`	`LD F6,-8(R1)`			
`LD F10,-16(R1)`	`LD F14,-24(R1)`			
`LD F18,-32(R1)`	`LD F22,-40(R1)`	`ADDD F4,F0,F2`	`ADDD F8,F6,F2`	
`LD F26,-48(R1)`		`ADDD F12,F10,F2`	`ADDD F16,F14,F2`	
		`ADDD F20,F18,F2`	`ADDD F24,F22,F2`	
`SD 0(R1),F4`	`SD -8(R1),F8`	`ADDD F28,F26,F2`		
`SD -16(R1),F12`	`SD -24(R1),F16`			
`SD -32(R1),F20`	`SD -40(R1),F24`			`SUB R1,R1,#48`
`SD -0(R1),F28`				`BNEZ R1,LOOP`

FIGURE 6.49 VLIW instructions that occupy the inner loop and replace the unrolled sequence. This code takes nine cycles assuming no branch delay; normally the branch would also be scheduled. The issue rate is 23 operations in 9 clock cycles, or 2.5 operations per cycle. The efficiency, the percentage of available slots that contained an operation, is about 60%. To achieve this issue rate requires a much larger number of registers than DLX would normally use in this loop.

What are the limitations and costs of a VLIW approach? If we can issue 5 operations per clock cycle, why not 50? Three different limitations are encountered: limited parallelism, limited hardware resources, and code size explosion. The first is the simplest: There is a limited amount of parallelism available in instruction sequences. Unless loops are unrolled very large numbers of times, there may not be enough operations to fill the instructions. At first glance, it might appear that 5 instructions that could be executed in parallel would be sufficient to keep our VLIW completely busy. This, however, is not the case. Several of these functional units—the memory, the branch, and the floating-point units—will be pipelined, requiring a much larger number of operations that can be executed in parallel. For example, if the floating-point pipeline has 8 steps, the 2 operations being issued on a clock cycle cannot depend on any of the 14 operations already in the floating-point pipeline. Thus, we need to find a number of independent operations roughly equal to the average pipeline depth times the number of functional units. This means about 15 to 20 operations would be needed to keep a VLIW with 5 functional units busy.

The second cost, the hardware resources for a VLIW, seem quite straightforward; duplicating the floating-point and integer functional units is easy and cost scales linearly. However, there is a large increase in the memory- and register-file bandwidth. Even with a split floating-point and integer register file, our VLIW will require 5 read ports and 2 write ports on the integer register file and 4 read ports and 2 write ports on the floating-point register file. This bandwidth cannot be supported without some substantial cost in the size of the register file and possible degradation of clock speed. Our 5-unit VLIW also has 2 data memory ports. Furthermore, if we wanted to expand it, we would need to continue adding memory ports. Adding only arithmetic units would not help, since the machine would be starved for memory bandwidth. As the number of data memory ports grows, so does the complexity of the memory system. To allow multiple memory accesses in parallel, the memory must be broken into banks containing different addresses with the hope that the operations in a single instruction do not have conflicting accesses. A conflict will cause the entire machine to stall, since all the functional units must be kept synchronized. This same factor makes it extremely difficult to use data caches in a VLIW.

Finally, there is the problem of code size. There are two different elements that combine to increase code size substantially. First, generating enough operations in a straightline code fragment requires ambitiously unrolling loops, which increases code size. Second, whenever instructions are not full, the unused functional units translate to wasted bits in the instruction encoding. In Figure 6.49, we saw that only about 60% of the functional units were used; almost half of each instruction was empty. To combat this problem, clever encodings are sometimes used. For example, there may be only one large immediate field for use by any functional unit. Another technique is to compress the instructions in main memory and expand them when they are read into the cache or are decoded.

The major challenge for these machines is to try to exploit large amounts of instruction-level parallelism. When the parallelism comes from unrolling simple loops, the original loop probably could have been run efficiently on a vector machine (see the next chapter). It is not clear that a VLIW is preferred over a vector machine for such applications; the costs are similar, and the vector machine is typically the same speed or faster. The open question in 1990 is whether there are large classes of applications that are not suitable for vector machines, but still offer enough parallelism to justify the VLIW approach rather than a simpler one, such as a superscalar machine.

Increasing Instruction-Level Parallelism with Software Pipelining and Trace Scheduling

We have already seen that one compiler technique, loop unrolling, is used to help exploit parallelism among instructions. Loop unrolling creates longer sequences of straightline code, which can be used to exploit more instruction-level parallelism. There are two other more general techniques that have been developed for this purpose: software pipelining and trace scheduling.

Software pipelining is a technique for reorganizing loops such that each iteration in the software-pipelined code is made from instruction sequences chosen from different iterations in the original code segment. This is most easily understood by looking at the scheduled code for the superscalar version of DLX. The scheduler essentially interleaves instructions from different loop iterations, putting together all the loads, then all the adds, then all the stores. A software-pipelined loop interleaves instructions from different iterations without unrolling the loop. This technique is the software counterpart to what Tomasulo's algorithm does in hardware. The software-pipelined loop would contain one load, one add, and one store, each from a different iteration. There is also some startup code that is needed before the loop begins as well as code to finish up after the loop is completed. We will ignore these in this discussion.

Example

Show a software-pipelined version of this loop:

```
Loop:  LD    F0,0(R1)
       ADDD  F4,F0,F2
       SD    0(R1),F4
       SUB   R1,R1,#8
       BNEZ  R1,LOOP
```

You may omit the start-up and clean-up code.

Answer Given the vector M in memory, and ignoring the start-up and finishing code, we have:

```
Loop:   SD   0(R1),F4    ;stores into M[i]
        ADDD F4,F0,F2    ;adds to M[i-1]
        LD   F0,-16(R1)  ;loads M[i-2]
        BNEZ R1,LOOP
        SUB  R1,R1,#8    ;subtract in delay slot
```

This loop can be run at a rate of 5 cycles per result, ignoring the start-up and clean-up portions. Because the load fetches two array elements beyond the element count, the loop should run for two fewer iterations. This would be accomplished by decrementing R1 by 16 prior to the loop.

Software pipelining can be thought of as symbolic loop unrolling. Indeed, some of the algorithms for software pipelining use loop unrolling to figure out how to software pipeline the loop. The major advantage of software pipelining over straight loop unrolling is that software pipelining consumes less code space. Software pipelining and loop unrolling, in addition to yielding a better scheduled inner loop, each reduce a different type of overhead. Loop unrolling reduces the overhead of the loop—the branch and counter-update code. Software pipelining reduces the time when the loop is not running at peak speed to once per loop at the beginning and end. If we unroll a loop that does 100 iterations a constant number of times, say 4, we pay the overhead 100/4 = 25 times—every time the inner unrolled loop is reinitiated. Figure 6.50 shows this behavior graphically. Because these techniques attack two different types of overhead, the best performance comes from doing both.

The other technique used to generate additional parallelism is *trace scheduling*. This is particularly useful for VLIWs, for which the technique was originally developed. Trace scheduling is a combination of two separate processes. The first process, called *trace selection* tries to find the most likely sequence of operations to put together into a small number of instructions; this sequence is called a *trace*. Loop unrolling is used to generate long traces, since loop branches are taken with high probability. Once a trace is selected, the second process, called *trace compaction*, tries to squeeze the trace into a small number of wide instructions. Trace compaction attempts to move operations as early as it can in a sequence (trace), packing the operations into as few wide instructions as possible.

There are two different considerations in compacting a trace: data dependences, which force a partial order on operations, and branch points, which create places across which code cannot be easily moved. In essence, the code wants to be compacted into the shortest possible sequence that preserves the data dependences; branches are the main impediment to this process. The major advantage of trace scheduling over simpler pipeline-scheduling techniques is that it includes a method to move code across branches. Figure 6.51 shows a code fragment, which may be thought of as an iteration of an unrolled loop, and the trace selected.

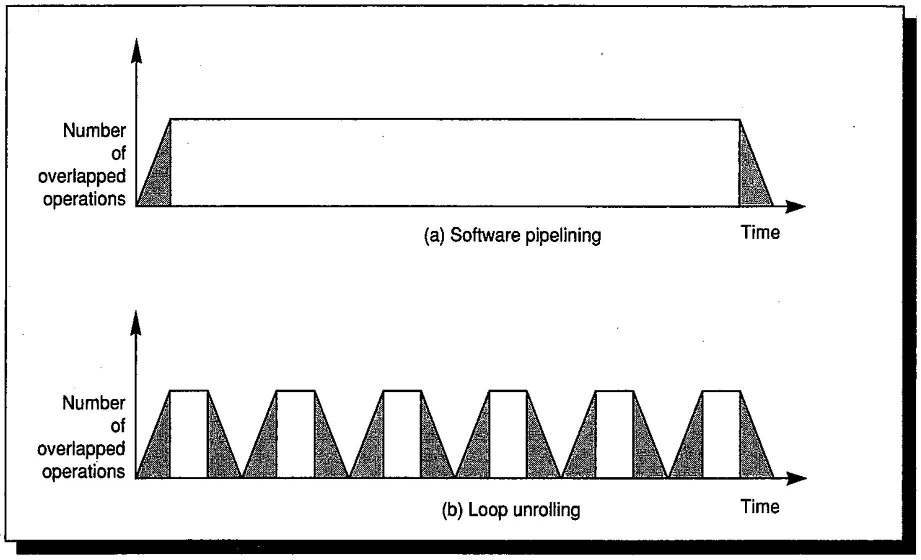

FIGURE 6.50 This shows the execution pattern for (a) a software-pipelined loop and (b) an unrolled loop. The shaded areas are the times when the loop is not running with maximum overlap or parallelism among instructions. This occurs once at loop beginning and once at the end for the software-pipelined loop. For the unrolled loop it occurs $\frac{m}{n}$ times if the loop has a total of *m* executions and is unrolled *n* times. Each block represents an unroll of *n* iterations. Increasing the number of unrolls will reduce the start-up and clean-up overhead.

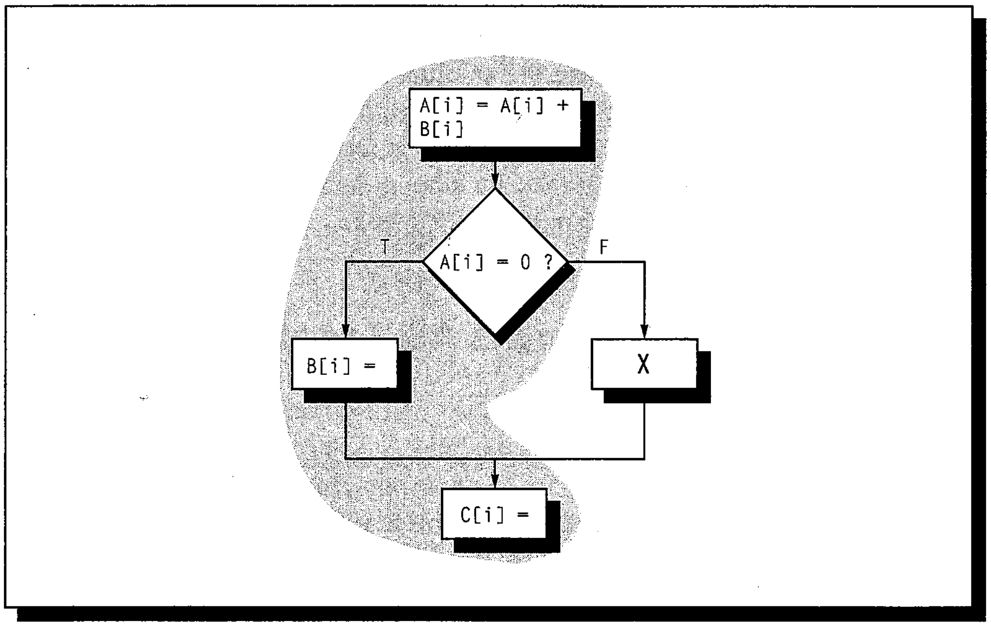

FIGURE 6.51 A code fragment and the trace selected shaded with gray. This trace would be selected first, if the probability of the true branch being taken were much higher than the probability of the false branch being taken. The branch from the decision (A[i]=0) to X is a branch **out** of the trace, and the branch from X to the assignment to C is a branch **into** the trace. These branches are what make compacting the trace difficult.

Once the trace is selected as shown in Figure 6.51, it must be compacted so as to fill the wide instruction word. Compacting the trace involves moving the assignments to variables B and C up to the block before the branch decision. Let's first consider the problem of moving the assignment to B. If the assignment to B is moved above the branch (and thus out of the trace), the code in X would be affected if it used B, since moving the assignment would change the value of B. Thus, to move the assignment to B, B must not be read in X. One could imagine more clever schemes if B were read in X—for example, making a shadow copy and updating B later. Such schemes are generally not used, both because they are complex to implement and because they will slow down the program if the trace selected is not optimal and the operations end up requiring additional instructions. Also, because the assignment to B is moved before the if test, for this schedule to be valid either X also assigns to B or B is not read after the if statement.

Moving the assignment to C up to before the first branch requires first moving it over the branch from X into the trace. To do this, a copy is made of the assignment to C on the branch into the trace. A check must still be done, as was done for B, to make sure that the assignment can be moved over the branch out of the trace. If C is successfully moved to before the first branch and the "false" direction of the branch—the branch off the trace—is taken, the assignment to C will have been done twice. This may be slower than the original code, depending on whether this operation or other moved operations create additional work in the main trace. Ironically, the more successful the trace-scheduling algorithm is in moving code across the branch, the higher the penalty for misprediction.

Loop unrolling, trace scheduling, and software pipelining all aim at trying to increase the amount of local instruction parallelism that can be exploited by a machine issuing more than one instruction on every clock cycle. The effectiveness of each of these techniques and their suitability for various architectural approaches are among the most significant open research areas in pipelined-processor design.

6.9 Putting It All Together: A Pipelined VAX

In this section we will examine the pipeline of the VAX 8600, a macropipelined VAX. This machine is described in detail by DeRosa et al. [1985] and Troiani et al. [1985]. The 8600 pipeline is a more dynamic structure than the DLX integer pipeline. This is because the processing steps may take multiple cycles in one stage of the pipeline. Additionally, the hazard detection is more complicated because of the possibility that stages progress independently and because instructions may modify registers before they complete. Techniques similar to those used in the DLX FP pipeline to handle variable-length instructions are used in the 8600 pipeline.

The 8600 is macropipelined—the pipeline understands the structure of VAX instructions and overlaps their execution, checking the hazards on the instruction operands. By comparison, the VAX 8800 is micropipelined—microinstructions are overlapped and hazard detection occurs in the microprogram unit. A different issue of the Digital Technical Journal [Digital 1987] describes this machine, and Clark [1987] describes the pipeline and its performance. The designs are interesting to compare.

Figure 6.52 shows the 8600 partitioned into four major structural components. The MBox is responsible for address translation and memory access (see Chapter 8). The IBox is the heart of the 8600 pipeline; it is responsible for instruction fetch and decode, operand address calculation, and operand fetch. The EBox and FBox are responsible for execution of integer and floating-point operations, and their primary function is to implement the opcode portion of an instruction. (Because the FBox is optional, the EBox also contains microcode to do the floating point, albeit at much lower performance. The optional presence of the FBox further complicates the operand processing in the EBox.) Since the EBox and FBox are not pipelined, we will focus our attention primarily on the IBox. In explaining the IBox function we will refer to the EBox occasionally; usually the same comments apply to the FBox.

Figure 6.53 breaks the execution of a VAX instruction into four overlapped steps. The number of clock cycles per step may vary widely, though each step in the pipeline takes at least one clock.

A VAX instruction may take many clock cycles in a given step. For example, with multiple memory operands, the instruction will take multiple clock cycles in the Opfetch step. Because of this, an instruction that takes many cycles at a

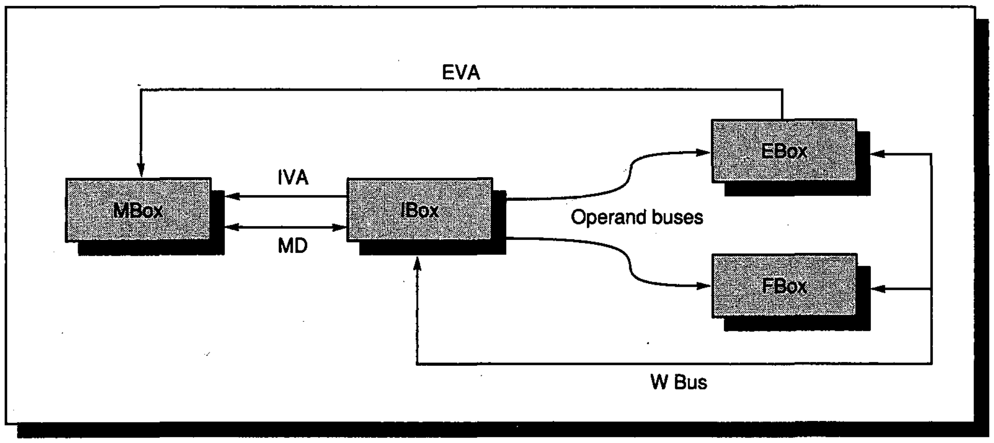

FIGURE 6.52 The basic structure of the 8600 consists of an MBox (responsible for memory access), IBox (handles instruction and operand processing), EBox (all opcode interpretation except floating point), and FBox (performs floating-point operations). These four units are connected by six major buses. The IVA and EVA carry the address for a memory access to the MBox from the IBox and EBox. The MD bus carries memory data to or from the MBox; all such data flows through the IBox. The EBox initiates memory access directly with the MBox only under unusual conditions (e.g., misaligned references). The operand buses carry operands from the IBox (where they are fetched from memory or registers) to the EBox and FBox. Finally, the W Bus carries results to be written from the EBox and FBox to the GPRs and to memory, via the IBox.

Step	Function	Located in
1. Ifetch	Prefetch instruction bytes and decode them	IBox
2. Opfetch	Operand address calculation and fetch	IBox
3. Execution	Execute opcode and write result	EBox, FBox
4. Result store	Write result to memory or registers	EBox, IBox

FIGURE 6.53 The basic structure of the 8600 pipeline has four stages, each taking from 1 to a large number of clock cycles. Up to four VAX instructions are being processed at once.

stage may cause a back up in the pipeline; this back up may eventually reach the Ifetch step, where it will cause the pipeline to simply stop fetching instructions. Additionally, several resources (e.g., the W Bus and GPR ports) are contended for by multiple stages in the pipeline. In general, these problems are resolved on the fly using a fixed-priority scheme.

Operand Decode and Fetch

Much of the work in interpreting a VAX instruction is in the operand specifier and decode process, and this is the heart of the IBox. Substantial effort is devoted to decoding and fetching operands as fast as possible to keep instructions flowing through the pipeline. Figure 6.54 shows the number of cycles spent in Opfetch under ideal conditions (no cache misses or other stalls from the memory hierarchy) for each operand specifier. If the result is a register, the EBox stores

Specifier	Cycles
Literal or immediate	1
Register	1
Deferred	1
Displacement	1
PC-relative and absolute	1
Autodecrement	1
Autoincrement	2
Autoincrement deferred	5
Displacement deferred	4
PC-relative deferred	4

FIGURE 6.54 The minimum number of cycles spent in Opfetch by operand specifier. This shows the data for an operand of type byte, word, or longword that is read. Modified and written operands take an additional cycle, except for register mode and immediate or literal, where writes are not allowed. Quadword and octaword operands may take much longer. If any stalls are encountered, the cycle count will increase.

the result. If the result is a memory operand, Opfetch calculates the address and waits for the EBox to signal ready, then the IBox stores the result during the Result store step. If an instruction result is to be stored in memory, the EBox signals to the IBox when it enters the last cycle of execution for the instruction. This allows Opfetch to overlap the first cycle of a two-cycle memory write with the last cycle of execution (even if the operation only takes one cycle).

To maximize the performance of the machine, there are three copies of the GPRs—in the IBox, EBox, and FBox. A write is broadcast from the FBox, EBox, or IBox (in the case of autoincrement or autodecrement addressing) to the other two units, so that their copies of the registers can be updated.

Handling Data Dependences

Register hazards are tracked in Opfetch by maintaining a small table of registers that will be written. Whenever an instruction passes through Opfetch, its result register is marked as busy. If an instruction that uses that register arrives in Opfetch and sees the busy flag set, it stalls until the flag is cleared. This prevents RAW hazards. The busy flag is cleared when the register is written. Because there are only two stages after Opfetch (execute and write memory result), the busy flag can be implemented as a two-entry associative memory. Writes are maintained in order and always at the end of the pipeline, and all reads are done in Opfetch. This eliminates all explicit WAW and WAR hazards. The only possible remaining hazards are those that can occur on implicit operands, such as the registers written by a `MOVC3`. Hazards on implicit operands are prevented by explicit control in the microcode.

Opfetch optimizes the case when the last operand specifier is a register by processing the register operand specifier at the same time as the next-to-last specifier. In addition, when the result register of an instruction is the source operand of the next instruction, rather than stall the dependent instruction, Opfetch merely signals this relationship to the EBox, allowing execution to proceed without a stall. This is like the bypassing in our DLX pipeline.

Memory hazards between reads and writes are easily resolved because there is a single memory port, and the IBox decodes all operand addresses.

Handling Control Dependences

There are two aspects to handling branches in a VAX: synchronizing on the condition code and dealing with the branch hazard. Most of the branch processing is handled by the IBox. A predict-taken strategy is used; the following steps are taken when the IBox sees a branch:

1. Compute the branch target address, send it to the MBox, and initiate a fetch from the target address. Wait for the EBox to issue CCSYNC, which indicates that the condition codes will be available in the next clock cycle.

2. Evaluate the condition codes from the EBox to check the prediction. If the prediction was incorrect, the access initiated in the MBox is aborted. The current PC points at the next instruction or its first operand specifier.
3. Assuming the branch was taken, the IBox flushes the prefetch and decode stages and begins loading the instruction register and processing the new target stream. If the branch was not taken, the access to the potential target has already been killed and the pipeline can continue just using what is in the prefetch and decode stages.

Simple conditional branches (BEQL, BNEQ), the unconditional branches (BRB, BRW), and the computed branches (e.g., AOBLEQ) are handled by the IBox. The EBox handles more complex branches and also the instructions used for calls and returns.

An Example

To really understand how this pipeline works, let's look at how a code sequence executes. This example is somewhat simplified, but is sufficient to demonstrate the major pipeline interactions. The code sequence we will consider is as follows (remember that for consistency the result of the ADDL3 is given first):

```
         ADDL3    R1,R2,56(R3)
         CMPL     45(R1),@54(R2)
         BEQL     target
         MOVL     ...
target:  SUBL3    ...
```

Figure 6.55 shows an annotated pipeline diagram of how these instructions would progress through the 8600 pipeline.

Dealing with Interrupts

The 8600 maintains three program counters so that instruction interruption and restart are possible. These program counters and what they designate are:

- Current Program Counter—points to the next byte to be processed and consumed in Opfetch.
- IBox Starting Address—points to the instruction currently in Opfetch.
- EBox Starting Address—points to the instruction executing in the EBox or FBox.

In addition, the prefetch unit keeps an address to prefetch from (the VIBA, Virtual Instruction Buffer Address), but this does not affect interrupt handling. When an exception is caused by a prefetch operation, the byte in the instruction buffer is marked. When Opfetch eventually asks for the byte, it will see the exception, and the Current Program Counter will have the address of the byte that caused the exception.

	Clock Cycle								
Instr.	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
`ADDL3`	**IF**: Fetch `ADDL.`	**IF**: Continue prefetch if space and MBox available.	**IF**: Decode `R1.`	**IF**: Decode `R2.` **OP**: Fetch `R1.`	**IF**: Decode `56(R3).` **OP**: Fetch `R2.`	**OP**: Compute `56+(R3).` **EX**: get first operand.	**OP**: Start write. **EX**: Add.	**WR**: Store.	
`CMPL`						**IF**: Decode `45(R1).`		**IF**: Decode `@54(R2).` **OP**: Fetch `45(R1).`	**OP**: Fetch `54(R2).`
`BEQL`									**IF**: Decode `BEQL` displace.
`SUBL`									

	Clock Cycle								
Instr.	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**
`ADDL3`									
`CMPL`	**OP**: stall. **EX**: get first operand.	**OP**: get indirect address.	**OP**: Fetch `@54(R2).`		**EX**: compare and set CC.				
`BEQL`				**OP**: Load VA.	**OP**: Fetch branch target.	**OP**: Fetch target +4; load VIBA; flush IBuffer.			
`SUBL`							**IF**: Decode `SUBL3.`	**OP**: Fetch first operand.	**OP**: Fetch second operand.

FIGURE 6.55 The VAX 8600 executing a code sequence. The top portion shows the events on clock ticks 1–9, while the bottom portion shows the events on clock ticks 10–18. The pipeline stages are abbreviated as IF (Instruction Fetch), OP (Opfetch), EX (Execution), and WR (Write Result) and are shown in bold. Each instruction passes through the 8600 pipeline as soon as the pipe stage is empty and the required data is available. Note that an instruction can be in both the IF and OP stages at the same time. This figure assumes that at the beginning of cycle 1, the prefetch buffer is empty. The prefetch in the IF stage continues to fetch instructions as long as there is room in the prefetch buffer and an available MBox cycle. It is omitted from the diagram for simplicity. The action "stall" indicates a stall for a memory operand during Opfetch. In total, the three VAX instructions executed take 15 cycles, assuming no stalls from the memory system. This sequence was chosen to demonstrate the functioning of the pipeline—it is not necessarily typical.

These PCs are updated when an instruction enters the corresponding pipeline stage. Hence, if an interrupt occurs in a given stage, the PC can be set back to the beginning of that instruction. These PCs are needed because the length of VAX instructions is variable and can only be determined by finding the opcode byte.

In addition to restoring the starting address of the instruction that caused the interrupt, we must unwind any register updates done by addressing modes processed in Opfetch for instructions that are after the instruction that interrupts the processor. The IBox maintains a log of updates to the register file done on behalf of multiple instructions, as we did in Section 5.6. The effects of any changes are undone and the PC is restored. This allows the operating system to have a clean machine state to work from.

Final Remarks

The 8600 uses a four-step pipeline. The theoretical peak performance with the 80-ns clock is 12.5 million VAX instructions per second. Some simple sequences of instructions can actually attain this peak performance with a CPI of 1. Typically, the performance on integer code is about 1.75 million VAX instructions per second for a CPI of about 7. This yields about 3.5 times the performance of a VAX-11/780.

6.10 Fallacies and Pitfalls

Fallacy: Instruction set design has little impact on pipelining.

This is perhaps the most prominent misconception about pipelining and one that was widely held until recently. Many of the difficulties of pipelining arise because of instruction set complications. Here are some examples, many of which are mentioned in the chapter:

- Variable instruction lengths and running times can lead to imbalance among pipeline stages causing other stages to back up. They also severely complicate hazard detection and the maintenance of precise interrupts. Of course, there are exceptions to every rule. For example, caches cause instruction running times to vary when they miss; however, the performance advantages of caches make the added complexity acceptable. To minimize the complexity, most machines freeze the pipeline on a cache miss. Other machines try to continue running parts of the pipeline; though this is very complex, it may overcome some of the performance losses from cache misses.
- Sophisticated addressing modes can lead to different sorts of problems. Addressing modes that update registers, such as post autoincrement, complicate

hazard detection. They also slightly increase the complexity of instruction restart. Other addressing modes that require multiple memory accesses substantially complicate pipeline control and make it difficult to keep the pipeline flowing smoothly.

- Architectures that allow writes into the instruction space (self-modifying code) can cause trouble for pipelining (as well as for cache designs). For example, if an instruction in the pipeline can modify another instruction, we must constantly check if the address being written to by an instruction corresponds to the address of an instruction further on in the pipeline. If so, the pipeline must be flushed or the instruction in the pipeline somehow updated.
- Implicitly set condition codes increase the difficulty of finding when a branch has been decided and the difficulty of scheduling branch delays. The former problem occurs when the condition-code setting is not uniform, making it difficult to decide which instruction sets the condition code last. The latter problem occurs when the setting of the condition code is not under program control. This makes it hard to find instructions that can be scheduled between the condition evaluation and the branch. Many newer architectures avoid condition codes or set them explicitly under program control to eliminate the pipelining difficulties.

As a simple example, suppose the DLX instruction format were more complex, so that a separate, decode pipe stage were required before register fetch. This would increase the branch delay to two clock cycles. At best, the second branch-delay slot would be wasted at least as often as the first. Gross [1983] found that a second delay slot was only used half as often as the first. This would lead to a performance penalty for the second delay slot of more than 0.1 clock cycles per instruction.

Pitfall: Unexpected execution sequences may cause unexpected hazards.

At first glance, WAW hazards look like they should never occur because no compiler would ever generate two writes to the same register without an intervening read. But they can occur when the sequence was unexpected. For example, the first write might be in the delay slot of a taken branch when the scheduler thought the branch would not be taken. Here is the code sequence that could cause this:

```
        BNEZ    R1,foo
        DIVD    F0,F2,F4    ; moved into delay slot
                            ; from fall through
        .....
        .....
foo:    LD      F0,qrs
```

If the branch is taken, then before the DIVD can complete the LD will reach WB, causing a WAW hazard. The hardware must detect this and may stall the issue of the LD. Another way this can happen is if the second write is in a trap routine. This occurs when an instruction that traps and is writing results continues and completes after an instruction that writes the same register in the trap handler. The hardware must detect and prevent this as well.

Fallacy: Increasing the depth of pipelining always increases performance.

Two factors combine to limit the performance improvement gained by pipelining. Data dependences in the code mean that increasing the pipeline depth will increase the CPI, since a larger percentage of the cycles will become stalls. Second, clock skew and latch overhead combine to limit the decrease in clock period obtained by further pipelining. Figure 6.56 shows the tradeoff between pipeline depth and performance for the first 14 of the Livermore Loops (see Chapter 2, page 43). The performance flattens out when the pipeline depth reaches 4 and actually drops when the execution portion is pipelined 16 deep.

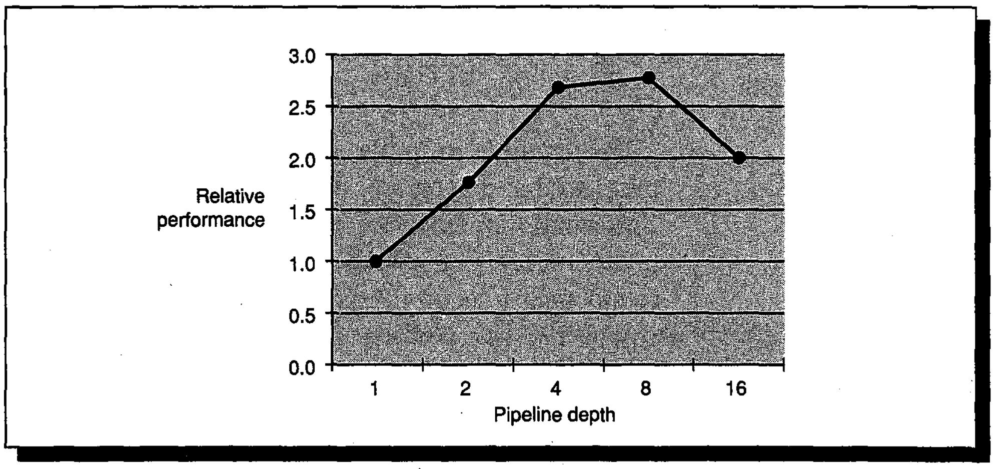

FIGURE 6.56 The depth of pipelining versus the speedup obtained. This data is based on Table 2 in Kunkel and Smith [1986]. The x axis shows the number of stages in the EX portion of the floating-point pipeline. A single-stage pipeline corresponds to 32 levels of logic, which might be appropriate for a single FP operation.

Pitfall: Evaluating a scheduler on the basis of unoptimized code.

Unoptimized code—containing redundant loads, stores, and other operations that might be eliminated by an optimizer—is much easier to schedule than "tight" optimized code. In GCC running on a DECstation 3100, the frequency of idle clock cycles increases by 18% from the unoptimized and scheduled code to the optimized and scheduled code. TeX shows a 20% increase for the same measurement. To fairly evaluate a scheduler you must use optimized code, since in the real system you will derive a good performance from other optimizations in addition to scheduling.

Pitfall: Extensive pipelining can impact other aspects of a design, leading to overall lower cost/performance.

The best example of this phenomenon comes from two implementations of the VAX, the 8600 and the 8700. We discussed the instruction pipeline of the 8600 in Section 6.9. When the 8600 was initially delivered, it had a cycle time of 80 ns. Subsequently, a redesigned version, called the 8650, with a 55-ns clock was introduced. The 8700 has a much simpler pipeline that operates at the microinstruction level. The 8700 CPU is much smaller and has a faster clock rate, 45 ns. The overall outcome is that the 8650 has a CPI advantage of about 20%, but the 8700 has a clock rate that is about 20% faster. Thus, the 8700 achieves the same performance with much less hardware.

6.11 Concluding Remarks

Figure 6.57 shows how the various pipelining approaches affect both clock speed and CPI. This figure does not account for instruction-count differences. Since performance is clock speed divided by CPI (ignoring instruction-count differences), machines in the top left corner will be slowest, and machines in the bottom right corner will be fastest. However, the machines that move towards the lower right corner will probably achieve their maximum performance on the narrowest range of applications.

Machines that are *underpipelined* lump multiple DLX pipestages into one. The clock cannot be run as fast, and the CPI will be only marginally lower. The DLX pipeline achieves a CPI very close to 1 (ignoring memory-system stalls) at a reasonable clock speed. Architectural simplicity and efficient pipelining are two of the most important attributes of the RISC (Reduced Instruction Set Computer) machines. DLX constitutes an example of such a machine. We have chosen to use the term load/store architecture because the ideas apply to a broad range of machines, and not just to the machines that identify themselves as RISCs. Much of the discussion in the first part of this chapter centered around the key ideas developed by the RISC projects.

Machines with higher clock rates and deeper pipelines have been called *superpipelined.* Superpipelined machines are characterized by pipelining all functional units. A superpipelined version of DLX might have a 10-stage pipeline, rather than the 5-stage pipeline described earlier. Other than increasing the complexity of pipeline scheduling and pipeline control, superpipelined machines are not fundamentally different from the machines we have already examined in this chapter. Due to limited instruction-level parallelism, a superpipelined machine will have a slightly higher CPI than a DLX-style pipeline, but its advantage in clock cycle time should be larger than the disadvantage in CPI.

Superscalar processors can have clock cycle times very close to that of a DLX pipeline and maintain a smaller CPI. The VLIW machines can have a substantially lower CPI, but tend to have a significantly higher clock cycle time for the reasons discussed in this chapter. The vector machines effectively use both techniques. They are usually superpipelined and have powerful vector operations that can be considered equivalent to issuing multiple independent operations on a machine like DLX. We will explore vector machines in detail in the next chapter.

Going out from the top left corner on either axis in Figure 6.57, the requirement to exploit more instruction-level parallelism increases; at the same time, of course, fewer programs will run at maximum speed.

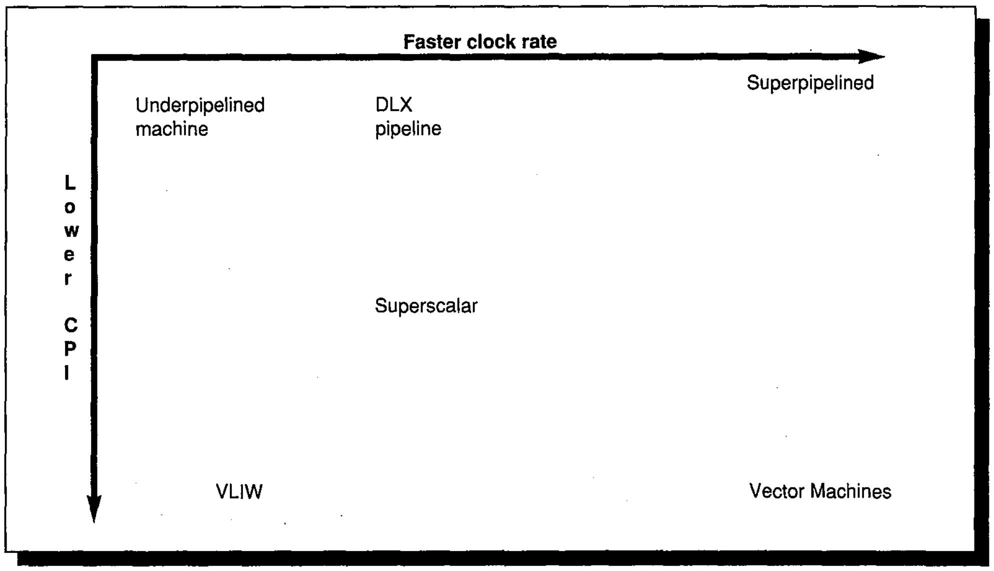

FIGURE 6.57 Increasing the instruction-issue rate lowers the CPI, while a deeper pipeline increases the clock rate. Various machines combine these techniques.

6.12 Historical Perspective and References

This section describes some of the major advances in pipelining and ends with some of the recent literature on high-performance pipelining.

The first general-purpose pipelined machine is considered to be Stretch, the IBM 7030. Stretch followed on the IBM 704 and had a goal of being 100 times faster than the 704. The goals were a stretch from the state of the art at that time—hence the nickname. The plan was to obtain a factor of 1.6 from overlapping fetch, decode, and execute, using a 4-stage pipeline. Bloch [1959] and Bucholtz [1962] describe the design and engineering tradeoffs, including the use of ALU bypasses.

In 1964 CDC delivered the first CDC 6600. The CDC 6600 was unique in many ways. In addition to introducing scoreboarding, the CDC 6600 was the first machine to make extensive use of multiple functional units. It also had peripheral processors that used a timeshared pipeline. The interaction between pipelining and instruction set design was understood, and the instruction set was kept simple to promote pipelining. The CDC 6600 also used an advanced packaging technology. Thornton [1964] describes the pipeline and I/O processor architecture, including the concept of out-of-order instruction execution. Thornton's book [1970] provides an excellent description of the entire machine, from technology to architecture, and includes a foreword by Cray. (Unfortunately, this book is currently out of print.) The CDC 6600 also has an instruction scheduler for the FORTRAN compilers, described by Thorlin [1967].

The IBM 360/91 introduced many new concepts, including tagging of data, register renaming, dynamic detection of memory hazards, and generalized forwarding. Tomasulo's algorithm is described in his 1967 paper. Anderson, Sparacio, and Tomasulo [1967] describe other aspects of the machine, including the use of branch prediction. Patt and his colleagues have described an approach, called HPSm, that is an extension of Tomasulo's algorithm [Hwu and Patt 1986].

A series of general pipelining descriptions that appeared in the late 1970s and early 1980s provided most of the terminology and described most of the basic techniques used in simple pipelines. These surveys include Keller [1975], Ramamoorthy and Li [1977], Chen [1980], and Kogge's book [1981], devoted entirely to pipelining. Davidson and his colleagues [1971, 1975] developed the concept of pipeline reservation tables as a design methodology for multicycle pipelines with feedback (also described in Kogge [1981]). Many designers use a variation of these concepts, as we did in Figures 6.3 and 6.4.

The RISC machines refined the notion of compiler-scheduled pipelines in the early 1980s. The concepts of delayed branches and delayed loads—common in microprogramming—were extended into the high-level architecture. The Stanford MIPS architecture made the pipeline structure purposely visible to the compiler and allowed multiple operations per instruction. Schemes for scheduling the pipeline in the compiler were described by Sites [1979] for the Cray, by Hennessy and Gross [1983], (and in Gross's thesis [1983]) and by Gibbons and Muchnik [1986]. Rymarczyk [1982] describes the interlock conditions that programmers should be aware of for a 360-like machine; this paper also shows the complex interaction between pipelining and an instruction set not designed to be pipelined.

J. E. Smith and his colleagues have written a number of papers examining instruction issue, interrupt handling, and pipeline depth for high-speed scalar machines. Kunkel and Smith [1986] evaluate the impact of pipeline overhead and dependences on the choice of optimal pipeline depth; they also have an excellent discussion of latch design and its impact on pipelining. Smith and Plezkun [1988] evaluate a variety of techniques for preserving precise interrupts, including the future file concept mentioned in Section 6.6. Weiss and Smith [1984] evaluate a variety of hardware pipeline scheduling and instruction-issue techniques.

Dynamic hardware branch-prediction schemes are described by J. E. Smith [1981] and by A. Smith and Lee [1984]. Ditzel [1987] describes a novel branch-target buffer for CRISP. McFarling and Hennessy [1986] is a quantitative comparison of a variety of compile-time and run-time branch-prediction schemes.

A series of early papers, including Tjaden and Flynn [1970] and Foster and Riseman [1972], concluded that only small amounts of parallelism could be available at the instruction level without investing an enormous amount of hardware. These papers dampened the appeal of multiple instruction issue for more than ten years. Nicolau and Fisher [1984] published a paper asserting the presence of large amounts of potential instruction-level parallelism.

Charlesworth [1981] reports on the Floating Point Systems AP-120B, one of the first wide-instruction machines containing multiple operations per instruction. Floating Point Systems applied the concept of software pipelining—albeit by hand, rather than with a compiler—by writing assembly language libraries to use the machine efficiently. Weiss and J. E. Smith [1987] compare software pipelining versus loop unrolling as techniques for scheduling code on a pipelined machine. Lam [1988] presents algorithms for software pipelining and evaluates their use on Warp, a wide-instruction-word machine. Along with his colleagues at Yale, Fisher [1983] proposed creating a machine with a very wide instruction (512 bits), and named this type of machine a VLIW. Code was generated for the machine using trace scheduling, which Fisher [1981] had developed originally for generating horizontal microcode. The implementation of trace scheduling for the Yale machine is described by Fisher, et. al. [1984] and by Ellis [1986]. The Multiflow machine (see Colwell et. al. [1987]) commercialized the concepts developed at Yale.

Several researchers proposed techniques for multiple instruction issue. Agerwala and Cocke [1987] proposed this approach as an extension of the RISC ideas, and coined the name "superscalar." IBM described a machine based on these ideas in late 1989 (see Bakoglu et al. [1989]). In 1990, the IBM was announced as the RS/6000. The implementation can issue up to four instructions per clock. A good description of the machine, its background, and software appears in IBM [1990]. The Apollo DN 10000 and the Intel i860 both offer multiple instruction issue, though the requirements for multiple issue are rather rigid. The Intel i860 should probably be considered a LIW machine because the program must explicitly indicate whether instruction pairs should be dual issued. Although the pairs are ordinary instructions, there are substantial limitations on what can appear as a member of a dual-issued pair. The Intel 960CA and Tandem Cyclone are examples of superscalar machines with complex instruction sets.

J. E. Smith and his colleagues at Wisconsin [1984] proposed the decoupled approach that included multiple issue with dynamic pipeline scheduling. The Astronautics ZS-1 described by Smith et al. [1987] embodies this approach and uses queues to connect the load/store unit and the operation units. J. E. Smith [1989] also describes the advantages of dynamic scheduling and compares that approach to static scheduling. Dehnert, Hsu, and Bratt [1989] explain the architecture and performance of the Cydrome Cydra 5, a machine with a wide instruction word that provides dynamic register renaming. The Cydra 5 is a unique blend of hardware and software aimed at extracting instruction-level parallelism.

Recently there have been a number of papers exploring the tradeoffs among alternative pipelining approaches. Jouppi and Wall [1989] examine the performance differences between superpipelined and superscalar systems, concluding that their performance is similar, but that superpipelined machines may require less hardware to achieve the same performance. Sohi and Vajapeyam [1989] give measurements of available parallelism for wide-instruction-word machines. Smith, Johnson, and Horowitz [1989] recount studies of available instruction-level parallelism in nonscientific code using an ambitious hardware scheme that allows multiple-instruction execution.

References

AGERWALA, T. AND J. COCKE [1987]. "High performance reduced instruction set processors," IBM Tech. Rep. (March).

ANDERSON, D. W., F. J. SPARACIO, AND R. M. TOMASULO [1967]. "The IBM 360 Model 91: Machine philosophy and instruction handling," *IBM J. of Research and Development* 11:1 (January) 8–24.

BAKOGLU, H. B., G. F. GROHOSKI, L. E. THATCHER, J. A. KAHLE, C. R. MOORE, D. P. TUTTLE, W. E. MAULE, W. R. HARDELL, D. A. HICKS, M. NGUYEN PHU, R. K. MONTOYE, W. T. GLOVER , AND S. DHAWAN [1989]. "IBM second-generation RISC machine organization," Proc. Int'l Conf. on Computer Design, IEEE (October) Rye, N.Y., 138–142.

BLOCH, E. [1959]. "The engineering design of the Stretch computer," *Proc. Fall Joint Computer Conf.*, 48–59.

BUCHOLTZ, W. [1962]. *Planning a Computer System: Project Stretch,* McGraw-Hill, New York.

CHARLESWORTH, A. E. [1981]. "An approach to scientific array processing: The architecture design of the AP-120B/FPS-164 family," *Computer* 14:12 (December) 12–30.

CHEN, T. C. [1980]. "Overlap and parallel processing" in *Introduction to Computer Architecture,* H. Stone, ed., Science Research Associates, Chicago, 427–486.

CLARK, D. W. [1987]. "Pipelining and performance in the VAX 8800 processor," *Proc. Second Conf. on Architectural Support for Programming Languages and Operating Systems,* IEEE/ACM (March), Palo Alto, Calif., 173–177.

COLWELL, R. P., R. P. NIX, J. J. O'DONNELL, D. B. PAPWORTH, AND B. K. RODMAN [1987]. "A VLIW architecture for a trace scheduling compiler," *Proc. Second Conf. on Architectural Support for Programming Languages and Operating Systems,* IEEE/ACM (March), Palo Alto, Calif., 180–192.

DAVIDSON, E. S. [1971]. "The design and control of pipelined function generators," *Proc. Conf. on Systems, Networks, and Computers,* IEEE (January), Oaxtepec, Mexico, 19–21.

DAVIDSON, E. S., A. T. THOMAS, L. E. SHAR, AND J. H. PATEL [1975]. "Effective control for pipelined processors," *COMPCON, IEEE* (March), San Francisco, 181–184.

DEHNERT, J. C., P. Y.-T. HSU, AND J. P. BRATT [1989]. "Overlapped loop support on the Cydra 5," *Proc. Third Conf. on Architectural Support for Programming Languages and Operating Systems* (April), IEEE/ACM, Boston, 26–39.

DEROSA, J., R. GLACKEMEYER, AND T. KNIGHT [1985]. "Design and implementation of the VAX 8600 pipeline," *Computer* 18:5 (May) 38–48.

DIGITAL EQUIPMENT CORPORATION [1987]. *Digital Technical J.* 4 (March), Hudson, Mass. (This entire issue is devoted to the VAX 8800 processor.)

DITZEL, D. R. AND H. R. MCLELLAN [1987]. "Branch folding in the CRISP microprocessor: Reducing the branch delay to zero," *Proc. 14th Symposium on Computer Architecture* (June), Pittsburgh, 2–7.

EARLE, J. G. [1965]. "Latched carry-save adder," *IBM Technical Disclosure Bull.* 7 (March) 909–910.

ELLIS, J. R., [1986]. *Bulldog: A Compiler for VLIW Architectures*, The MIT Press,1986.

EMER, J. S. AND D. W CLARK [1984]. "A characterization of processor performance in the VAX-11/780," *Proc. 11th Symposium on Computer Architecture* (June), Ann Arbor, Mich., 301–310.

FISHER, J. A. [1981]. "Trace Scheduling: A Technique for Global Microcode Compaction," *IEEE Trans. on Computers* 30:7 (July), 478-490.

FISHER, J. A. [1983]. "Very long instruction word architectures and ELI-512," *Proc. Tenth Symposium on Computer Architecture* (June), Stockholm, Sweden., 140-150.

FISHER J. A., J. R. ELLIS, J. C. RUTTENBERG, AND A. NICOLAU [1984]. "Parallel processing: A smart compiler and a dumb machine," *Proc. SIGPLAN Conf. on Compiler Construction* (June), Palo Alto, CA, 11-16.

FOSTER, C. C. AND E. M. RISEMAN [1972]. "Percolation of code to enhance parallel dispatching and execution," *IEEE Trans. on Computers* C-21:12 (December) 1411–1415.

GIBBONS, P. B. AND S. S. MUCHNIK [1986]. "Efficient Instruction Scheduling for a Pipelined Processor," *SIGPLAN '86 Symposium on Compiler Construction, ACM* (June), Palo Alto, CA, 11-16.

GROSS, T. R. [1983]. *Code Optimization of Pipeline Constraints*, Ph.D. Thesis (December), Computer Systems Lab., Stanford Univ.

HENNESSY, J. L. AND T. R. GROSS [1983]. "Postpass code optimization of pipeline constraints," *ACM Trans. on Programming Languages and Systems* 5:3 (July) 422-448

HWU, W.-M. AND Y. PATT [1986]. "HPSm, a high performance restricted data flow architecture having minimum functionality," *Proc. 13th Symposium on Computer Architecture* (June), Tokyo, 297–307.

IBM [1990]. "The IBM RISC System/6000 processor," collection of papers, *IBM Jour. of Research and Development* 34:1, (January), 119 pages.

JOUPPI N. P. AND D. W. WALL [1989]. "Available instruction-level parallelism for superscalar and superpipelined machines," *Proc. Third Conf. on Architectural Support for Programming Languages and Operating Systems*, IEEE/ACM (April), Boston, 272–282.

KELLER R. M. [1975]. "Look-ahead processors," *ACM Computing Surveys* 7:4 (December) 177–195.

KOGGE, P. M. [1981]. *The Architecture of Pipelined Computers*, McGraw-Hill, New York.

KUNKEL, S. R. AND J. E. SMITH [1986]. "Optimal pipelining in supercomputers," *Proc. 13th Symposium on Computer Architecture* (June), Tokyo, 404–414.

LAM, M. [1988]. "Software pipelining: An effective scheduling technique for VLIW machines," *SIGPLAN Conf. on Programming Language Design and Implementation*, ACM (June), Atlanta, Ga., 318–328.

MCFARLING, S. AND J. HENNESSY [1986]. "Reducing the cost of branches," *Proc. 13th Symposium on Computer Architecture* (June), Tokyo, 396–403.

NICOLAU, A. AND J. A. FISHER [1984]. "Measuring the parallelism available for very long instruction work architectures," *IEEE Trans. on Computers* C-33:11 (November) 968–976.

RAMAMOORTHY, C. V. AND H. F. LI [1977]. "Pipeline architecture," *ACM Computing Surveys* 9:1 (March) 61–102.

RYMARCZYK, J. [1982]. "Coding guidelines for pipelined processors," *Proc. Symposium on Architectural Support for Programming Languages and Operating Systems,* IEEE/ACM (March), Palo Alto, Calif., 12–19.

SITES, R. [1979]. *Instruction Ordering for the CRAY-1 Computer,* Tech. Rep. 78-CS-023 (July), Dept. of Computer Science, Univ. of Calif., San Diego.

SMITH, A. AND J. LEE [1984]. "Branch prediction strategies and branch target buffer design," *Computer* 17:1 (January) 6–22.

SMITH, J. E. [1981]. "A study of branch prediction strategies," *Proc. Eighth Symposium on Computer Architecture* (May), Minneapolis, 135–148.

SMITH, J. E. [1984]. "Decoupled access/execute computer architectures," *ACM Trans. on Computer Systems* 2:4 (November), 289–308.

SMITH, J. E. [1989]. "Dynamic instruction scheduling and the Astronautics ZS-1," *Computer* 22:7 (July) 21–35.

SMITH, J. E. AND A. R. PLEZKUN [1988]. "Implementing precise interrupts in pipelined processors," *IEEE Trans. on Computers* 37:5 (May) 562–573.

SMITH, J. E., G. E. DERMER, B. D. VANDERWARN, S. D. KLINGER, C. M. ROZEWSKI, D. L. FOWLER, K. R. SCIDMORE, J. P. LAUDON [1987]. "The ZS-1 central processor," *Proc. Second Conf. on Architectural Support for Programming Languages and Operating Systems,* IEEE/ACM (March), Palo Alto, Calif., 199–204.

SMITH, M. D., M. JOHNSON, AND M. A. HOROWITZ [1989]. "Limits on multiple instruction issue," *Proc. Third Conf. on Architectural Support for Programming Languages and Operating Systems,* IEEE/ACM (April), Boston, Mass., 290–302.

SOHI , G. S., AND S. VAJAPEYAM [1989]. "Tradeoffs in instruction format design for horizontal architectures," *Proc. Third Conf. on Architectural Support for Programming Languages and Operating Systems,* IEEE/ACM (April), Boston, Mass. 15–25.

THORLIN, J. F. [1967]. "Code generation for PIE (parallel instruction execution) computers," *Spring Joint Computer Conf.* (April), Atlantic City, N.J.

THORNTON, J. E. [1964]. "Parallel operation in the Control Data 6600," *Proc. Fall Joint Computer Conf.* 26, 33–40.

THORNTON, J. E. [1970]. *Design of a Computer, the Control Data 6600,* Scott, Foresman, Glenview, Ill.

TJADEN, G. S. AND M. J. FLYNN [1970]. "Detection and parallel execution of independent instructions," *IEEE Trans. on Computers* C-19:10 (October) 889–895.

TOMASULO, R. M. [1967]. "An efficient algorithm for exploiting multiple arithmetic units," *IBM J. of Research and Development* 11:1 (January) 25–33.

TROIANI, M., S. S. CHING, N. N. QUAYNOR, J. E. BLOEM, AND F. C. COLON OSORIO [1985]. "The VAX 8600 I Box, a pipelined implementation of the VAX architecture," *Digital Technical J.* 1 (August) 4–19.

WEISS, S. AND J. E. SMITH [1984]. "Instruction issue logic for pipelined supercomputers," *Proc. 11th Symposium on Computer Architecture* (June), Ann Arbor, Mich., 110–118.

WEISS, S. AND J. E. SMITH [1987]. "A study of scalar compilation techniques for pipelined supercomputers," *Proc. Second Conf. on Architectural Support for Programming Languages and Operating Systems* (March), IEEE/ACM, Palo Alto, Calif., 105–109.

EXERCISES

6.1 [12/12/15/20/15/15] <6.2–6.4> Consider an architecture with two instruction formats: a register–register format and a register–memory format. There is a single memory addressing mode (offset + base register).

There is a set of ALU operations with format:

ALUop Rdest, Rsrc_1, Rsrc_2

or

ALUop Rdest, Rsrc_1, MEM

Where the ALUop is one of the following: Add, Subtract, And, Or, Load (Rsrc_1 ignored), Store (Rdest ignored). Rsrc or Rdest are registers. MEM is a base register and offset pair and is a source for any ALUop, except a store instruction where it is the destination.

Branches use a full compare of two registers and are PC-relative. Assume that this machine is pipelined so that a new instruction is started every clock cycle. The following pipeline structure—similar to that used in the VAX 8800 micropipeline—is used:

IF	RF	ALU1	MEM	ALU2	WB					
	IF	RF	ALU1	MEM	ALU2	WB				
		IF	RF	ALU1	MEM	ALU2	WB			
			IF	RF	ALU1	MEM	ALU2	WB		
				IF	RF	ALU1	MEM	ALU2	WB	
					IF	RF	ALU1	MEM	ALU2	WB

The first ALU stage is used for effective address calculation for memory references and branches. The second ALU cycle is used for operations and branch comparison. RF is both a decode and register-fetch cycle. Assume reading in RF and writing in WB occur as in Figure 6.8 (page 262).

a. [12] Find the number of adders needed, counting any adder or incrementer; show a combination of instructions and pipe stages that justify this answer. You need only give one combination that maximizes the adder count.

b. [12] Find the number of register read and write ports and memory read and write ports required. Show that your answer is correct by showing a combination of instructions and pipeline stage indicating the instruction and the number of read ports and write ports required for that instruction.

c. [15] Determine any *data forwarding* between the two separate ALUs used for the ALU1 and ALU2 pipe stages. Put in all forwarding of ALU to ALU needed to avoid or reduce stalls. Show the relationship between the two instructions involved in forwarding.

d. [20] Show any other data-forwarding requirements for the units listed below by giving an example of the source instruction and destination instruction of the forwarding. Each example should show the maximum separation of the two instructions. How many instructions can each example forward across? You need only consider the following units: MDR_{in} (memory data in register), MDR_{out} (memory-data register for outgoing data), ALU_1, and ALU_2. Include any forwarding that is required to prevent or reduce stalls.

e. [15] Give an example of all remaining hazards after all forwarding of parts C and D above has been implemented. What is the maximum number of stalls for each hazard?

f. [15] Show all control hazard types by example and state the length of the stall. The control hazards should be resolved as early as possible (but not using a delayed branch).

6.2 [12] <6.1–6.4> A machine is called "underpipelined" if additional levels of pipelining can be added without changing the pipeline-stall behavior appreciably. Suppose that the DLX pipeline was changed to four stages by merging ID and EX and lengthening the clock cycle by 50%. How much faster would the conventional DLX pipeline be versus the underpipelined DLX on integer code only? Make sure you include the effect of any change in pipeline stalls using the data in Figure 6.24 (page 278).

6.3 [15] <6.2–6.4> We know that a four-deep pipelined implementation has the following hazard frequencies and stall requirements between an instruction *i* and its successors:

$i + 1$ (and not on $i + 2$)	20%	2 cycle stall
$i + 2$	5%	1 cycle stall

Assume that the clock rate of the pipelined machine is four times the clock rate of the nonpipelined implementation. What is the effective performance increase from pipelining if we ignore the effect of hazards? What is the effective performance increase from pipelining if we account for the effect of pipelining hazards?

6.4 [15] <6.3> Suppose the branch frequencies (as percentages of all instructions) are as follows:

Conditional branches	20%
Jumps and calls	5%
Conditional branches	60% are taken

We are examining a four-deep pipeline where the branch is resolved at the end of the second cycle for unconditional branches, and at the end of the third cycle for conditional branches. Assuming that only the first pipe stage can always be done independent of whether the branch goes and ignoring other pipeline stalls, how much faster would the machine be without any branch hazards?

6.5 [20] <6.4> Several designers have proposed the concept of canceling branches (also called squashing or nullifying), as a way to improve the performance of delayed branches. (Several of the machines discussed in Appendix E have this capability.) The idea is to allow the branch to indicate that the instruction in the delay slot should be aborted if the branch is mispredicted. The advantage of canceling branches is that the delay slot can **always** be filled, since the branch can abort the contents of the delay slot if mispredicted. The compiler need not worry about whether the instruction is OK to execute when the branch is mispredicted.

A simple version of canceling branches cancels if the branch is not taken; assume this type of canceling branch. Use the data in Figure 6.18 (page 272) for branch frequency. Assume that 27% of the branch-delay slots are filled using strategy (a) of Figure 6.20 (page 274) with standard delayed branches, and that the rest of the slots are filled using canceling branches and strategy (b). Using the taken/not taken data for Spice from Figure 3.22 on page 107, show the effectiveness of this scheme with canceling branches for Spice using the same format as the graph in Figure 6.22 (page 276). How much faster on Spice would a machine with canceling branches run, assuming there is no clock-speed penalty compared to a machine with only delayed branches? Assume CPI without branch stalls is 1.

6.6 [20/15/20] <6.2–6.4> Suppose that we have the following pipeline layout:

Stage	Function
1	Instruction fetch
2	Operand decode
3	Execution or memory access (branch resolution)

All data dependences are between the register written in Stage 3 of instruction i and a register read in Stage 2 of instruction $i + 1$, before instruction i has completed. The probability of such an interlock occurring is $1/p$.

We are considering a change in the machine organization that would write back the result of an instruction during an effective 4th pipe stage. This would decrease the length of the clock cycle by d (i.e., if the length of the clock cycle was T, it is now T–d). The probability of a dependence between instruction i and instruction i +2 is p^{-2}. (Assume that the value of p^{-1} excludes instructions that would interlock on i +2.) The branch would also be resolved during the fourth stage.

a. [20] Considering only the data hazard, find the lower bound on d that makes this a profitable change. Assume that each result has exactly one use and that the basic clock cycle has length T.

b. [15] Suppose that the probability of an interlock between i and i+n were $0.3 - 0.1n$ for $1 \le n \le 3$. What increase in the clock rate is needed so that this change improves performance?

c. [20] Now assume that we have used forwarding to eliminate the extra hazard introduced by the change. That is, for all *data* hazards the pipeline length is *effectively* 3. This design may still not be worthwhile because of the impact of control hazards coming from a four-stage versus a three-stage pipeline. Assume that only Stage 1 of the pipeline can be safely executed before we decide whether a branch goes or not and that all branches are conditional. We want to know what the impact of branch hazards can be before this longer pipeline does not yield high performance. Find an upper bound on the percent of conditional branches in programs in terms of the ratio of d to the original clock-cycle time, so that the longer pipeline has better performance. What if d is a 10% reduction, what is the maximum percentage of conditional branches, before we lose with this longer pipeline? Assume the taken-branch frequency for conditional branches is 60%.

6.7 [12] <6.7> A shortcoming of the scoreboard approach occurs when multiple functional units that share input buses are waiting for a single result. The units cannot start simultaneously, but must serialize. This is not true in Tomasulo's algorithm. Give a code sequence that uses no more than 10 instructions and shows this problem. Use the FP latencies from Figure 6.29 (page 289) and the same functional units in both examples. Indicate where the Tomasulo approach can continue, but the scoreboard approach must stall.

6.8 [15] <6.7> Tomasulo's algorithm also has a disadvantage versus the scoreboard: only one result can complete per clock, due to the CDB. Using the FP latencies from Figure 6.29 (page 289) and the same functional units in both cases, find a code sequence of no more than 10 instructions where scoreboard does not stall, but Tomasulo's algorithm must. Indicate where this occurs in your sequence.

6.9 [15] <6.7> Suppose we have a deeply pipelined machine, for which we implement a branch-target buffer for the conditional branches only. Assume that the misprediction penalty is always 4 cycles and the buffer miss penalty is always 3 cycles. Assume 90% hit rate and 90% accuracy, and the branch statistics in Figure 6.18 (page 272). How much faster is the machine with the branch-target buffer versus a machine that has a fixed 2-cycle branch penalty? Assume a base CPI without branch stalls of 1.

6.10 [15] <6.7> Some designers have proposed using branch-target buffers to obtain a zero-delay unconditional branch (see Ditzel and McLellan [1987]). The buffer simply caches the target instruction rather than the target PC. On an unconditional branch that hits in the branch-target buffer, the target instruction is fetched and sent to the pipeline in place of the unconditional branch. Assuming a 90% hit rate, a base CPI of 1, and the data in Figure 6.18 (page 272), how much improvement is gained by this enhancement versus a machine whose effective CPI is 1.1.

6.11–6.19 For these problems we will look at how a common vector loop runs on a variety of pipelined versions of DLX. The loop is the so-called SAXPY loop (discussed extensively in Chapter 7). The loop implements the vector operation Y = a*X+Y for a vector of length 100. Here is the DLX code for the loop:

```
foo:  LD  double F2,0(R1) FP  ;load X(i)
      MULTD      F4,F2,F0     ;multiply a*X(i)
      LD         F6,0(R2)     ;load Y(i)
      ADDD       F6,F4,F6     ;add aX(i) + Y(i)
      SD         0(R2),F6 FP  ;store Y(i)
      ADDI immed R1,R1,8      ;increment X index
      ADDI       R2,R2,8      ;increment Y index
      SGTI       R3,R1,done   ;test if done
      BEQZ       R3,foo       ; loop if not done
```

For these problems, assume that the integer operations issue and complete in one clock cycle and that their results are fully bypassed. Ignore the branch delay. You will use the FP latencies shown in Figure 6.29 (page 289) unless stated otherwise. Assume the FP units are not pipelined unless the problem states otherwise.

6.11 [20] <6.2–6.6> For this problem use the pipeline constraints shown in Figure 6.29 (page 289). Show the number of stall cycles for each instruction and what clock cycle the instruction begins execution (i.e., enters its first EX cycle) on the first iteration of the loop. How many clock cycles does each loop iteration take?

6.12 [22] <6.7> Using the DLX code for SAXPY above, show the state of the scoreboard tables (as in Figure 6.32) when the `SGTI` instruction reaches Write result. Assume that issue and read operands each take a cycle. Assume that there are three integer functional units and they take only a single execution cycle (including loads and stores). Assume the functional unit count described in Section 6.7 with the FP latencies of Figure 6.29. The branch should not be included in the scoreboard.

6.13 [22] <6.7> Use the DLX code for SAXPY above and the latencies of Figure 6.29. Assuming Tomasulo's algorithm for the hardware with the functional units described in Section 6.7, show the state of the reservation stations and register-status tables (as in

Figure 6.37) when the SGTI writes its result on the CDB. Make the same assumptions about latencies and functional units as Exercise 6.12.

6.14 [22] <6.7> Using the DLX code for SAXPY above, assume a scoreboard with the functional units described in the algorithm for the hardware, plus three integer functional units (also used for load/store). Assume the following latencies in clock cycles:

FP multiply	10
FP add	6
FP load/store	2
All integer operations	1

Show the state of the scoreboard (as in Figure 6.32) when the branch issues for the second time. Assume the branch was correctly predicted taken and took one cycle. How many clock cycles does each loop iteration take? You may ignore any register port/bus conflicts.

6.15 [25] <6.7> Use the DLX code for SAXPY above. Assume Tomasulo's algorithm for the hardware using the functional-unit count shown in Section 6.7. Assume the following latencies in clock cycles:

FP multiply	10
FP add	6
FP load/store	2
All integer operations	1

Show the state of the reservation stations and register status tables (as in Figure 6.37) when the branch is executed for the second time. Assume the branch was correctly predicted as taken. How many clock cycles does each loop iteration take?

6.16 [22] <6.8> Unwind the DLX code for SAXPY three times and schedule it for the standard DLX pipeline. Assume the FP latencies of Figure 6.29. When unwinding, you should optimize the code as in Section 6.8. Significant reordering of the code will be needed to maximize performance. What is the speedup over the original loop?

6.17 [25] <6.8> Assume a superscalar architecture that can issue any two independent operations in a clock cycle (including two integer operations). Unwind the DLX code for SAXPY three times and schedule it assuming the FP latencies of Figure 6.29. Assume one fully-pipelined copy of each functional unit (e.g., FP adder, FP multiplier). How many clock cycles will each iteration on the original code take? When unwinding, you should optimize the code as in Section 6.8. What is the speedup versus the original code?

6.18 [25] <6.8> In a superpipelined machine, rather than have multiple functional units, we would fully pipeline all the units. Suppose we designed a superpipelined DLX that had twice the clock rate of our standard DLX pipeline and could issue any two unrelated operations in the same time that the normal DLX pipeline issued one operation. Unroll the DLX SAXPY code three times and schedule it for this superpipelined machine assuming the FP latencies of Figure 6.29. How many clock cycles does each loop iteration take? Remember that these clock cycles are half as long as those on a standard DLX pipeline or a superscalar DLX.

6.19 [20] <6.8> Start with the SAXPY code and the machine used in Figure 6.49. Unroll the SAXPY loop three times, performing simple optimizations (as on page 315). Fill in a table like Figure 6.49 for the unrolled loop. How many clock cycles does each loop iteration take?

6.20 [35] <6.1–6.4> Change the DLX instruction simulator to be pipelined. Measure the frequency of empty branch-delay slots, the frequency of load delays, and the frequency of FP stalls for a variety of integer and FP programs. Also, measure the frequency of forwarding operations. Determine what the performance impact of eliminating forwarding and stalling would be.

6.21 [35] <6.6> Using a DLX simulator, create a DLX pipeline simulator. Explore the impact of lengthening the FP pipelines, assuming both fully pipelined and nonpipelined FP units. How does clustering of FP operations affect the results? Which FP units are most susceptible to changes in the FP pipeline length?

6.22 [40] <6.4–6.6> Write an instruction scheduler for DLX that works on DLX assembly language. Evaluate your scheduler using either profiles of programs or with a pipeline simulator. If the DLX C compiler does optimization, evaluate your scheduler's performance both with and without optimization.

6.23 [35] <6.4–6.6> Write a DLX pipeline simulator that uses Tomasulo's algorithm with the functional units described. Evaluate the performance of this machine compared to the straightforward DLX pipeline.

6.24 [Discussion] <6.7> Dynamic instruction scheduling requires a considerable investment in hardware. In return, this capability allows the hardware to run programs that could not be run at full speed with only compile-time, static scheduling. What tradeoffs should be taken into account in trying to decide between a dynamically and a statically scheduled scheme? What sort of situations in both hardware technology and program characteristics are likely to favor one approach or the other?

6.25 [Discussion] <6.7> There is a subtle problem that must be considered when implementing Tomasulo's algorithm. It might be called the "two ships passing in the night problem." What happens if an instruction is being passed to a reservation station during the same clock period as one of its operands is going onto the common data bus? Before an instruction is in a reservation station, the operands are fetched from the register file; but once it is in the station, the operands are always obtained from the CDB. Since the instruction and its operand tag are in transit to the reservation station, the tag cannot be matched against the tag on the CDB. So there is a possibility that the instruction will then sit in the reservation station forever waiting for its operand, which it just missed. How might this problem be solved? You might consider subdividing one of the steps in the algorithm into multiple parts. (This intriguing problem is courtesy of J. E. Smith.)

6.26 [Discussion] <6.8> Discuss the advantages and disadvantages of a superscalar implementation, a superpipelined implementation, and a VLIW approach in the context of DLX. What levels of instruction-level parallelism favor each approach? What other concerns would you consider in choosing which type of machine to build?

I'm certainly not inventing vector machines. There are three kinds that I know of existing today. They are represented by the Illiac-IV, the (CDC) Star machine, and the TI (ASC) machine. Those three were all pioneering machines. . . . 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)

7 Vector Processors

7.1 Why Vector Machines?

In the last chapter we looked at pipelining in detail and saw that pipeline scheduling, issuing multiple instructions per clock cycle, and more deeply pipelining a processor could as much as double the performance of a machine. Yet there are limits on the performance improvement that pipelining can achieve. These limits are set by two primary factors:

- Clock cycle time—The clock cycle time can be decreased by making the pipelines deeper, but a deeper pipeline will increase the pipeline dependences and result in a higher CPI. At some point, each increase in pipeline depth has a corresponding increase in CPI. As we saw in Section 6.10, very deep pipelining can slow down a processor.
- Instruction fetch and decode rate—This limitation, sometimes called the *Flynn bottleneck* (based on Flynn [1966]), prevents fetching and issuing of more than a few instructions per clock cycle. We saw that for most pipelined machines the average number of instruction issues per clock was less than one.

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

High-speed, pipelined machines are particularly useful for large scientific and engineering applications. A high-speed pipelined machine will usually use a cache to avoid forcing memory reference instructions to have very long latency. However, big, long-running, scientific programs often have very large active data sets that are often accessed with low locality, yielding poor performance from the memory hierarchy. The resulting impact is a decrease in cache performance. This problem could be overcome by not caching these structures if it were possible to determine the memory-access patterns and pipeline the accesses efficiently. Compiler assistance may help address this problem in the future (see Section 10.7).

Vector machines provide high-level operations that work on *vectors*—linear arrays of numbers. A typical vector operation might add two 64-entry, floating-point vectors to obtain a single 64-entry 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 operations have several important properties that solve most of the problems mentioned above:

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

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

As mentioned above, vector machines pipeline the operations on the individual elements of a vector. The pipeline includes not only the arithmetic operations (multiplication, addition, and so on), but also memory accesses and effective address calculations. In addition, most high-end vector machines allow multiple vector operations to be done at the same time, creating parallelism among the operations on different elements. In this chapter, we focus on vector machines that gain performance by pipelining and instruction overlap. In Chapter 10, we discuss parallel machines that operate on many elements in parallel rather than in pipelined fashion.

7.2 Basic Vector Architecture

A vector machine 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 machines allow the vectors to be dealt with as floating-point numbers (FP), as integers, or as logical data, though we will focus on floating point. The scalar unit is basically no different from the type of pipelined CPU discussed in Chapter 6.

There are two primary types of vector architectures: vector-register machines and memory–memory vector machines. In a *vector-register machine*, all vector operations—except load and store—are among the vector registers. These machines are the vector counterpart of a load/store architecture. All major vector machines being shipped in 1990 use a vector-register architecture; these include the Cray Research machines (CRAY-1, CRAY-2, X-MP, and Y-MP), the Japanese supercomputers (NEC SX/2, Fujitsu VP200, and the Hitachi S820), and the mini-supercomputers (Convex C-1 and C-2). In a *memory–memory vector machine* all vector operations are memory to memory. The first vector machines were of this type, as were CDC's machines. 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 chapter (Section 7.8) to discuss why they have not been as successful as vector-register architectures.

We begin with a vector-register machine consisting of the primary components shown in Figure 7.1 (page 354). This machine, which is loosely based on the CRAY-1, is the foundation for discussion throughout most of this chapter. We will call it DLXV; its integer portion is DLX, and its vector portion is the logical vector extension of DLX. The rest of this section examines how the basic architecture of DLXV relates to other machines.

The primary components of the instruction set architecture of DLXV are:

- Vector registers—Each vector register is a fixed-length bank holding a single vector. DLXV has eight vector registers, and each vector register holds 64 doublewords. Each vector register must have at least two read ports and one write port in DLXV. This will allow a high degree of overlap among vector operations to different vector registers. (The CRAY-1 manages to implement the register file with only a single port per register using some clever implementation techniques.)

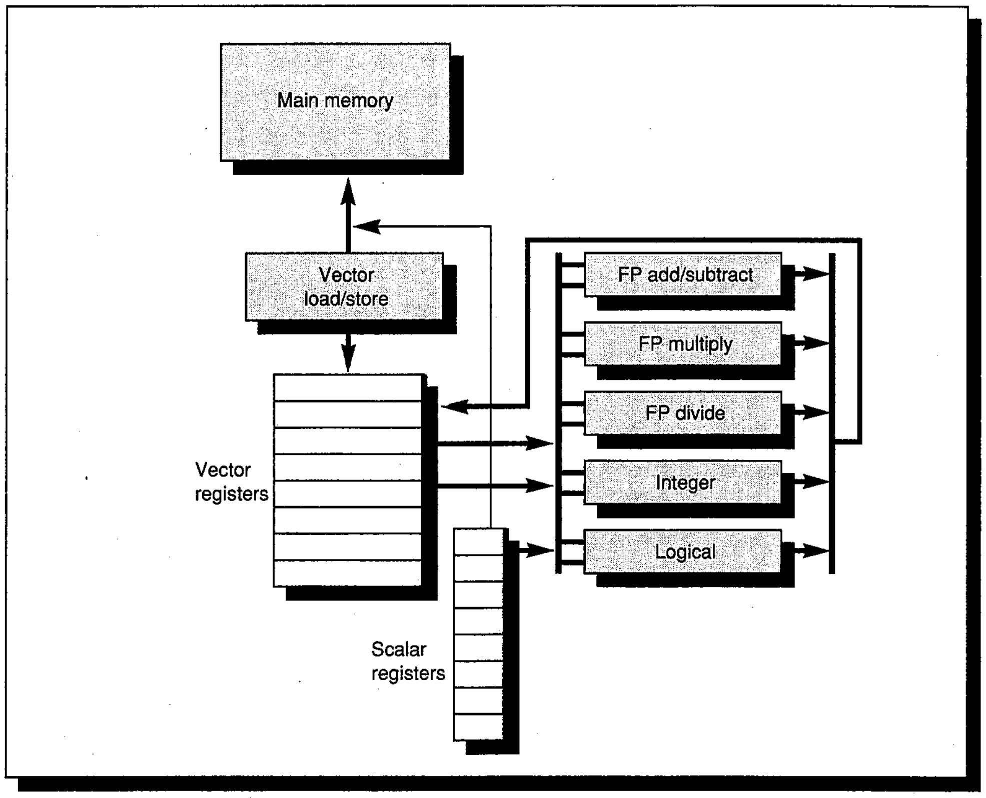

FIGURE 7.1 The basic structure of a vector-register architecture, DLXV. This machine has a scalar architecture just like DLX. There are also eight 64-element vector registers, and all the functional units are vector functional units. Special vector operations and vector loads and stores are defined. We show vector units for logical and integer operations. These are included so that DLXV looks like a standard vector machine, which usually includes these units. However, we will not be discussing these units except in the Exercises. In Section 7.6 we add chaining, which will require additional interconnect capability.

- 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 on conflicts for the functional units (structural hazards) and on conflicts for register accesses (data hazards). DLXV has five functional units, as shown in Figure 7.1. For simplicity, we will focus exclusively on the floating-point functional units.
- Vector load/store unit—A vector memory unit that loads or stores a vector to or from memory. The DLXV vector loads and stores are fully pipelined, so that words can be moved between the vector registers and memory with a bandwidth of one word per clock cycle, after an initial latency.
- A set of scalar registers—These can also provide data as input to the vector functional units, as well as compute addresses to pass to the vector load/store unit. These are the normal 32 general-purpose registers and 32 floating-point registers of DLX.

Figure 7.2 shows the characteristics of some typical vector machines, including the size and count of the registers, the number and types of functional units, and the number of load/store units.

In DLXV, the vector operation has the same name as the DLX name with the letter "V" appended. These are double-precision, floating-point, vector operations. (We have omitted single-precision FP operations and integer and logical operations for simplicity.) Thus, ADDV is an add of two double-precision vectors. The vector operations take as their input either a pair of vector registers (ADDV) or a vector register and a scalar register designated by appending "SV" (ADDSV). In the latter case, the value in the scalar register is used as the input for all operations—the operation ADDSV will add the contents of a scalar register to each element in a vector register. Vector operations always have a vector destination register. The names LV and SV denote vector load and vector store, and load or store an entire vector of double-precision data. One operand is

Machine	Year announced	Vector registers	Elements per vector register (64-bit elements)	Vector functional units	Vector load / store units
CRAY-1	1976	8	64	6: add, multiply, reciprocal, integer add, logical, shift	1
CRAY X-MP CRAY Y-MP	1983 1988	8	64	8: FP add, FP multiply, FP reciprocal, integer add, 2 logical, shift, population count/parity	2 loads 1 store
CRAY-2	1985	8	64	5: FP add, FP multiply, FP reciprocal/sqrt, integer (add shift, population count), logical	1
Fujitsu VP100/200	1982	8–256	32–1024	3: FP or integer add/logical, multiply, divide	2
Hitachi S810/820	1983	32	256	4: 2 integer add/logical, 1 multiply-add and 1 multiply/divide–add unit	4
Convex C-1	1985	8	128	4: multiply, add, divide, integer/logical	1
NEC SX/2	1984	8 + 8192	256 variable	16: 4 integer add/logical, 4 FP multiply/divide, 4 FP add, 4 shift	8
DLXV	1990	8	64	5: multiply, divide, add, integer add, logical	1

FIGURE 7.2 Characteristics of several vector-register architectures. The vector functional units include all operation units used by the vector instructions. The functional units are floating point unless stated otherwise. If the machine is a multiprocessor, the entries correspond to the characteristics of one processor. Each vector load/store unit represents the ability to do an independent, overlapped transfer to or from the vector registers. The Fujitsu VP200's vector registers are configurable: The size and count of the 8K 64-bit entries may be varied inversely to one another (e.g., 8 registers each 1K elements long, or 128 registers each 64 elements long). The NEC SX/2 has 8 fixed registers of length 256, plus 8K of configurable 64-bit registers. The reciprocal unit on the CRAY machines is used to do division (and square root on the CRAY-2). Add pipelines perform floating-point add and subtract. The multiply/divide–add unit on the Hitachi S810/200 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 machines use the vector FP multiply and divide units for vector integer multiply and divide, just like DLX, and several of the machines use the same units for FP scalar and FP vector operations.

the vector register to be loaded or stored; the other operand, which is a DLX general-purpose register, is the starting address of the vector in memory. Figure 7.3 lists the DLXV vector instructions. In addition to the vector registers, we need two additional special-purpose registers: the vector-length and vector-mask registers. We will discuss these registers and their purpose in Sections 7.3 and 7.6, respectively.

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

FIGURE 7.3 The DLXV vector instructions. Only the double-precision FP operations are shown. In addition to the vector registers there are two special registers VLR (discussed in Section 7.3) and VM (discussed in Section 7.6). The operations with stride are explained in Section 7.3, and the use of the index creation and indexed load/store operations are explained in Section 7.6.

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

```
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 (Single-precision or Double-precision A*X Plus Y) loop that forms the inner loop of the Linpack benchmark. Linpack is a collection of linear algrebra routines; the Gaussian elimination portion of Linpack is the segment used as a benchmark. SAXPY represents a small piece of the program, though it takes most of the time in the benchmark.

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

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

Answer Here is the DLX code.

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

Here is the code for DLXV for DAXPY.

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

There are some interesting comparisons between the two code segments in the example above. The most dramatic is that the vector machine greatly reduces the dynamic instruction bandwidth, executing only 6 instructions versus almost 600 for DLX. This reduction occurs both because the vector operations work on

64 elements, and because the overhead instructions that constitute nearly half the loop on DLX are not present in the DLXV code.

Another important difference is the frequency of pipeline interlocks. In the straightforward DLX code every ADDD must wait for a MULTD, and every SD must wait for the ADDD. On the vector machine, each vector instruction operates on all the vector elements independently. Thus, pipeline stalls are required only once per vector operation, rather than once per vector element. In this example, the pipeline-stall frequency on DLX will be about 64 times higher than it is on DLXV. The pipeline stalls can be eliminated on DLX by using software pipelining or loop unrolling (as we saw in Chapter 6, Section 6.8). However, the large difference in instruction bandwidth cannot be reduced.

Vector Start-up Time and Initiation Rate

Let's investigate the running time of this vector code on DLXV. The running time of each vector operation in the loop has two components—the *start-up time* and the *initiation rate*. 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. For example, a latency of 10 clock cycles means both that the operation takes 10 clock cycles and that the pipeline is 10 deep. (In discussions of the performance of vector operations, clock cycles are customarily used as the metric.) The initiation rate is the time per result once a vector instruction is running; this rate is usually one per clock cycle for individual operations, though some supercomputers have vector operations that can produce 2 or more results per clock, and others have units that may not be fully pipelined. The *completion rate* must at least equal the initiation rate—otherwise there is no place to put results. Hence, the time to complete a single vector operation of length n is:

$$\text{Start-up time} + n * \text{Initiation rate}$$

Example

Suppose the start-up time for a vector multiply is 10 clock cycles. After start-up the initiation rate is one per clock cycle. What is the number of clock cycles per result (i.e., one element of the vector) for a 64-element vector?

Answer

$$\text{Clock cycles per result} = \frac{\text{Total time}}{\text{Vector length}}$$

$$= \frac{\text{Start-up time} + 64 * \text{Initiation rate}}{64}$$

$$= \frac{10 + 64}{64} = 1.16 \text{ clock cycles.}$$

Figure 7.4 shows the effect of start-up time and initiation rate on vector performance. The effect of increasing start-up time on a slow-running vector is small, while the same increase in start-up time on a system with an initiation rate of one per clock decreases performance by a factor of nearly two.

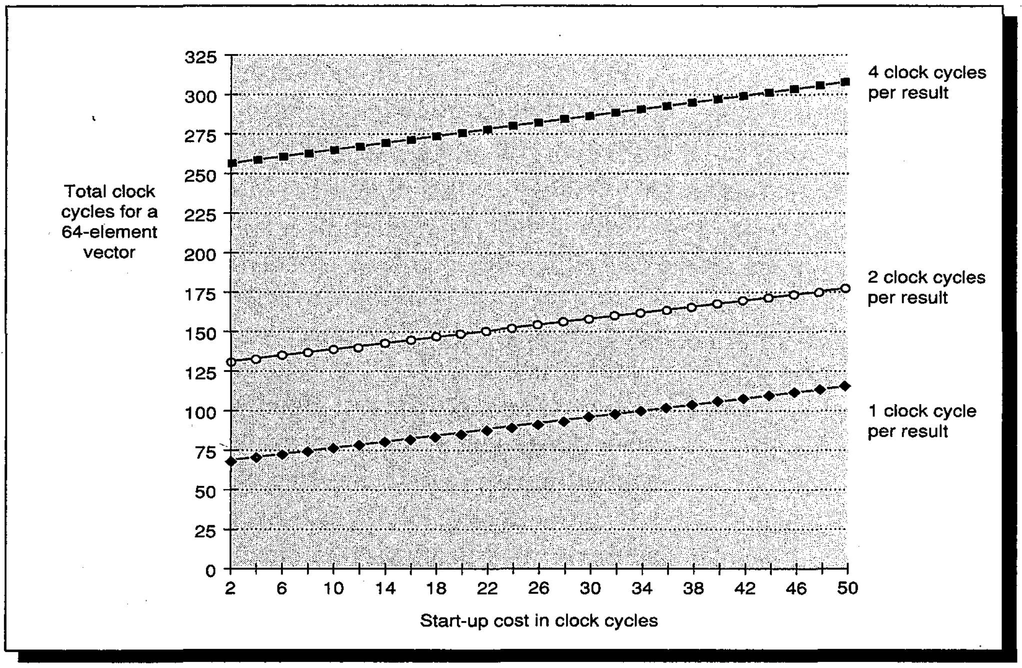

FIGURE 7.4 Total running time increases with start-up cost from 2 to 50 clock cycles per operation on the x axis. The impact of start-up time is much greater for fast-running than for slow-running vectors. The operation running at one clock cycle per result increases its run time by 75%, while the operation running at four clock cycles per result increases by less than 20%.

What determines the start-up and initiation rates? Let's first consider the operations that do not involve a memory access. For register–register operations the start-up time (in clock cycles) is equal to the depth of the functional unit pipeline, since this is the time to get the first result. In the earlier example, the depth of 10 gave a start-up time of 10 clock cycles. In the next few sections, we will see that there are other costs involved that increase the start-up time. The initiation rate is determined by how often the corresponding vector functional unit can accept an operand. If it is fully pipelined, then it can start an operation on new operands every clock cycle, yielding an initiation rate of one per clock (as in the earlier example).

Start-up time for an operation comprises the total latency for the functional unit implementing that operation. If the initiation rate is to be kept at 1 clock 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. Pipeline depth, then, is determined by the complexity of the operation and the clock cycle time of the machine. The pipeline depths of functional units vary widely—from 2 to 20 stages is not uncommon—though the most heavily used units have start-up times of 4 to 8 clocks.

For DLXV, we will choose the same pipeline depths as the CRAY-1. All functional units are fully pipelined. Pipeline depths are six clock cycles for floating-point add and seven clock cycles for floating-point multiply. If a vector computation depends on an uncompleted computation and will need to be stalled, it adds an extra 4-clock-cycle start-up penalty. This penalty is typical on vector machines and arises due to the lack of bypassing: the penalty is the time to write and then read the operands and is only seen when there is a dependence. Thus, back-to-back dependent vector operations will see the full latency of a vector operation. On DLXV, as on most vector machines, independent vector operations using different functional units can issue without any penalty or delay. Independent vector operations may also be fully overlapped, and each instruction issue only takes one clock. Thus, when the operations are independent and different, DLXV can overlap vector operations, just as DLX can overlap integer and floating-point operations.

Because DLXV is fully pipelined, the initiation rate for a vector instruction is always 1. However, a sequence of vector operations will not be able to run at that rate, due to start-up costs. The term *sustained rate* is applied to this situation and refers to the time per element for a collection of related vector operations. Here an element is not the result of a single vector operation, but one result of a series of vector operations. The time per element, then, is the time required for each operation to produce an element. For example, in the SAXPY loop, the sustained rate will be the time to compute and store one element of the result vector Y.

Example For a vector length of 64 on DLXV and the following two vector instructions, what is the sustained rate for the sequence, and the effective number of floating-point operations per clock for the sequence?

```
MULTV V1,V2,V3
ADDV  V4,V5,V6
```

Answer Let's look at the start and completion times of these independent operations (remember that the start-up times are 7 cycles for multiply and 6 cycles for add):

Operation	Start	Complete
MULTV	0	7 + 64 = 71
ADDV	1	1 + 6 + 64 =71

The sustained rate is one element per clock—remember that sustained rate requires all vector operations to produce a result. The sequence executes 128

FLOPs (FLoating-point OPerations) in 71 clock cycles, for a rate of 1.8 FLOPs per clock. A vector machine can sustain a throughput of more than one operation per clock cycle by issuing independent vector operations to different vector functional units.

The behavior of the load/store vector unit is significantly more complicated. 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. Typically, penalties for start-ups on load/store units are higher than for functional units—up to 50 clock cycles on some machines. For DLXV we will assume a low start-up time of 12 clock cycles, since the CRAY-1 and CRAY X-MP have load/store start-up times of between 9 and 17 clock cycles. For stores, we will not usually care about the start-up time, since stores do not directly produce results. However, when an instruction must wait for a store to complete (as a load might have to with only one memory pipeline), the load may see part or all of the 12-cycle latency of a store. Figure 7.5 summarizes the start-up penalties for DLXV vector operations.

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

FIGURE 7.5 Start-up penalties on DLXV. These are the start-up penalties in clock cycles for DLXV vector operations. When a vector instruction depends on another vector instruction that has not completed at the time the second vector instruction issues, the start-up penalty is increased by 4 clock cycles.

To maintain an initiation rate of one word fetched or stored per clock, the memory system must be capable of producing or accepting this much data. This is usually done by creating multiple *memory banks*. Each memory bank is like a small, separate memory that can access different addresses in parallel with other banks. The words are then transferred from the memory at the maximum rate (one per clock in DLXV).

There are two possible implementation techniques for memory banks. One approach is to synchronize all the banks and to access them in parallel, latching the result in each bank. Once the result is latched, the next access can begin while the words are transferred. An alternative implementation technique uses independent bank phasing. On the first access, all the banks are accessed in parallel, and then the words are transferred one at a time from the banks. Once a bank has transmitted or stored its data, it begins the next access immediately. The first approach (synchronized accesses) requires more latches, but has simpler control than an approach that uses independent bank phasing. The concept of memory banks is similar to but not identical to interleaving, as we will see in Figure 7.6. We discuss interleaving extensively in Chapter 8, Section 8.4.

Assuming each bank is one double-precision-word wide, if an initiation rate of one per clock is to be maintained, the following must hold:

$$\text{Number of memory banks} \geq \text{Memory-bank access time in clock cycles}$$

To see why this relationship exists, think about a vector load of 64 double-precision words. Let the addresses of the vector elements be given by k_i, where

$$k_i = \text{Starting address of the vector} + (i\text{-}1) * \text{Distance between vector elements.}$$

For double-precision vector elements that are adjacent, the distance between elements will be 8 bytes. The addresses of the vector elements to be accessed by a bank will be the values of k_i such that

$$k_i \bmod \text{number of banks} = \text{Bank number}$$

Let's look at the first access by each bank. After a time equal to the memory-access time, all the memory banks will have fetched a double-precision word, and the words can begin returning to the vector registers. (This requires, of course, that the accesses be aligned on doubleword boundaries.) Words are sent serially from the banks, starting with the bank fetching from the lowest address. If the banks are synchronized, the next accesses start immediately; if the banks are phased, then the next access begins after an element is transmitted from the bank. In either case, a bank begins its next access at a byte address that is (8 * number of banks) higher than the last byte address. Because the memory-access time in clock cycles is less than the number of memory banks and because the words are transferred from the banks in round-robin order at a rate of one transfer per clock cycle, a bank will complete the next access before its turn to transmit data comes again. To simplify addressing, the number of memory banks is usually made a power of two. As we will see shortly, designers will probably want to have more than the minimum number of required banks so as to minimize memory stalls.

Example

Suppose we want to fetch a vector of 64 elements starting at byte address of 136, and a memory access takes 6 clocks. How many memory banks must we have? With what addresses are the banks accessed? When will the various elements arrive at the CPU?

Answer

Six clocks per access require at least 6 banks, but because we want the number of banks to be a power of two, we choose to have 8 banks. Figure 7.6 shows what byte addresses each bank accesses within each time period. Remember that a bank begins a new access as soon as it has completed the old access.

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

FIGURE 7.6 Memory addresses (in bytes) by bank number and time slot at which access begins. The exact time when a bank transmits its data is given by the address it accesses minus the starting address divided by 8 plus the memory latency (6 clocks). It is important to observe that Bank 0 accesses a word in the next block (i.e., it accesses 192 rather than 128 and then 256 rather than 192, and so on). If Bank 0 were to start at the lower address we would require an extra cycle to transmit the data, and we would transmit one value unnecessarily. While this problem is not severe for this example, if we had 64 banks, up to 63 unnecessary clock cycles and transfers could occur. The fact that Bank 0 does not access a word in the same block of 8 distinguishes this type of memory system from interleaved memory. Normally, interleaved memory systems combine the bank address and the base starting address by concatenation rather than addition. Also, interleaved memories are almost always implemented with synchronized access. Memory banks require address latches for each bank, which are not normally needed in a system with only interleaving.

Figure 7.7 shows the timing for the first few sets of accesses for an 8–bank system with a 6–clock-cycle access latency. Two important observations about these two figures are these: First, notice that the exact address fetched by a bank is largely determined by the lower-order bits in the bank number; however, the initial access to a bank is always within 8 doublewords of the initial address. Second, notice that once the initial latency is overcome (6 clocks in this case), the pattern is to access a bank every n clock cycles, where n is the total number of banks (n=8 in this case).

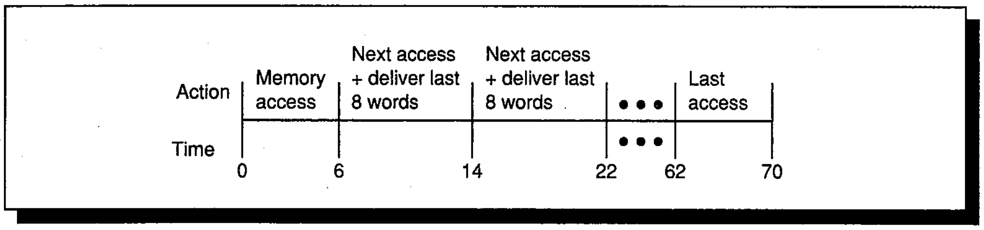
Action
Memory access
Next access + deliver last 8 words
Next access + deliver last 8 words
• • •
Last access
Time
0
6
14
22
• • •
62
70

FIGURE 7.7 Access timing for the first 64 double-precision words of the load. After the 6–clock-cycle initial latency, 8 double-precision words are returned every 8 clock cycles.

The number of banks in the memory system and the pipeline depth in the functional units are essentially counterparts, since they determine the initiation rates for operations using these units. The processor cannot access memory faster than the memory cycle time. Thus, if memory is built from DRAM, where cycle time is about twice the access time, the processor will usually need twice as many banks as the computations above would give. This characteristic of DRAM is discussed further in Chapter 8, Section 8.4.

7.3 Two Real-World Issues: Vector Length and Stride

This section deals with two issues that transpire in real programs. These are what to do when the vector length in a program is not exactly 64, and how to deal with nonadjacent elements in vectors when a matrix is laid out in memory. First, let's deal with the issue of vector length.

Vector-Length Control

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

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

The size of all the vector operations depends on n, which may not even be known until run-time! The value of n might also be a parameter to the procedure 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 any greater than the length of the vector registers. This solves our problem as long as the real length is less than the *maximum vector length* (MVL) defined by the machine.

What if the value of n is not known at compile time, and thus may be greater than MVL? To tackle this problem, 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. The strip-mined version of the SAXPY loop written in FORTRAN, the major language used for scientific applications, is shown with C-style comments:

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

The term n / MVL represents truncating integer division (which is what FORTRAN does) and is used throughout this section. The effect of this loop is to block the vector into segments which are then processed by the inner loop. The length of the first segment is (n mod MVL) and all subsequent segments are of length MVL. This is depicted in Figure 7.8.

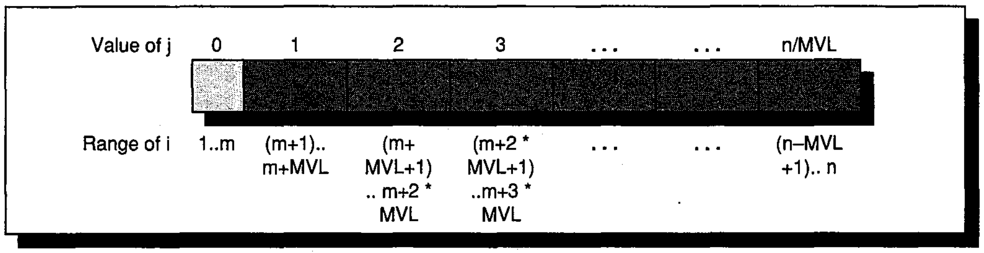

FIGURE 7.8 A vector of arbitrary length processed with strip mining. All blocks but the first are of length MVL, utilizing the full power of the vector machine. In this figure, the variable *m* is used for the expression (n mod MVL).

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

In the previous section, start-up overhead could be computed independently for each vector operation. With strip mining, a significant percentage of the start-up cost will be the strip-mining overhead itself; and, therefore, computing the start-up overhead will be more complex.

Let's see how significant these added overheads are. Consider a simple loop:

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

The compiler will generate two nested loops for this code, just as our earlier example does. The inner loop contains a sequence of two vector operations, LV (load vector) followed by SV (store vector). Each loop iteration of the original vector operation would require two clocks if there were no start-up penalties of any kind. The start-up penalties consist of two types: vector start-up overhead and strip-mining overhead. For DLXV the vector start-up overhead is 12 clock cycles for the vector load plus a 4-clock-cycle delay because the store depends on the load, for a total of 16 clock cycles. We can ignore the store latency, since nothing depends on it. Figure 7.9 (page 366) shows the impact of the vector start-up cost alone as the vector grows from length 1 to length 64. This start-up cost can decrease the throughput rate by a factor of as much as 9, depending on the vector length.

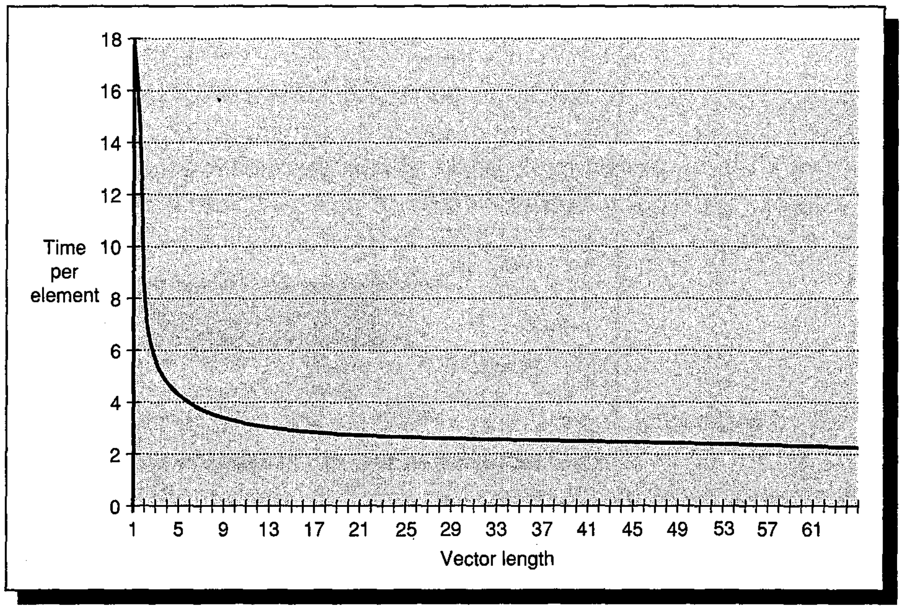

FIGURE 7.9 The impact of just the vector start-up cost on a loop consisting of a vector assignment. For short vectors, the impact of the 16-cycle start-up cost is enormous, decreasing performance by up to nine times. The strip-mining overhead has not been included.

In Section 7.4, we will see a unified performance model that incorporates all the start-up and overhead costs. First, let's examine how to implement vectors with nonsequential memory accesses.

Vector Stride

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

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

At the statement labeled 10 we could vectorize the multiplication of each row of B with each column of C and strip-mine the inner loop with k as the index variable. To do so, we must consider how adjacent elements in B and adjacent elements in C are addressed. When an array is allocated memory it is linearized and must be laid out in either *row-major* or *column-major* order. Row-major order, used by most languages except FORTRAN, lays out the rows first, making elements B(i,j) and B(i,j+1) adjacent. Column-major order, used by FORTRAN, makes B(i,j) and B(i+1,j) adjacent. Figure 7.10 illustrates these two alternatives. Let's look at the accesses to B and C in the inner loop of the matrix multiply. In FORTRAN, the accesses to the elements of B will be nonadjacent in memory, and each iteration will access an element that is separated by an entire row of the array. In this case, 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.

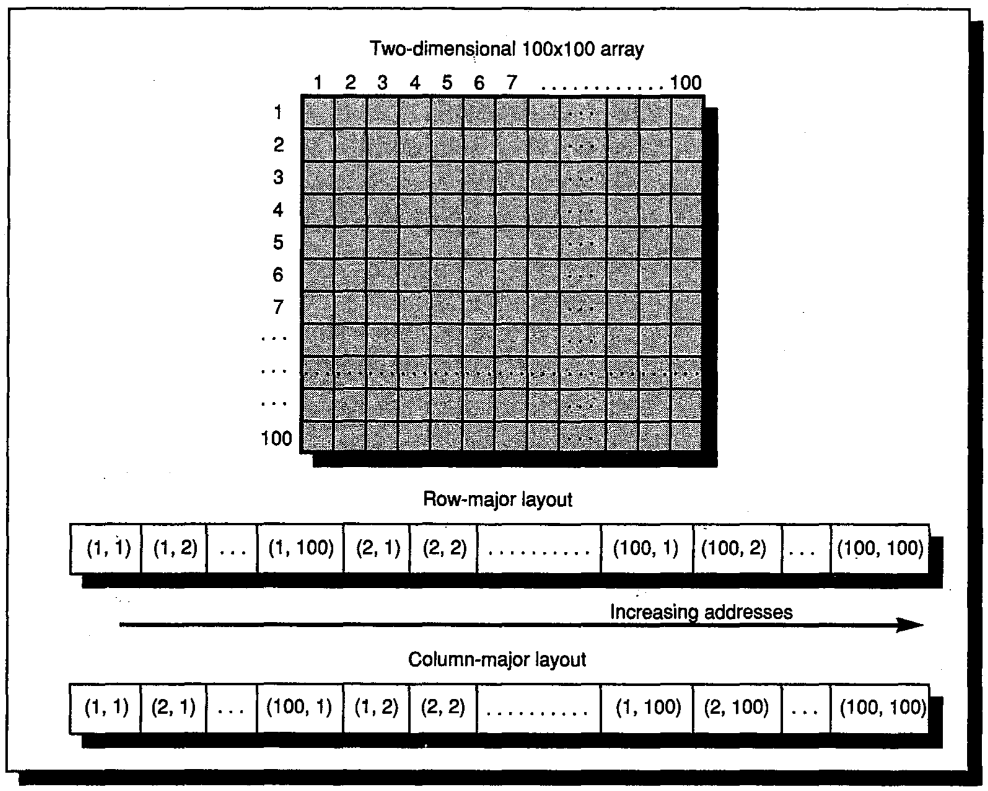

FIGURE 7.10 Matrix for a two-dimensional array and corresponding layouts in one-dimensional storage. In row-major order, successive row elements are adjacent in storage, while in column-major order, successive column elements are adjacent. It is easy to imagine extending this to arrays with more dimensions.

This distance separating elements that are to be merged into a single vector 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 doubleword (8 bytes), separating successive elements, and matrix B has a stride of 100, or 100 doublewords (800 bytes).

Once a vector is loaded into a vector register it acts as if it had logically adjacent elements. This enables a vector-register machine to handle strides greater than one, called *nonunit strides*, by making more general vector-load and vector-store operations. For example, if we could load a row of B into a vector register, we could then treat the row as logically adjacent.

Thus, it is desirable for the vector load and store operations to specify a stride in addition to a starting address. On a DLXV, where the addressable unit is a byte, the stride for our example would be 800. The value must be computed dynamically, since the size of the matrix may not be known at compile time, or—just like vector length—may change for different executions of the same statement. The vector stride, like the vector starting address, can be put in a general-purpose register, where it is used for the life of the vector operation. Then the DLXV instruction `LVWS` (Load Vector With Stride) can be used to fetch the vector into a vector register. Likewise, when a nonunit stride vector is being stored, `SVWS` (Store Vector With Stride) can be used. In some vector machines the loads and stores always have a stride value stored in a register, so there is only a single instruction.

Memory-unit complications can occur from supporting strides greater than one. Earlier, we saw that a vector-memory operation could proceed at full speed if the number of memory banks was at least as large as the memory-access time in clock cycles. However, once nonunit strides are introduced it becomes possible to request accesses from the same bank at a higher rate than the memory-access time. This situation is called *memory-bank conflict* and results in each load seeing a larger portion of the memory-access time. A memory-bank conflict occurs whenever the same bank is asked to do an access before it has completed another. Thus, a bank conflict, and hence a stall, will occur if:

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

Example

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

Answer

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

Memory bank conflicts will not occur if the stride and number of banks are relatively prime with respect to each other and there are enough banks to avoid conflicts in the unit-stride case. Increasing the number of memory banks to a number greater than the minimum to prevent stalls with a stride of length 1 will decrease the stall frequency for some other strides. For example, with 64 banks, a stride of 32 will stall on every other access, rather than every access. If we originally had a stride of 8 and 16 banks, every other access would stall; while with 64 banks, a stride of 8 will stall on every eighth access. If we have multiple memory pipelines, we will also need more banks to prevent conflicts. In the

1990s, most vector supercomputers have at least 64 banks, and some have as many as 512.

7.4 A Simple Model for Vector Performance

This section presents a model for understanding the performance of a vectorized loop. There are three key components of the running time of a strip-mined loop whose body is a sequence of vector instructions:

1. The time for each vector operation in the loop to process one element, ignoring the start-up costs, which we call $T_{element}$. The vector sequence often has a single result, in which case T_{element} is the time to produce an element in that result. If the vector sequence produces multiple results, T_{element} is the time to produce one element in each result. This time depends only on the execution of vector instructions. We will see an example shortly.

2. The overhead for each strip-mined block of vector instructions. This overhead consists of the cost of executing the scalar code for strip mining of each block, T_{loop}, plus the vector start-up cost for each block, T_{start}.

3. The overhead from computing the starting addresses and setting up the vector control. This occurs once for the entire vector operation. This time, T_{base}, consists solely of scalar overhead instructions.

These 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:

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

The values of T_{start} and T_{loop} are both compiler and machine dependent, while the value of T_{element} depends mainly on the hardware. The exact vector sequence affects all three values; the effect on T_{element} is probably the most pronounced, with T_{start} and T_{loop} less affected.

For simplicity, we will use constant values for T_{base} and for T_{loop} on DLXV. Based on a variety of measurements of CRAY-1 vector execution, the values chosen are 10 for T_{base} and 15 for T_{loop}. At first glance, you might think that these values, especially T_{loop}, are too small. The overhead in each loop requires: setting up the vector starting addresses and the strides, incrementing counters, and executing a loop branch. However, these scalar instructions can be overlapped with the vector instructions, minimizing the time spent on these overhead functions. The values of T_{base} and T_{loop} of course depend on the loop structure, but the dependence is slight compared to the connection between the vector code and the values of T_{element} and T_{start}.

Example

What is the execution time for the vector operation A = B * s, where s is a scalar and the length of the vectors A and B is 200?

Answer

Here is the strip-mined DLXV code, assuming the addresses of A and B are initially in `Ra` and `Rb`, and s is in `Fs`:

```
      ADDI    R2,R0,#1600 ;no. bytes in vector
      ADD     R2,R2,Ra    ;end of A vector
      ADDI    R1,R0,#8    ;strip-mined length
      MOVI2S  VLR,R1      ;load vector length
      ADDI    R1,R0,#64   ;length in bytes
      ADDI    R3,R0,#64   ;vector length of other pieces
loop: LV      V1,Rb       ;load B
      MULTSV  V2,Fs,V1    ;vector * scalar
      SV      Ra,V2       ;store A
      ADD     Ra,Ra,R1    ;next segment of A
      ADD     Rb,Rb,R1    ;next segment of B
      ADDI    R1,R0,#512  ;full vector length (bytes)
      MOVI2S  VLR,R3      ;set length to 64
      SUB     R4,R2,Ra    ;at the end of A?
      BNZ     R4,LOOP     ;if not, go back
```

From this code, we can see that: $T_{element} = 3$, for the load, multiply and store of each value of the vector. Furthermore, our assumptions for DLXV are $T_{loop} = 15$ and $T_{base} = 10$. Let's use our basic formula:

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

$$T_{200} = 10 + (4) * (15 + T_{start}) + 200 * 3$$

$$T_{200} = 10 + 4 * (15 + T_{start}) + 600 = 670 + 4 * T_{start}$$

The value of T_{start} is the sum of

- The vector load start-up of 12 clock cycles,
- The 4–clock-cycle stall due to the dependence between the load and multiply,
- A 7–clock-cycle start-up for the multiply, plus
- A 4–clock-cycle stall due to the dependence between the multiply and store.

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

$$T_{start} = 12 + 4 + 7 + 4 = 27$$

So, the overall value becomes

$$T_{200} = 670 + 4 * 27 = 778$$

The execution time per element with all start-up costs is then $\frac{778}{200} = 3.9$, compared with an ideal case of 3.

Figure 7.11 shows the overhead and effective rates per element for the above example (A = B*s) with various vector lengths. Compared to the simpler model of start-up, illustrated in Figure 7.9 on page 366, we see that the overhead accounting for all sources is higher. In this example, the vector start-up cost, which is what is plotted in Figure 7.9, accounts for only about half the total overhead per element.

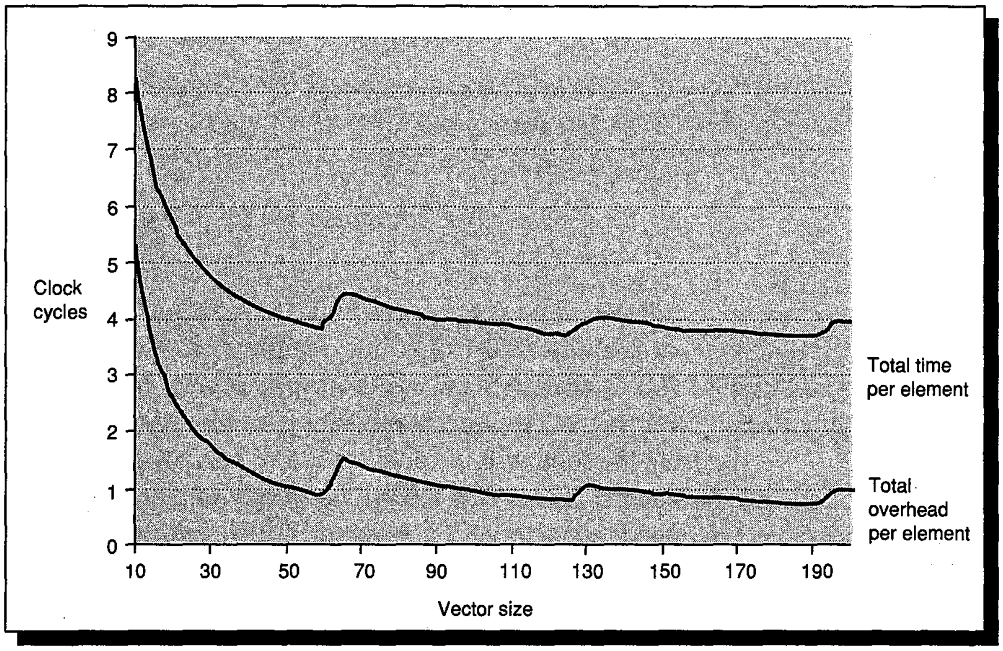

FIGURE 7.11 This shows the total execution time per element and the total overhead time per element, versus the vector length for the example on page 370. For short vectors the total start-up time is more than one-half of the total time, while for long vectors it reduces to about one-third of the total 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 $T_{loop} + T_{start}$.

7.5 Compiler Technology for Vector Machines

To make effective use of a vector machine a compiler must be able to recognize that a loop (or part of a loop) is vectorizable and generate the appropriate vector code. This involves determining what dependences exist among the operands in the loop. For now, we will consider only dependences that occur when an operand is written at one point and read at a later point. These correspond to RAW (read after write—see page 264) hazards. Consider a loop like this one:

```
            do 10 i=1,100
1              A(i+1) = A(i) + B(i)
2              B(i+1) = B(i) + A(i+1)
10          continue
```

Call the numbered statements 1 and 2 in the loop body S1 and S2, respectively. The possible different types of dependences are

1. S1 uses a value computed by S1 in an earlier iteration. This is true for S1 since iteration i+1 uses the value `A(i)` that was computed in iteration i as `A(i+1)`. The same is true of S2 for `B(i)` and `B(i+1)`.
2. S1 uses a value computed by S2 in an earlier iteration. This is true since S1 uses the value of `B(i+1)` in iteration i+1 that is computed by S2 in iteration i.
3. S2 uses a value computed by S1 in the same iteration. This is true for the value `A(i+1)`.

Because the vector operations are pipelined and the latency may be quite long, an early iteration may not complete before a later iteration begins: Thus, the values that will be written by the early iteration may not have been written before the later iteration begins. Consequently, if situation 1 or 2 exists, vectorizing the loop will introduce a RAW hazard—a hazard that a vector machine does not check for. This means that if any of the three dependences in situation 1 and 2 exist, the loop is not vectorizable, and the compiler will not generate vector instructions for this code. In situation 3, the normal hazard-detection hardware could handle the situation. A loop containing only dependences like those in situation 3 can therefore be vectorized, as we will see soon. The dependences in the first two situations, which involve the use of values computed on earlier loop iterations, are called *loop-carried dependences.*

The first task of the compiler is to determine whether there are any loop-carried dependences within the loop body. The compiler accomplishes this with a dependence-analysis algorithm. Because the statements in the loop body involve arrays, dependence analysis is complex. (If there weren't arrays, there would be nothing to vectorize.) The simplest case occurs when an array name appears only on one side of an assignment statement. Take, for example, this variation of our earlier loop:

```
          do   10 i=1,100
               A(i) = B(i) + C(i)
               D(i) = A(i) * E(i)
10        continue
```

If the arrays A, B, C, D, and E are different, then no loop-carried dependence can exist. There is a dependence between the two statements for the vector A. If the compiler realized that there were two accesses to A, it might try not to reload A the second statement, instead doing the vector multiply using the result register from the vector add. In this case, the processor would see the potential RAW hazard and stall the issue of the vector multiply. If the compiler stored A and reloaded it, then the loads and stores would occur in order, yielding correct execution.

Often the same name appears as both a source and destination within a loop, as it did in the SAXPY loop. There, Y appears on both sides of the assignment:

```
          do 10 i=1,100
               Y(i) = a*X(i) + Y(i)
     10   continue
```

In this case there is still no loop-carried dependence because the assignment to Y does not depend on a value of Y computed in an earlier iteration. However, the following loop, which is called a *recurrence*, does contain a loop-carried dependence:

```
          do 10 i=2,100
               Y(i) = Y(i-1) + Y(i)
     10   continue
```

The dependence can be seen by unwinding the loop: In iteration *j* the value of Y(*j*–1) is used, but that element is stored in iteration *j*–1, creating a loop-carried dependence.

How does the compiler detect dependences in general? Suppose we have written to an array element with index value $a * i + b$ and accessed with index value $c * i + d$, where i is the for-loop index variable that runs from m to n. A dependence exists if two conditions hold:

1. There are two iteration indices, j and k, both within the limits of the for loop.

2. The loop stores into an array element indexed by $a*j+b$ and later fetches from that **same** array element when it is indexed by $c*k+d$. That is, $a*j+b = c*k+d$.

In general, we may not be able to determine whether a dependence exists at compile time. For example, the values of *a, b, c,* and *d* may not be known, making it impossible to tell if a dependence exists. In other cases, the dependence testing may be very expensive but decidable at compile time. For example, the accesses may depend on the iteration indices of multiply nested loops. Many programs do not contain these complex structures, but instead contain simple indices where *a, b, c,* and *d* are all constants. For these cases, it is possible to devise reasonable tests for dependence.

A simple and sufficient test used to detect dependences is the *greatest common divisor*, or GCD. It is based on the observation that if a loop-carried dependence exists, then GCD (c,a) must **divide** $(d–b)$. (Remember that an integer, x, *divides* another integer, y, if there is no remainder when we do the division $\frac{y}{x}$ and get an integer result.) The GCD test is sufficient to guarantee that no dependence exists (see Exercise 7.10); however, there are cases where the GCD test succeeds, but no dependence exists. For example, this can arise because the GCD test does not take the loop bounds into account. A more complex test is the Banerjee test, named after U. Banerjee [1979], that accounts for loop bounds, but is still not exact. An exact test can always be done by solving equations for integer values, but this can be expensive for complex loop structures.

Example

Use the GCD test to determine whether dependences exist in the following loop:

```
        do 10 i=1,100
10         X(2*i+3) = X(2*i) * 5.0
```

Answer

Given the values *a*=2, *b*=3, *c*=2, and *d*=0, then GCD(*a*,*c*) = 2, and *d*–*b* = –3. Since 2 does not divide –3, no dependence is possible.

A *true data dependence* arises from a RAW hazard and will prevent vectorization of the loop as a single vector sequence. There are cases where the loop can be vectorized as two separate vector sequences (see Exercise 7.11). There are also dependences corresponding to a WAR (write after read) hazard, called an *antidependence*, and to a WAW (write after write) hazard, called an *output dependence*. Antidependences and output dependences are not true data dependences. They are name conflicts and can be eliminated by renaming of registers in the compiler in a method similar to how Tomasulo's algorithm renames registers at run time (see Section 6.7 in Chapter 6). Vectorizing compilers often use compile-time renaming to eliminate antidependences and output dependences.

Example

The following loop has an antidependence (WAR) and an output dependence (WAW). Find all the true dependences, output dependences, and antidependences, and eliminate the output dependences and antidependences by renaming.

```
         do 10 i=1,100
1               Y(i) = X(i) / s
2               X(i) = X(i) + s
3               Z(i) = Y(i) + s
4               Y(i) = s - Y(i)
10       continue
```

Answer

There are true dependences from statement 1 to statement 3 and from statement 1 to statement 4 because of `Y(i)`. These are not loop carried, so they will not prevent vectorization. However, the dependences will force statements 3 and 4 to wait for statement 1 to complete, even though statements 3 and 4 use a different functional unit than statement 1. In the next section we will see a technique for eliminating this serialization.

There is an antidependence from statement 1 to statement 2, and an output dependence from statement 1 to statement 4. The following version of the loop eliminates these false (or pseudo) dependences.

```
        do 10 i=1,100
C       Y renamed to T to remove output dependence
1          T(i) = X(i) / s
C       X renamed to X1 to remove antidependence
2          X1(i) = X(i) + s
3          Z(i) = T(i) + s
4          Y(i) = s - T(i)
10      continue
```

After the loop the variable X has been renamed X1. In code that follows the loop, the compiler can simply replace the name X by X1. Renaming does not require an actual copy operation; it can be done by substituting names or by register allocation.

Besides deciding which loops are vectorizable, the compiler must generate strip-mining code and allocate vector registers. Most vectorization transformations are done at the source level, although some optimizations involve coordinating high-level source transformations with lower-level, machine-dependent transformations. Efficient allocation of vector registers is such an optimization and is perhaps the most difficult optimization—one that many vectorizing compilers do not attempt.

Effectiveness of Vectorization Techniques

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, 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.

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, discussed in Section 2.7 of Chapter 2. These benchmarks are large, real scientific applications. Figure 7.12 (page 376) shows the percentage of floating-point operations in each benchmark and the percentage executed in vector mode on the CRAY X-MP. The wide variation in level of vectorization has been observed by several studies of the performance of applications on vector machines. While better compilers might improve the level of vectorization in some of these programs, most will require rewriting to achieve significant increases in vectorization. For example, let's look at our version of the Spice benchmark in detail. In Spice with the input chosen we found that only 3.7% of the floating-point operations are executed in vector mode on the CRAY X-MP, and the vector version runs only 0.5% faster than the scalar version. Clearly, a new program or a significant rewrite will be needed to obtain the benefits of a vector machine on Spice.

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

FIGURE 7.12 Level of vectorization among the Perfect Club benchmarks when executed on the CRAY X-MP. The first column contains the percentage of operations that are floating point, while the second contains the percentage of FP operations executed in vector instructions. Note that this run of Spice with different inputs shows a higher vectorization ratio.

There is also tremendous variation in how well compilers do in vectorizing programs. As a summary of the state of vectorizing compilers, consider the data in Figure 7.13, which shows the extent of vectorization for different machines using a test suite of 100 hand-written FORTRAN kernels. The kernels were designed to test vectorization capability and can all be vectorized by hand; we will see several examples of these loops in the Exercises.

Machine	Compiler	Completely vectorized	Partially vectorized	Not vectorized
Ardent Titan-1	FORTRAN V1.0	62	6	32
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
Stellar GS 1000	F77 prerelease	48	11	41

FIGURE 7.13 Result of applying vectorizing compilers to the 100 FORTRAN test kernels. For each machine we indicate how many loops were completely vectorized, partially vectorized, and unvectorized. These loops were collected by Callahan, Dongarra, and Levine [1988]. The machines shown are those mentioned at some point in this chapter. Two different compilers for the CRAY X-MP show the large dependence on compiler technology.

7.6 Enhancing Vector Performance

Three techniques for improving the performance of vector machines are discussed in this section. The first deals with making a sequence of dependent vector operations run faster. The other two deal with expanding the class of loops that can be run in vector mode. The first technique, chaining, originated in the CRAY-1, but is now supported on many vector machines. The techniques discussed in the second and third parts of this section are taken from a variety of machines and are, in general, more extensive than the capabilities provided on the CRAY-1 or CRAY X-MP architectures.

Chaining—The Concept of Forwarding Extended to Vector Registers

Consider the simple vector sequence

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

In DLXV as it currently stands these two instructions run in time equal to

$$T_{\text{element}} * \text{Vector length} + \text{Start-up time}_{\text{ADDV}} + \text{stall time} + \text{Start-up time}_{\text{MULTV}}$$
$$= 2 * \text{Vector length} + 6 + 4 + 7$$
$$= 2 * \text{Vector length} + 17$$

Because of the dependence, the MULTV must complete before the ADDV can begin. However, if the vector register, V1 in this case, is treated not as a single entity but as a group of individual registers, then the pipelining concept of forwarding can be extended to work on individual elements of a vector. This idea, which will allow the ADDV to start earlier in this example, is called *chaining*. Chaining allows a vector operation to start as soon as the individual elements of its vector source operand become available: The results from the first functional unit in the chain are forwarded to the second functional unit. (Of course, they must be different units to avoid using the same unit twice per clock!) In a chained sequence the initiation rate is equal to one per clock cycle if the functional units in the chained operations are all fully pipelined. Even though the operations depend on one another, chaining allows the operations to proceed in parallel on separate elements of the vector. A sustained rate (ignoring start-up) of two floating-point operations per clock cycle can be achieved, even though the operations are dependent!

The total running time for the above sequence becomes

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

Figure 7.14 shows the timing of a chained and an unchained version of the above pair of vector instructions with a vector length of 64. In Figure 7.14, the total time for chained operation is 77 clock cycles. With 128 floating-point operations done in that time, 1.7 FLOPs per clock cycle are obtained, versus a total time of 145 clock cycles or 0.9 FLOPs per clock cycle for the unchained version.

We will see in Section 7.7 that chaining plays a major role in boosting vector performance.

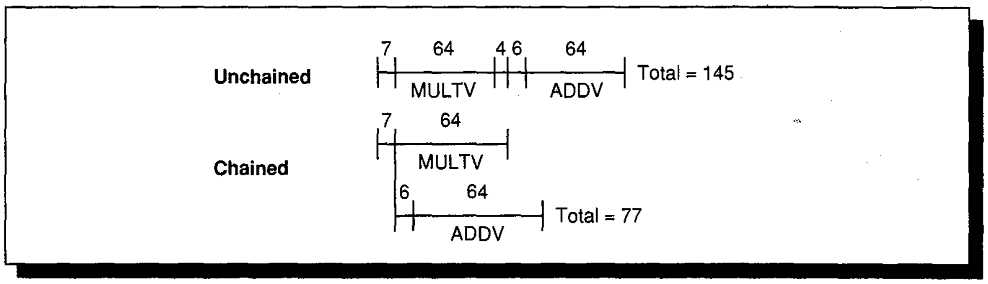

FIGURE 7.14 Timings for a sequence of dependent vector operations ADDV and MULTV, both unchained and chained. The 4–clock-cycle delay comes from a stall for dependence, described earlier; the 6– and 7–clock-cycle delays are the latency of the adder and multiplier.

Conditionally Executed Statements and Sparse Matrices

In the last section, we saw that many programs only achieved low to moderate levels of vectorization. Because of Amdahl's Law, the speedup on such programs will be very limited. Two reasons why higher levels of vectorization are not achieved are the presence of conditionals (if statements) inside loops and the use of sparse matrices. Programs that contain if statements in loops cannot be run in vector mode using the techniques we have discussed so far because the if statements introduce control flow into a loop. Likewise, sparse matrices cannot be efficiently implemented using any of the capabilities we have seen so far; this is a major factor in the lack of vectorization for Spice. This section discusses techniques that allow programs with these structures to execute in vector mode. Let's start with conditional execution.

Consider the following loop:

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

This loop cannot normally be vectorized because of the conditional execution of the body. However, if the inner loop could be run for the iterations for which A(i) ≠ 0, then the subtraction could be vectorized.

Vector-mask control helps us do this. The *vector-mask control* takes a Boolean vector of length MVL. When the *vector-mask register* is loaded with the result of a vector test, any vector instructions to be 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. Clearing the vector-mask register sets it to all 1s, making subsequent vector instructions operate on all vector elements. The following code can now be used for the above loop, assuming that the starting addresses of A and B are in Ra and Rb respectively:

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

Most modern vector machines provide vector-mask control. The vector-mask capability described here is available on some machines, but others allow the use of the vector mask with only a small number of instructions.

Using a vector-mask register does, however, have disadvantages. First, execution time is not decreased, even though some elements in the vector are not operated on. Second, in some vector machines the vector mask serves only to disable the storing of the result into the destination register, and the actual operation still occurs. Thus, if the operation in the above example were a divide rather than a subtract and the test was on B rather than A, false floating-point exceptions might result since the operation was actually done. Machines that mask the operation as well as the result store avoid this problem.

Now, let's turn to sparse matrices; later we will show another method for handling conditional execution. We have dealt with vectors in which the elements are separated by a constant stride. If an application called for a sparse matrix, we might see code that looks like:

```
        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 to 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 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.

The primary mechanism for supporting sparse matrices is scatter-gather operations using index vectors. 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 appeared on the CDC STAR-100. The instructions LVI (Load Vector Indexed) and SVI (Store Vector Indexed) provide these operations in DLXV. For example, assuming that Ra, Rc, Rk, and Rm contain the starting addresses of the vectors in the above sequence, the inner loop of the sequence can be coded with vector instructions such as:

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

This technique allows code with sparse matrices to be run in vector mode. The source code above would **never** be automatically vectorized by a compiler because the compiler cannot know that the elements of K are distinct values, and thus that no dependences exist. Instead, a programmer directive would tell the compiler that it could run the loop in vector mode.

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

The indexed loads/stores and the CVI instruction provide an alternative method to support conditional execution. Here is a vector sequence that implements the loop we saw on page 379:

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

Whether the implementation using scatter/gather is better than the conditionally executed version depends on the frequency with which the condition holds and the cost of the operations. Ignoring chaining, the running time of the first version (on page 379) is $5n + c_1$. The running time of the second version using indexed loads and stores with a running time of one element per clock is $4n + 4*f*n + c_2$, where f is the fraction of elements for which the condition is true (i.e., $A \neq 0$). If we assume that the values of c_1 and c_2 are comparable, or that they are much smaller than n, we can find when this second technique is better.

$$\text{Time}_1 = 5n$$

$$\text{Time}_2 = 4n + 4*f*n$$

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

$$5n \geq 4n + 4*f*n$$

$$\frac{1}{4} \geq f$$

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

Vector Reduction

As we saw in Section 7.5, 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.*

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. Many vector machines provide hardware assist for doing reductions, as we will see next.

Example Show how the FORTRAN code would look for execution of the second loop in the code fragment above using recursive doubling.

Answer Here is the code:

```
            len = 32
            do 100 j=1,6
                  do 10 i=1,len
      10                dot(i) = dot(i) + dot(i+len)
                  len = len / 2
      100   continue
```

When the loop is done, the sum is in dot(1).

In some vector machines, the vector registers are addressable, and another technique, sometimes called partial sums, can be used. This is discussed in Exercise 7.12. 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.

7.7 Putting It All Together: Evaluating the Performance of Vector Processors

In this section we look at different measures of performance for vector machines and what they tell us about the machine. To determine the performance of a machine 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 machine on a loop is to give the execution time of the vector loop. For vector loops people often give the MFLOPS (Millions 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 2, Section 2.2, page 35 for an extensive discussion on MFLOPS). In any event, either measurement should include the overhead.

In this section we examine the performance of DLXV on our SAXPY 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 7.4. At the same time, we will look at different ways to measure performance using the computed time. The constant values for T_{loop} and T_{base} 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 machine, length-related measures are often applied in addition to time and MFLOPs. These length-related measures tend to vary dramatically across different machines and are interesting to compare. (Remember, though, that **time** is always the measure of interest when comparing the relative speed of two machines.) 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. (R_n is the MFLOPS rate for a vector of length n.)

$N_{1/2}$—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 SAXPY problem running on DLXV. When chained, the inner loop of the SAXPY code looks like this (assuming that Rx and Ry hold starting addresses):

```
LV      V1,Rx       ;load the vector X
MULTSV  V2,S1,V1    ;vector*scalar-chained to LV X
LV      V3,Ry       ;vector load Y
ADDV    V4,V2,V3    ;sum aX + Y, chained to LV Y
SV      Ry,V4       ;store the vector Y
```

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

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

Since there are three memory references and only one memory pipeline, the value of $T_{element}$ must be at least 3, and chaining allows it to be exactly 3. If $T_{element}$ were a complete indication of performance, the loop would run at a MFLOPS rate of $\frac{2}{3}$ * clock rate (since there are 2 FLOPS per iteration). Thus, based only on the $T_{element}$ time, an 80-MHz DLXV would run this loop at 53 MFLOPS. But the Linpack benchmark, whose core is this computation, runs at only 13 MFLOPS (without some sophisticated compiler optimization we discuss in the Exercises) on an 80-MHz CRAY-1, DLXV's cousin! Let's see what accounts for the difference.

The Peak Performance of DLXV on SAXPY

First, we should determine what the peak performance, R_∞, really is, since we know it differs from the ideal 53-MFLOPS rate. Figure 7.15 shows the timing within each block of strip-mined code.

Operation	Starts at clock number	Completes at clock number	Comment
`LV V1,Rx`	0	$12 + 64 = 76$	Simple latency
`MULTV a,V1`	$12 + 1 = 13$	$13 + 7 + 64 = 84$	Chained to `LV`
`LV V2,Ry`	$76 + 1 = 77$	$77 + 12 + 64 = 153$	Starts after first `LV` done (memory contention)
`ADDV V3,V1,V2`	$77 + 1 + 12 = 90$	$90 + 6 + 64 = 160$	Chained to `MULTV` and `LV`
`SV Ry,V3`	$160 + 1 + 4 = 165$	$165 + 12 + 64 = 241$	Must wait on `ADDV`; not chained (memory contention)

FIGURE 7.15 The SAXPY loop when chained in DLXV. There are three distinct types of delays: 4–clock-cycle delays when a nonchained dependence occurs, latency delays that occur when waiting for a result for the pipeline (6 for add, 7 for multiply, and 12 for memory access), and delays due to contention for the memory pipeline. The last cause is what makes the time per element at least 3 clocks.

From the data in Figure 7.15 and the value of T_{element}, we know that

$$T_{\text{start}} = 241 - 64 * T_{\text{element}} = 241 - 192 = 49$$

This value is equal to the sum of the latencies of the functional units: 12 + 7 + 12 + 6 + 12 = 49.

Using MVL = 64, $T_{\text{loop}} = 15$, $T_{\text{base}} = 10$, and $T_{\text{element}} = 3$ in the performance equation, the time for an n-element operation is

$$T_n = 10 + \left\lceil \frac{n}{64} \right\rceil * (15 + 49) + 3n$$

$$T_n = 10 + n + 64 + 3n = 4n + 74$$

The sustained rate is actually over 4 clock cycles per iteration, rather than the theoretical rate of 3 clocks per iteration, which ignores overhead. The major part of the difference is the cost of the overhead for each block of 64 elements. The basic start-up overhead, T_{base}, adds only $\frac{10}{n}$ to the time for each element. This overhead disappears with long vectors.

We can now compute R_∞ for an 80-MHz clock as

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

The numerator is independent of n, hence

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

$$\lim_{n\to\infty} (\text{Clock cycles per iteration}) = \lim_{n\to\infty} \left(\frac{T_n}{n}\right) = \lim_{n\to\infty} \left(\frac{4n+74}{n}\right) = 4$$

$$R_\infty = \frac{2 * 80\text{ MHz}}{4} = 40\text{ MFLOPS}$$

Sustained Performance of Linpack on DLXV

The Linpack benchmark is a Gaussian elimination on a 100x100 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 SAXPY using a vector length of 66.

$$T_{66} = 10 + 2 * (15 + 49) + 66 * 3 = 10 + 128 + 198 = 336$$

$$R_{66} = \frac{2 * 66 * 80}{336}\text{MFLOPS} = 31.4\text{ MFLOPS}$$

In reality, Linpack does not spend all its time in the inner loop. The benchmark's actual performance can be found by taking the weighted harmonic mean of the MFLOPS ratings inside the inner loop (31.4 MFLOPS) and outside that loop (about 0.5 MFLOPS). We can compute the weighting factors by knowing the percentage of the time inside the inner loop after vectorization.

The percentage in the inner loop after vectorization can be obtained using Amdahl's Law if we know the percentage in scalar and the speedup from vectorization. In scalar mode, about 75% of the execution time is spent in the inner loop, and the speedup from vectorization is about 5 times. With this information the percentage of time in the inner loop after vectorization can be computed:

$$\text{Total relative time after vectorization} = \frac{0.75}{5} + 0.25$$

$$= 0.15 + 0.25 = 0.40$$

$$\text{Percentage of time in inner loop after vectorization} = \frac{0.15}{0.40} = 37.5\%$$

The remaining 62.5% of the time is spent outside the main loop. Thus, the overall MFLOPS rating is

$$\text{Percentage}_{\text{inner}} * \text{MFLOPS}_{\text{inner}} + \text{Percentage}_{\text{other}} * \text{MFLOPS}_{\text{other}}$$

$$= 37.5\% * 31.4 + 62.5\% * 0.5 = 12.1 \text{ MFLOPS}$$

This is comparable to the rate at which the CRAY-1 runs this benchmark.

Example What is $N_{1/2}$ for just the inner loop of SAXPY for DLXV with an 80-MHz clock?

Answer Using R_∞ as the peak rate, we want to know the vector length that will achieve about 20 MFLOPS. So,

$$\frac{\text{Clock cycles}}{\text{Iteration}} = \frac{\frac{\text{FLOPS}}{\text{Iteration}} * \frac{\text{Clocks}}{\text{Second}}}{\frac{\text{FLOPS}}{\text{Second}}}$$

$$= \frac{2 * 80 \text{ MHz}}{20 \text{ MFLOPS}} = 8$$

Hence, a rate of 20 MFLOPS means that a loop iteration completes every 8 clock cycles on average, or that $\frac{T_n}{n} = 8$. Using our equation and assuming that $n \le 64$,

$$T_n = 10 + 1 * 64 + 3 * n$$

Substituting for T_n in the first equation, we obtain

$$\begin{aligned} 8n &= 74 + 3 * n \\ 5n &= 74 \\ n &= 14.8 \end{aligned}$$

So $N_{1/2} = 15$; that is, a vector of length 15 gives approximately one-half the peak performance for the SAXPY loop on DLXV.

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

Answer

Again, we know that $N_v < 64$. The time to do one iteration in scalar mode can be estimated as 10 + 12 + 12 + 7 + 6 = 47 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 $T_n = 74 + 3*n$ clock cycles for a vector of length ≤ 64. Therefore,

$$\begin{aligned} 74 + 3n &= 47n \\ n &= \frac{74}{44} \\ N_v &= 2 \end{aligned}$$

For the SAXPY 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).

SAXPY Performance on an Enhanced DLXV

SAXPY, 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 the CRAY-1. The CRAY X-MP has three memory pipelines, compared to 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 SAXPY on DLXV if we added two more memory pipelines?

Answer

Figure 7.16 is a version of Figure 7.15 (page 385), adjusted for multiple memory pipelines.

Operation	Starts at clock number	Completes at clock number	Comment
`LV V1,Rx`	0	12 + 64 = 76	Simple latency
`MULTV a,V1`	12 + 1 = 13	13 + 7 + 64 = 84	Chained to `LV`
`LV V2,Ry`	2	2 + 12 + 64 = 78	Starts immediately
`ADDV V3,V1,V2`	13 + 1 + 7 = 21	21 + 6 + 64 = 91	Chained to `MULTV` and `LV`
`SV Ry,V3`	21 + 1 + 6 = 28	28 + 12 + 64 = 104	Chained to `ADDV`

FIGURE 7.16 The SAXPY loop when chained in DLXV with three memory pipelines. The only delays are latency delays that occur when waiting for a result for the pipeline (6 for add, 7 for multiply, and 12 for each memory access).

With three memory pipelines, the performance is greatly improved. Here's our standard performance equation:

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

With three memory pipelines the value of T_{element} becomes 1, so that

$$T_{\text{start}} = 104 - 64 * T_{\text{element}} = 104 - 64 = 40$$

The reduction in stalls reduces the start-up penalty for each sequence. The values of T_{loop} and T_{base}, 15 and 10, remain the same. Therefore, for an average vector length of 66, we have:

$$T_{66} = T_{\text{base}} + \left\lceil \frac{66}{64} \right\rceil * (T_{\text{loop}} + T_{\text{start}}) + 66 * T_{\text{element}}$$

$$T_{66} = 10 + 2 * (15 + 40) + 66 * 1 = 186$$

With three memory pipelines, we have reduced the clock-cycle count for sustained performance from 336 to 186, a factor of 1.8. Note the effect of Amdahl's Law: We improved the theoretical peak rate, as measured by T_{element}, by a factor of 3, but only achieved an overall improvement of a factor of 1.8 in sustained performance. Because the speedup outside the inner loop is likely to be less than 1.8, the overall improvement in run time for the benchmark will also be less.

Another improvement could come from allowing the start-up of one loop iteration before another completes. This requires that one vector operation be allowed to begin using a functional unit, before another operation has completed. This complicates the instruction issue logic substantially, but has the advantage that the start-up overhead will only occur once, independent of the vector length. On a long vector the overhead per block ($T_{\text{loop}} + T_{\text{start}}$) can be completely amortized. In this way a machine 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 SAXPY on DLXV if we added two more memory pipelines and allowed the strip-mining and start-up overhead to be fully overlapped?

Answer

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

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

Since $T_n = n + 40 + 10 + 15 = n + 65$,

$$\lim_{n \to \infty} \left(\frac{T_n}{n}\right) = \lim_{n \to \infty} \left(\frac{n + 65}{n}\right) = 1$$

$$R_\infty = \frac{2 * 80 \text{ MHz}}{1} = 160 \text{ MFLOPS}$$

Thus, adding the extra memory pipelines and more flexible issue logic yields an improvement in peak performance of a factor of 4. However, $T_{66} = 131$, so for shorter vectors, the sustained performance improvement is about 40%.

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

$$\frac{\text{Number of FLOPS per iteration} * \text{Clock rate}}{T_{\text{element}}}$$

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:

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

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 machines are useful in understanding the wide range of performance that applications may see on a vector machine.

7.8 Fallacies and Pitfalls

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

Early vector machines such as the TI ASC and the CDC STAR-100 had long start-up times. For some vector problems, N_v could be greater than 100! Today, the Japanese supercomputers often have higher sustained rates than the Cray Research machines. But with start-up overheads that are 50–100% higher, the faster sustained rates often provide no real advantage. On the CYBER-205 the start-up overhead for SAXPY 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 SAXPY for a vector of length n is therefore roughly $158 + 2n$. If the clock rates of the CRAY-1 and the CYBER-205 were identical, the CRAY-1 would be faster until $n > 64$. Because the CRAY-1 clock is also faster (even though the 205 is newer), the crossover point is over 100. Comparing a four-vector-pipeline CYBER-205 (the maximum-size machine) to the CRAY X-MP that was delivered shortly after the 205, the 205 completes two results per clock cycle—twice as fast as the X-MP. However, vectors must be longer than about 200 for the CYBER-205 to be faster. The problem of start-up overhead has been the major difficulty for the memory–memory vector architectures.

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

This is another area where Seymour Cray rewrote the rules. Many of the early vector machines had comparatively slow scalar units (as well as large start-up overheads). Even today, machines with higher peak vector performance, can be outperformed by a machine 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 machine and a vector machine with lower scalar performance. The Livermore FORTRAN kernels are a collection of 24 scientific kernels with varying degrees of vectorization (see Chapter 2; Section 2.2). Figure 7.17 shows the performance of two different machines on this benchmark. Despite the vector machine's higher peak performance, its low scalar performance makes it slower than a fast scalar machine. The next fallacy is closely related.

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

FIGURE 7.17 Performance measurements for the Livermore FORTRAN kernels on two different machines. 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 machine is more than a factor of 2.5 times faster for a highly vectorizable loop (maximum rate). However, the lower scalar performance of the Stardent-1500 negates the higher vector performance when total performance is measured by the harmonic mean on all 24 loops.

*Fallacy: The scalar performance of the **best** supercomputers is low.*

The supercomputers from Cray Research have always had good scalar performance. Measurements of the CRAY Y-MP running (the nonvectorizable) Spice benchmark show this. When our Spice benchmark is run on the CRAY Y-MP in scalar mode it executes 665 million instructions, with a CPI of 4.1. By comparison, the DECstation 3100 executes 738 million instructions with a CPI of 2.1.

Although the DECstation uses fewer cycles, the Y-MP uses fewer instructions and is much faster overall, since it has a clock cycle one-tenth as long.

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

As we saw with the SAXPY loop, memory bandwidth is quite important. SAXPY 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. Recently, the CRAY-1 performance on Linpack has jumped because the compiler used clever transformations to change the computation so that values could be kept in the vector registers. This lowered the number of memory references per FLOP and improved the performance by nearly a factor of 2! Thus, the memory bandwidth on the CRAY-1 became sufficient for a loop that formerly required more bandwidth.

7.9 Concluding Remarks

In the late 1980s rapid performance increases in efficiently pipelined scalar machines lead to a dramatic closing of the gap between vector supercomputers, costing millions of dollars, and fast, pipelined, VLSI microprocessors costing less than $100,000. The basic reason for this was the rapidly decreasing CPI of the scalar machines.

For scientific programs, an interesting counterpart to CPI is clock cycles per FLOP, or CPF. We saw in this chapter that for vector machines this number was typically in the range of 2 (for a CRAY X-MP style machine) to 4 (for a CRAY-1 style machine). In the last chapter, we saw that the pipelined machine varied from about 6 (for DLX) down to about 2.5 (for a superscalar DLX with no memory system losses running a SAXPY-type loop).

Recent trends in vector machine design have focused on high peak-vector performance and multiprocessing. Meanwhile, high-speed scalar machines concentrate on keeping the ratio of peak to sustained performance near one. Thus, if the peak rates advance comparably, the sustained rates of the scalar machines will advance more quickly, and the scalar machines will continue to close the CPF gap. These multiple-issue scalar machines can rival or exceed the performance of vector machines with comparable clock speeds, especially for levels of vectorization below 70%. Furthermore, the differences in clock rate are largely technology driven—the low-end, microprocessor-based vector machines have clock rates comparable to the pipelined machines using microprocessor technology. (In fact, they often use the same microprocessors!) In the future, we can expect high-speed pipelined scalar machines to be built with clock rates that will rival those of the current vector supercomputers. However, the vector machines should retain a performance advantage for problems with very long vectors that can use multiple memory pipelines and achieve performance close to the peak.

The 1990s will be interesting as the pipelined scalar machines that exploit more instruction-level parallelism and are usually much cheaper (because their peak performance and hence total hardware is much less) begin to offer performance levels for many applications that are difficult to distinguish from those of vector machines.

7.10 Historical Perspective and References

The first vector machines 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 machines. They had relatively slow scalar units—the STAR used the same units for scalars and vectors—making the scalar pipeline extremely deep. Both machines 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 machines.

Cray, who worked on the 6600 and the 7600 at CDC, founded Cray Research and introduced the CRAY-1 in 1976 (see Russell [1978]). The CRAY-1 used a vector-register architecture to significantly lower start-up overhead. He also had efficient support for nonunit stride and invented chaining. Most importantly, the CRAY-1 was also the fastest scalar machine 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 machine primarily for its outstanding scalar performance. Many subsequent vector machines are based on the architecture of this first commercially successful vector machine. Baskett and Keller [1977] is 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 expansibility of the vector unit with up to four vector pipelines, each with multiple functional units and a wide load/store pipe that provided multiple words per clock. The peak performance of the CYBER-205 greatly exceeded the performance of the CRAY-1. However, on real programs, the performance difference was much smaller.

The CDC STAR machine and its descendant, the CYBER-205, were memory–memory vector machines. To keep the hardware simple and support the high bandwidth requirements (up to 3 memory references per FLOP), these machines did not efficiently handle nonunit stride. While most loops have unit stride, a nonunit stride loop had poor performance on these machines because memory-to-memory data movements were required to gather together (and scatter back) the nonadjacent vector elements.

Schneck [1987] described several of the early pipelined machines (e.g., Stretch) through the first vector machines including the 205 and CRAY-1. Dongarra [1986] did another good survey, focusing on more recent machines.

In 1983, Cray shipped the first CRAY X-MP (see Chen [1983]). With an improved clock rate (9.5 ns versus 12.5 on the CRAY-1), better chaining support, and multiple memory pipelines, this machine maintained the Cray Research lead in supercomputers. The CRAY-2, a completely new design configurable with up to four processors, was introduced later. It has a much faster clock than the X-MP, but also much deeper pipelines. The CRAY-2 lacks chaining, has an enormous memory latency, and has only one memory pipe per processor. In general, it is only faster than the CRAY X-MP on problems that require its very large main memory.

In 1983, the Japanese computer vendors entered the supercomputer marketplace, starting with the Fujitsu VP100 and VP200 (Miura and Uchida [1983]), and later expanding to include the Hitachi S810, and the NEC SX/2 (see Watanabe [1987]). These machines have proved to be close to the CRAY X-MP in performance. In general, these three machines have much higher peak performance than the CRAY X-MP, though because of large start-up overhead, their typical performance is often lower than the CRAY X-MP (see Figure 2.24 in Chapter 2). The CRAY X-MP favored a multiple-processor approach, first offering a two-processor version and later a four-processor machine. In contrast, the three Japanese machines 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 8 processors and lowers the cycle time to 6 ns. With a full complement of 8 processors, the Y-MP is 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 machines available in the early 1990s. Seymour Cray continues to head the spin-off, which is now called Cray Computer Corporation.

In the early 1980s, CDC spun out a group, called ETA, to build a new supercomputer, the ETA-10, capable of 10 GigaFLOPs. The ETA machine delivered in the late 1980s (see Fazio [1987]) 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 machines, the 3090/VF routes its vectors through the cache.

The 1980s also saw the arrival of smaller-scale vector machines, called mini-supercomputers. Priced at roughly one-tenth the cost of a supercomputer ($0.5 to

$1 million versus $5 to $10 million), these machines caught on quickly. Although many companies joined the market, the two companies that have been most successful are Convex and Alliant. Convex started with a uniprocessor vector machine (C-1) and now offers a small multiprocessor (C-2); they emphasize Cray software capability. Alliant [1987] has concentrated more on the multiprocessor aspects; they build an eight-processor machine, with each processor offering vector capability.

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 Wolf [1986] gave a good overview of vectorizing compiler technology.

Benchmark studies of various supercomputers including attempts to understand the performance differences have been undertaken by Lubeck, Moore and Mendez [1985], Bucher [1983], and Jordan [1987]. In Chapter 2, we discussed several benchmark suites aimed at scientific usage and often employed for supercomputer benchmarking, including Linpack, the Lawrence Livermore Laboratories FORTRAN kernels, and the Perfect Club suite.

In the late 1980s, graphics supercomputers arrived on the market from Stellar [Sporer, Moss, and Mathais 1988] and Ardent [Miranker, Rubenstein, and Sanguinetti 1988]. The Stellar machine used a timeshared pipeline to allow high-speed vector processing and efficient multitasking. This approach was used earlier in a machine designed by B. J. Smith [1981] called the HEP and built by Denelcor in the mid-1980s. This approach does not yield high-speed scalar performance, as evident in the scalar benchmarks of the Stellar machine. The Ardent machine combines a RISC processor (the MIPS R2000) with a custom vector unit. These vector machines, which cost about $100K, brought vector capabilities to a new potential market. In late 1989, Stellar and Ardent were merged to form Stardent, and the Ardent architecture is being shipped from the combined company.

From this overview we can see the progress vector machines have made. In less than 20 years they 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.

References

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

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

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

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

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

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

DONGARRA, J. J. [1986]. "A survey of high performance computers," *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 computers: 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, 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., "Supercomputers: 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, Pa., 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 computer system," *Comm. of the ACM* 21:1 (January) 63–72.

SCHNECK, P. B. [1987]. *Supercomputer 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) 464–467.

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 computer architecture," *Proc. AFIPS Fall Joint Computer Conf.,* 221–228.

E X E R C I S E S

In these Exercises assume DLXV has a clock rate of 80 MHz and that $T_{base} = 10$ and $T_{loop} = 15$. Also assume that the store latency is always included in the running time.

7.1 [10] <7.1–7.2> Write a DLXV vector sequence that achieves the peak MFLOPS performance of the machine (use the functional unit and instruction description in Section 7.2). Assuming an 80-MHz clock rate, what is the peak MFLOPS?

7.2 [20/15/15] <7.1–7.6> Consider the following vector code run on an 80-MHz version of DLXV for a fixed vector length of 64:

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

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

a. [20] Assuming no chaining and a single memory pipeline, how many clock cycles per result (including both stores as one result) does this vector sequence require?

b. [15] If the vector sequence is chained, how many clock cycles per result does this sequence require?

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

7.3 [20/20/15/15/20/20/20] <7.2–7.7> 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 7.7 to estimate performance throughout this exercise assuming an 80-MHz version of DLXV.

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

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

c. [15] Find R_{∞} for this loop.

d. [15] Find $N_{1/2}$ for this loop.

e. [20] 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 3 cycles. Assume the scalar overhead is also T_{loop}.

f. [20] Assume DLXV has two memory pipelines. Write vector code that takes advantage of the second memory pipeline.

g. [20] Compute T_{100} and R_{100} for DLX with two memory pipelines.

7.4 [20/10] <7.3> Suppose we have a version of DLXV with eight memory banks (each a doubleword wide) and a memory-access time of eight cycles.

a. [20] If a load vector of length 64 is executed with a stride of 20 doublewords, how many cycles will the load take to complete?

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

7.5 [12/12/20] <7.4–7.7> 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] 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] Write the DLXV vector code for the loop without a dependence.

c. [20] Write the DLXV code to evaluate the dependent loop using recursive doubling.

7.6 [20/15/20/20] <7.5–7.7> 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 "SAXPY–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] Write the **straightforward** code sequence for just the inner loop in DLXV vector instructions.

b. [15] Using the techniques of Section 7.7, estimate the performance of this code on DLXV by finding T_{64} in clock cycles. You may assume that T_{base} applies once and T_{loop} of overhead is incurred for each iteration of the outer loop. What limits the performance?

c. [20] Rewrite the DLXV code to reduce the performance limitation; show the resulting inner loop in DLXV vector instructions. (Hint: think about what establishes $T_{element}$; can you affect it?) Find the total time for the resulting sequence.

d. [20] Estimate the performance of your new version using the techniques of Section 7.7 and finding T_{64}.

7.7 [15/15/25] <7.6> Consider the following code.

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

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

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

b. [15] Write the DLXV code for this loop using scatter/gather.

c. [25] 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 was typical?

7.8 [15/20/15/15] <7.1–7.7> In Figure 2.24 of Chapter 2 (page 75), 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 the CRAY X-MP, while for another more realistic problem the CRAY X-MP was twice as fast as the Hitachi machine. Let's examine why this might occur using two versions of DLXV and the following code sequences:

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

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

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

a. [15] Find the number of clock cycles for code sequence 1 on DLXV.

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

c. [15] Find the number of clock cycles on code sequence 2 for DLXV.

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

7.9 [15/15/20] <7.5> In this problem we will examine some of the vector loop tests discussed in Section 7.5 and summarized in Figure 7.13 (page 377).

a. [15] Here is a simple code fragment:

```
          do 400 i = 2,100,2
             a(i-1) = a(50*i+1)
   400    continue
```

To use the GCD test this loop must first be "normalized"—written so that the index starts at 1 and increments by 1 on every iteration. Write a normalized version of the loop (change the indices as needed), then use the GCD test to see if it vectorizes.

b. [15] Here is another loop:

```
          do 400 i = 2,100,2
             a(i) = a(i-1)
   400    continue
```

Normalize the loop and use the GCD test to detect a dependence. Is there a real dependence in this loop?

c. [20] Here is a tricky piece of code with two-dimensional arrays. Can it be vectorized? 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
```

7.10 [25] <7.5> Show that if for two array elements A(a*i +b) and A(c*i+d) there is a true dependence, then GCD(c,a) divides (d–b).

7.11 [12/15] <7.5> Consider the following loop:

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

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

b. [15] Rewrite the code in Fortran so that it can be vectorized as two separate vector sequences.

7.12 [25] <7.6> Because the difference between vector and scalar modes is so large on a supercomputer and the machines often cost tens of millions of dollars, programmers are frequently willing to go to extraordinary effort to achieve good performance. This often includes tricky assembly language programming. An interesting problem is to write a vectorizable sort for floating-point numbers—a task sometimes required in scientific code. Choose a sorting algorithm and write a version for DLXV that uses vector operations as much as possible. (Hint: One good choice is quicksort where the vector compares and compress/expand capability can be used.)

7.13 [25] <7.6> In some vector machines, 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 shown on page 382. The key idea in partial sums is to reduce the vector to *m* sums where *m* is the total latency through the vector functional unit including the operand read and write times. Assume that the DLXV vector registers are addressable (e.g., you can initiate a vector operation with the operand V1(16), indicating that the input operand began with element 16). Also, assume that the total latency for adds including operand read and write is eight cycles. Write a DLXV code sequence that reduces the contents of V1 to eight partial sums. It can be done with one vector operation.

7.14 [40] <7.2–7.6> Extend the DLX simulator to be a DLXV simulator including the ability to count clock cycles. Write some short benchmark programs in DLX and DLXV assembly language. Measure the speedup on DLXV, the percentage of vectorization, and usage of the functional units.

7.15 [50] <7.5> Modify the DLX compiler to include a dependence checker. Run some scientific code and loops through it and measure what percentage of the statements could be vectorized.

7.16 [Discussion] Some proponents of vector machines might argue that the vector processors have provided the best path to ever-increasing amounts of computer 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?

7.17 [Discussion] Consider the points raised in the Concluding Remarks (Section 7.9). This topic—the relative advantages of pipelined scalar machines versus FP vector machines—is the source of much debate in the early 1990s. What advantages do you see for each side? What would you do in this situation?

Ideally one would desire an indefinitely large memory capacity such that any particular . . . word would be immediately available. . . . We are . . . forced to recognize the possibility of constructing a hierarchy of memories, each of which has greater capacity than the preceding but which is less quickly accessible.

A. W. Burks, H. H. Goldstine, and J. von Neumann,
Preliminary Discussion of the Logical Design of an Electronic Computing Instrument (1946)

8 Memory-Hierarchy Design

8.1 Introduction: Principle of Locality

Computer pioneers correctly predicted that programmers would want unlimited amounts of fast memory. As the 90/10 rule in the first chapter predicts, most programs fortunately do not access all code or data uniformly (see Section 1.3, pages 8–12). The 90/10 rule can be restated as the *principle of locality*. This hypothesis, which holds that all programs favor a portion of their address space at any instant of time, has two dimensions:

- *Temporal locality* (locality in time)—If an item is referenced, it will tend to be referenced again soon.
- *Spatial locality* (locality in space)—If an item is referenced, nearby items will tend to be referenced soon.

A memory hierarchy is a natural reaction to locality and technology. The principle of locality and the guideline that smaller hardware is faster yield the concept of a hierarchy based on different speeds and sizes. Since slower memory is cheaper, a memory hierarchy is organized into several levels—each smaller, faster, and more expensive per byte than the level below. The levels of the hierarchy subset one another; all data in one level is also found in the level below, and all data in that lower level is found in the one below it, and so on until we reach the bottom of the hierarchy.

This chapter includes a half-dozen examples that demonstrate how taking advantage of the principle of locality can improve performance. All these strategies map addresses from a larger memory to a smaller but faster memory. As part of address mapping, the memory hierarchy is usually given the responsibility of address checking; protection schemes used for doing this are covered in this chapter. Later we will explore advanced memory hierarchy topics and trace a memory access through three levels of memory on the VAX-11/780.

8.2 General Principles of Memory Hierarchy

Before proceeding with examples of the memory hierarchy, let's define some general terms applicable to all memory hierarchies. A memory hierarchy normally consists of many levels, but it is managed between two adjacent levels at a time. The *upper* level—the one closer to the processor—is smaller and faster than the *lower* level (see Figure 8.1). The minimum unit of information that can be either present or not present in the two-level hierarchy is called a *block.* The size of a block may be either fixed or variable. If it is fixed, the memory size is a multiple of that block size. Most of this chapter will be concerned with fixed block sizes, although a variable block design is discussed in Section 8.6.

Success or failure of an access to the upper level is designated as a hit or a miss: A *hit* is a memory access found in the upper level, while a *miss* means it is not found in that level. *Hit rate,* or hit ratio—like a batting average—is the fraction of memory accesses found in the upper level. This is sometimes represented as a percentage. *Miss rate* (1.0 – hit rate) is the fraction of memory accesses not found in the upper level.

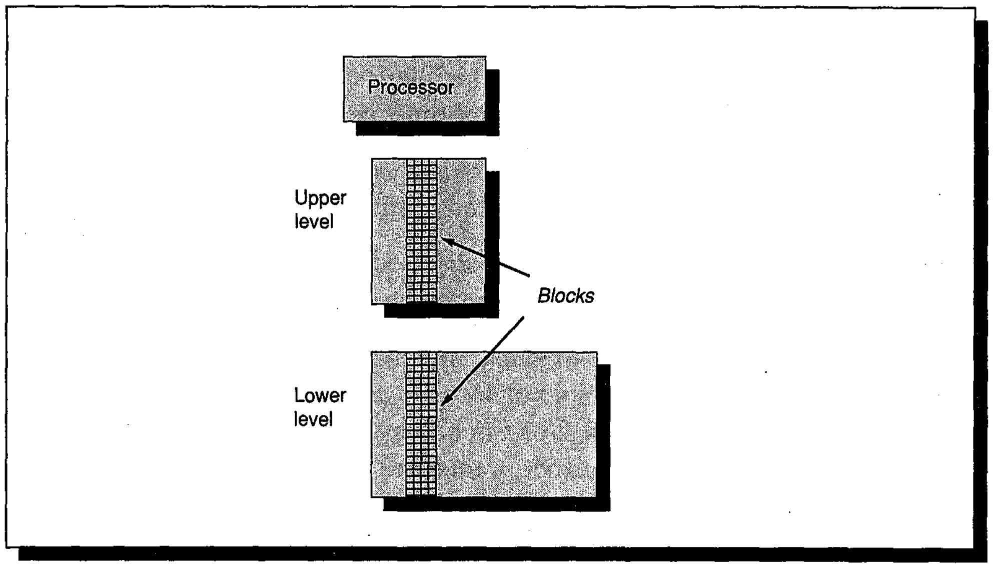

FIGURE 8.1 Every pair of levels in the memory hierarchy can be thought of as having an upper and lower level. Within each level the unit of information that is present or not is called a *block.*

Since performance is the major reason for having a memory hierarchy, the speed of hits and misses is important. *Hit time* is the time to access the upper level of the memory hierarchy, which includes the time to determine whether the access is a hit or a miss. *Miss penalty* is the time to replace a block in the upper level with the corresponding block from the lower level, plus the time to deliver this block to the requesting device (normally the CPU). The miss penalty is further divided into two components: *access time*—the time to access the first word of a block on a miss; and *transfer time*—the additional time to transfer the remaining words in the block. Access time is related to the latency of the lower-level memory, while transfer time is related to the bandwidth between the lower-level and upper-level memories. (Sometimes *access latency* is used to mean access time.)

The memory address is divided into pieces that access each part of the hierarchy. The *block-frame address* is the higher-order piece of the address that identifies a block at that level of the hierarchy (see Figure 8.2). The *block-offset address* is the lower-order piece of the address and identifies an item within a block. The size of the block-offset address is $\log_2$ (size of block); the size of the block-frame address is then the size of the full address at this level less the size of the block-offset address.

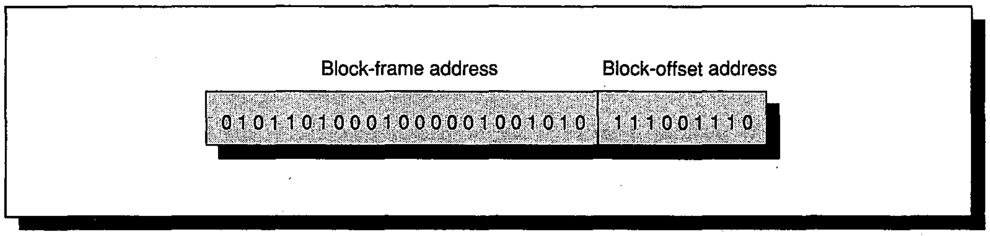

FIGURE 8.2 Example of the frame address and offset address portions of a 32-bit lower-level memory address. In this case the block size is 512, making the size of the offset address 9 bits and the size of the block-frame address 23 bits.

Evaluating Performance of a Memory Hierarchy

Because instruction count is independent of the hardware, it is tempting to evaluate CPU performance using that number. As we saw in Chapters 2 and 4, however, such indirect performance measures have waylaid many a computer designer. The corresponding temptation for evaluating memory-hierarchy performance is to concentrate on miss rate, for it, too, is independent of the speed of the hardware. As we shall see, miss rate can be just as misleading as instruction count. A better measure of memory-hierarchy performance is the average time to access memory:

$$\text{Average memory-access time} = \text{Hit time} + \text{Miss rate} * \text{Miss penalty}$$

The components of average access time can be measured either in absolute time—say, 10 nanoseconds on a hit—or in the number of clock cycles that the

CPU waits for the memory—such as a miss penalty of 12 clock cycles. Remember that average memory-access time is still an indirect measure of performance; so while it is a better measure than miss rate, it is not a substitute for execution time.

The relationship of block size to miss penalty and miss rate is shown abstractly in Figure 8.3. These representations assume that the size of the upper-level memory does not change. The access-time portion of the miss penalty is not affected by block size, but the transfer time does increase with block size. If access time is large, initially there will be little additional miss penalty relative to access time as block size increases. However, increasing block size means fewer blocks in the upper-level memory. Increasing block size lowers the miss rate until the reduced misses of larger blocks (spatial locality) are outweighed by the increased misses as the number of blocks shrinks (temporal locality).

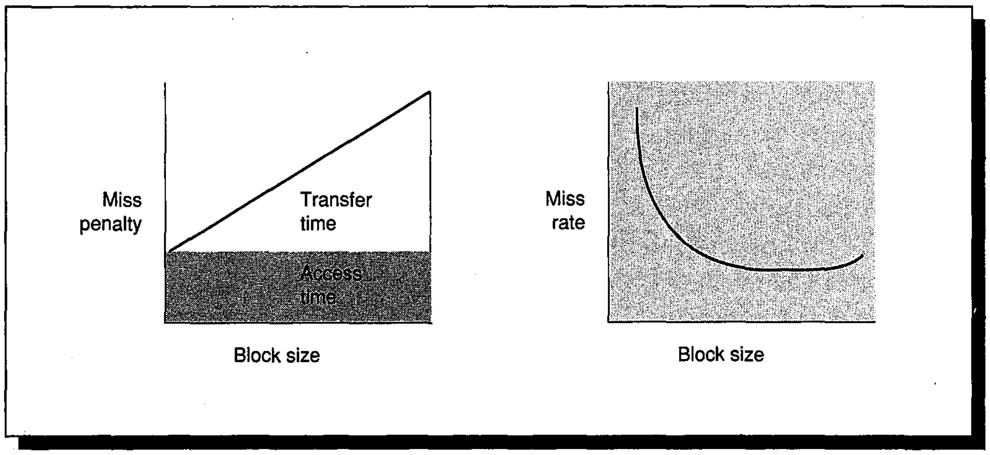

FIGURE 8.3 Block size versus miss penalty and miss rate. The transfer-time portion of the miss penalty obviously grows with increasing block size. For a fixed-size upper-level memory, miss rates fall with increasing block size until so much of the block is not used that it displaces useful information in the upper level, and miss rates begin to rise. The point on the curve on the right where miss rates begin to rise with increasing block size is sometimes called the *pollution point.*

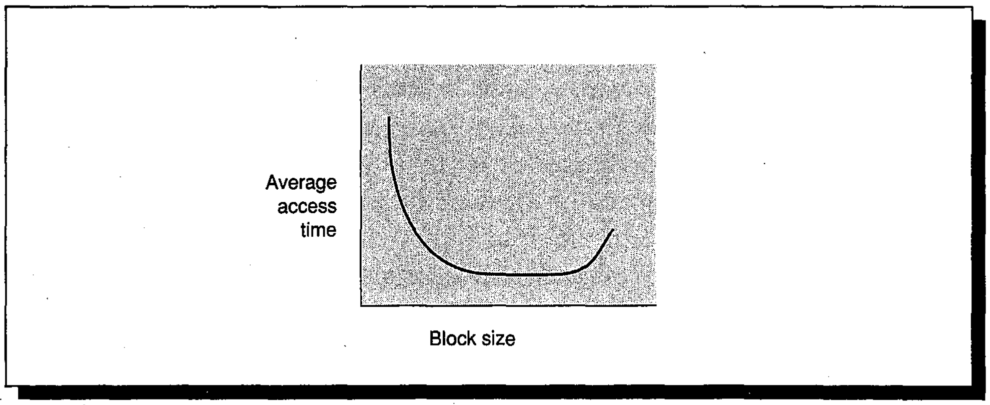

FIGURE 8.4 The relationship between average memory-access time and block size.

The goal of a memory hierarchy is to reduce execution time, not misses. Hence, computer designers favor a block size with the lowest average access time rather than the lowest miss rate. This is related to the product of miss rate and miss penalty, as Figure 8.4 shows abstractly. Of course, overall CPU performance is the ultimate performance test, so care must be taken when reducing average memory-access time to be sure that changes to clock cycle time and CPI improve overall performance as well as average memory-access time.

Implications of a Memory Hierarchy to the CPU

Processors designed without a memory hierarchy are simpler because memory accesses always take the same amount of time. Misses in a memory hierarchy mean that the CPU must be able to handle variable memory-access times. If the miss penalty is on the order of tens of clock cycles, the processor normally waits for the memory transfer to complete. On the other hand, if the miss penalty is thousands of processor clock cycles, it is too wasteful to let the CPU sit idle; in this case, the CPU is interrupted and used for another process during the miss handling. Thus, avoiding the overhead of a long miss penalty means any memory access can result in a CPU interrupt. This also means the CPU must be able to recover any memory address that can cause such an interrupt, so that the system can know what to transfer to satisfy the miss (see Section 5.6). When the memory transfer is complete, the original process is restored, and the instruction that missed is retried.

The processor must also have some mechanism to determine whether or not information is in the top level of the memory hierarchy. This check happens on every memory access and affects hit time; maintaining acceptable performance usually requires the check to be implemented in hardware. The final implication of a memory hierarchy is that the computer must have a mechanism to transfer blocks between upper- and lower-level memory. If the block transfer is tens of clock cycles, it is controlled by hardware; if it is thousands of clock cycles, it can be controlled by software.

Four Questions for Classifying Memory Hierarchies

The fundamental principles that drive all memory hierarchies allow us to use terms that transcend the levels we are talking about. These same principles allow us to pose four questions about any level of the hierarchy:

Q1: Where can a block be placed in the upper level? (*Block placement*)

Q2: How is a block found if it is in the upper level? (*Block identification*)

Q3: Which block should be replaced on a miss? (*Block replacement*)

Q4: What happens on a write? (*Write strategy*)

These questions will help us gain an understanding of the different tradeoffs demanded by the relationships of memories at different levels of a hierarchy.

8.3 Caches

Cache: a safe place for hiding or storing things.

Webster's New World Dictionary of the American Language, Second College Edition (1976)

Cache is the name first chosen to represent the level of the memory hierarchy between the CPU and main memory, and that is the dominant use of the term. While the concept of caches is younger than the IBM 360 architecture, caches appear today in every class of computer and in some computers more than once. In fact, the word has become so popular that it has replaced "buffer" in many computer-science circles.

The general terms defined in the prior section can be used for caches, although the word *line* is often used instead of block. Figure 8.5 shows the typical range of memory-hierarchy parameters for caches.

Block (line) size	4 – 128 bytes
Hit time	1 – 4 clock cycles (normally 1)
Miss penalty	8 – 32 clock cycles
(Access time)	(6 – 10 clock cycles)
(Transfer time)	(2 – 22 clock cycles)
Miss rate	1% – 20%
Cache size	1 KB – 256 KB

FIGURE 8.5 Typical values of key memory-hierarchy parameters for caches in 1990 workstations and minicomputers.

Now let's examine caches in more detail by answering the four memory-hierarchy questions.

Q1: Where Can a Block Be Placed in a Cache?

Restrictions on where a block is placed create three categories of cache organization:

- If each block has only one place it can appear in the cache, the cache is said to be *direct mapped*. The mapping is usually (block-frame address) modulo (number of blocks in cache).
- If a block can be placed anywhere in the cache, the cache is said to be *fully associative*.

- If a block can be placed in a restricted set of places in the cache, the cache is said to be *set associative*. A *set* is a group of two or more blocks in the cache. A block is first mapped onto a set, and then the block can be placed anywhere within the set. The set is usually chosen by bit selection; that is, (block-frame address) modulo (number of **sets** in cache). If there are *n* blocks in a set, the cache placement is called *n-way set associative*.

The range of caches from direct mapped to fully associative is really a continuum of levels of set associativity: Direct mapped is simply one-way set associative and a fully associative cache with *m* blocks could be called *m*-way set associative. Figure 8.6 shows where block 12 can be placed in a cache according to the block-placement policy.

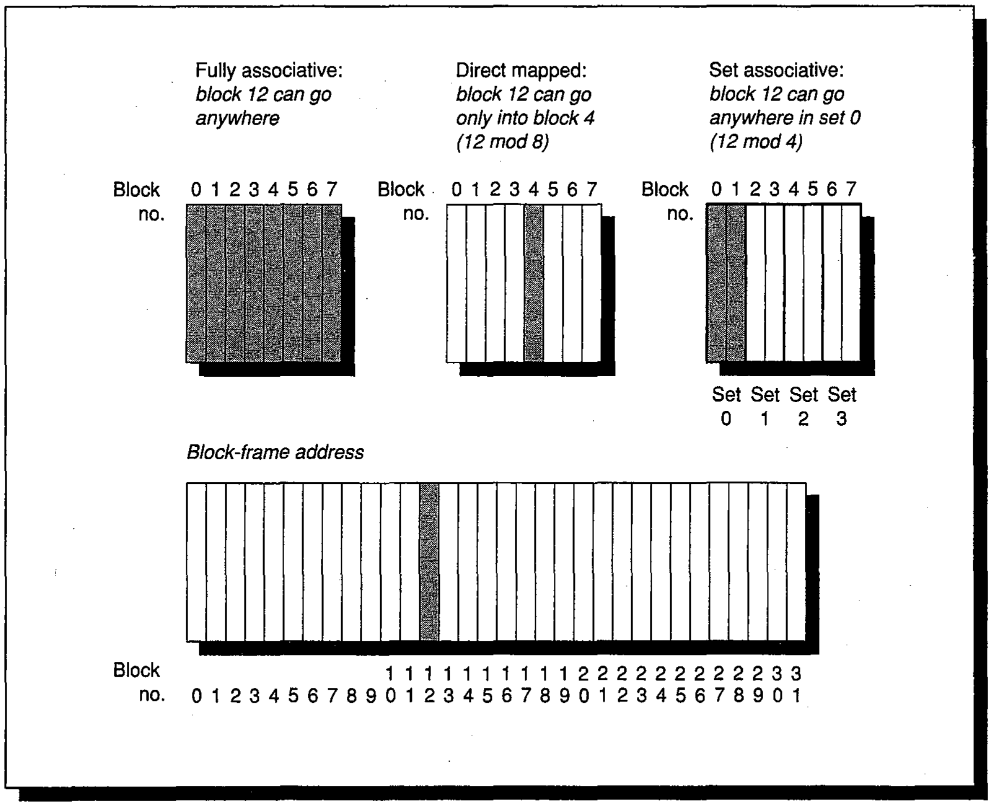
Fully associative: block 12 can go anywhere
Direct mapped: block 12 can go only into block 4 (12 mod 8)
Set associative: block 12 can go anywhere in set 0 (12 mod 4)
Block no.
0 1 2 3 4 5 6 7
Set 0 Set 1 Set 2 Set 3
Block-frame address
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

FIGURE 8.6 The cache has 8 blocks, while memory has 32 blocks. The set-associative organization has 4 sets with 2 blocks per set, called two-way set associative. (Real caches contain hundreds of blocks and real memories contain hundreds of thousands of blocks.) Assume that there is nothing in the cache and that the block-frame address in question identifies lower-level block 12. The three options for caches are shown left to right. In fully associative, block 12 from the lower level can go into any of the 8 blocks of the cache. With direct mapped, block 12 can only be placed into block 4 (12 modulo 8). Set associative, which has some of both features, allows the block to be placed anywhere in set 0 (12 modulo 4). With two blocks per set, this means block 12 can be placed either in block 0 or block 1 of the cache.

Q2: How Is a Block Found If It Is in the Cache?

Caches include an address tag on each block that gives the block-frame address. The tag of every cache block that might contain the desired information is checked to see if it matches the block-frame address from the CPU. Figure 8.7 gives an example. Because speed is of the essence, all possible tags are searched in parallel; serial search would make set associativity counterproductive.

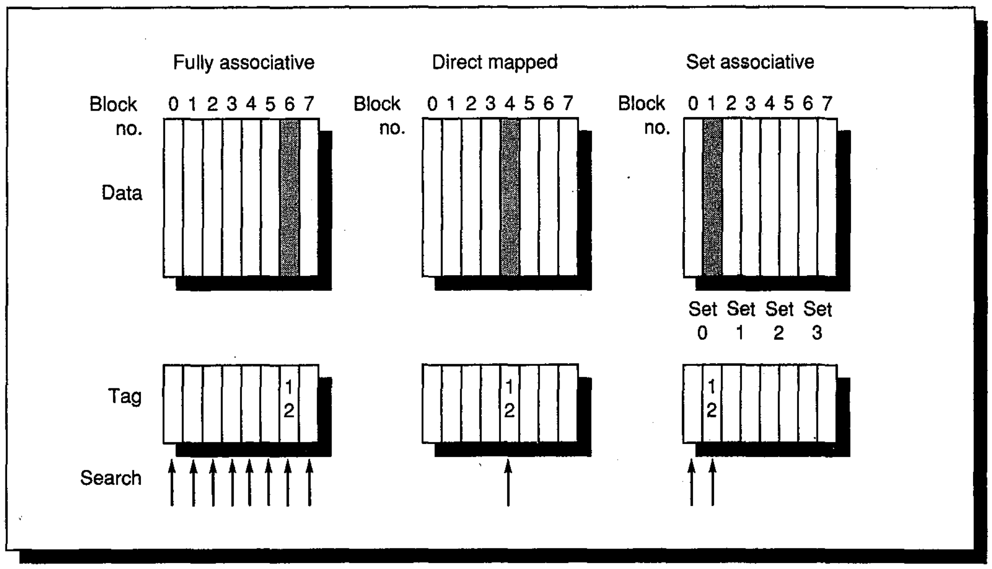

FIGURE 8.7 In fully associative placement, the block for block-frame address 12 can appear in any of the 8 blocks; thus, all 8 tags must be searched. The desired data is found in cache block 6 in this example. In direct-mapped placement there is only one cache block where memory block 12 can be found. In set-associative placement, with 4 sets, memory block 12 must be in set 0 (12 mod 4); thus, the tags of cache blocks 0 and 1 are checked. In this case the data is found in cache block 1. Speed of cache access dictates that searching must be performed in parallel for fully associative and set-associative mappings.

There must be a way to know that a cache block does not have valid information. The most common procedure is to add a *valid bit* to the tag to say whether or not this entry contains a valid address. If the bit is not set, there cannot be a match on this address.

A common omission in finding the cost of caches is to forget the cost of the tag memory. One tag is required for each block. An advantage of increasing block sizes is that the tag overhead per cache entry becomes a smaller fraction of the total cost of the cache.

Before proceeding to the next question, let's explore the relationship of a CPU address to the cache. Figure 8.8 shows how an address is divided into three fields to find data in a set-associative cache: the *block-offset* field used to select the desired data from the block, the *index* field used to select the set, and the *tag* field used for the comparison. While the comparison could be made on more of the address than the tag, there is no need:

- Checking the index would be redundant, since it was used to select the set to be checked (an address stored in set 0, for example, must have 0 in the index field or it couldn't be stored in set 0).
- The offset is unnecessary in the comparison because all block offsets match and the entire block is present or not.

If the total size is kept the same, increasing associativity increases the number of blocks per set, thereby decreasing the size of the index and increasing the size of the tag. That is, the tag/index boundary in Figure 8.8 moves to the right with increasing associativity.

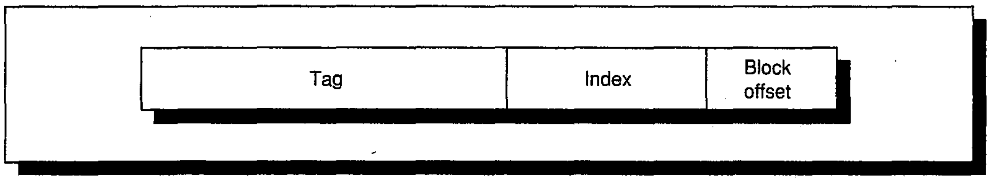

FIGURE 8.8 The 3 portions of an address in a set-associative or direct-mapped cache. The tag is used to check all the blocks in the set and the index is used to select the set. The block offset is the address of the desired data within the block.

Q3: Which Block Should Be Replaced on a Cache Miss?

If the choice were between a block that has valid data and a block that doesn't, then it would be easy to select which block to replace. Alas, the high hit rate of caches means that the overwhelming decision is between blocks that have valid data.

A benefit of direct-mapped placement is that hardware decisions are simplified. In fact, so simple that there is no choice: Only one block is checked for a hit, and only that block can be replaced. With fully associative or set-associative placement, there are several blocks to choose from on a miss. There are two primary strategies employed for selecting which block to replace:

- *Random*—To spread allocation uniformly, candidate blocks are randomly selected. Some systems use a scheme for spreading data across a set of blocks in a pseudorandomized manner to get reproducible behavior, which is particularly useful during hardware debugging.
- *Least-recently used* (LRU)—To reduce the chance of throwing out information that will be needed soon, accesses to blocks are recorded. The block replaced is the one that has been unused for the longest time. This makes use of a corollary of temporal locality: If recently used blocks are likely to be used again, then the best candidate for disposal is the least recently used. Figure 8.9 (page 412) shows which block is the least-recently used for a sequence of block-frame addresses in a fully associative memory hierarchy.

A virtue of random is that it is simple to build in hardware. As the number of blocks to keep track of increases, LRU becomes increasingly expensive and is frequently only approximated. Figure 8.10 shows the difference in miss rates between LRU and random replacement. Replacement policy plays a greater role in smaller caches than in larger caches where there are more choices of what to replace.

Block-frame addresses		3	2	1	0	0	2	3	1	3	0
LRU block number	0	0	0	0	3	3	3	1	0	0	2

FIGURE 8.9 Least-recently used blocks for a sequence of block-frame addresses in a fully associative memory hierarchy. This assumes that there are 4 blocks and that in the beginning the LRU block is number 0. The LRU block number is shown below each new block reference. Another policy, *First-in–first-out* (FIFO), simply discards the block that was used N unique accesses before, independent of its reference pattern in the last N – 1 references. Random replacement generally outperforms FIFO and it is easier to implement.

Associativity:	**2-way**		**4-way**		**8-way**	
Size	**LRU**	**Random**	**LRU**	**Random**	**LRU**	**Random**
16 KB	5.18%	5.69%	4.67%	5.29%	4.39%	4.96%
64 KB	1.88%	2.01%	1.54%	1.66%	1.39%	1.53%
256 KB	1.15%	1.17%	1.13%	1.13%	1.12%	1.12%

FIGURE 8.10 Miss rates comparing least-recently used versus random replacement for several sizes and associativities. This data was collected for a block size of 16 bytes using one of the VAX traces containing user and operating system code (SAVE0). This trace is included in the software supplement for course use. There is little difference between LRU and random for larger size caches in this trace.

Q4: What Happens on a Write?

Reads dominate cache accesses. All instruction accesses are reads, and most instructions don't write to memory. Figure 4.34 (page 181) suggests a mix of 9% stores and 17% loads for four DLX programs, making writes less than 10% of the memory traffic. Making the common case fast means optimizing caches for reads, but Amdahl's Law reminds us that high-performance designs cannot neglect the speed of writes.

Fortunately, the common case is also the easy case to make fast. The block can be read at the same time that the tag is read and compared, so the block read begins as soon as the block-frame address is available. If the read is a hit, the block is passed on to the CPU immediately. If it is a miss, there is no benefit—but also no harm.

Such is not the case for writes. The processor specifies the size of the write, usually between 1 and 8 bytes; only that portion of a block can be changed. In general this means a read-modify-write sequence of operations on the block: read the original block, modify one portion, and write the new block value. Moreover, modifying a block cannot begin until the tag is checked to see if it is a hit. Because tag checking cannot occur in parallel, then, writes normally take longer than reads.

Thus, it is the write policies that distinguish many cache designs. There are two basic options when writing to the cache:

- *Write through* (or *store through*)—The information is written to both the block in the cache **and** to the block in the lower-level memory.
- *Write back* (also called *copy back* or *store in*)—The information is written only to the block in the cache. The modified cache block is written to main memory only when it is replaced.

Write-back cache blocks are called *clean* or *dirty*, depending on whether the information in the cache differs from that in lower-level memory. To reduce the frequency of writing back blocks on replacement, a feature called the *dirty bit* is commonly used. This status bit indicates whether or not the block was modified while in the cache. If it wasn't, the block is not written, since the lower level has the same information as the cache.

Both write back and write through have their advantages. With write back, writes occur at the speed of the cache memory, and multiple writes within a block require only one write to the lower-level memory. Since every write doesn't go to memory, write back uses less memory bandwidth, making write back attractive in multiprocessors. With write through, read misses don't result in writes to the lower level, and write through is easier to implement than write back. Write through also has the advantage that main memory has the most current copy of the data. This is important in multiprocessors and for I/O, which we shall examine in Section 8.8. Hence, multiprocessors want write back to reduce the memory traffic per processor and write through to keep the cache and memory consistent.

When the CPU must wait for writes to complete during write throughs, the CPU is said to *write stall*. A common optimization to reduce write stalls is a *write buffer*, which allows the processor to continue while the memory is updated. As we shall see in Section 8.8, write stalls can occur even with write buffers.

There are two options on a write miss:

- *Write allocate* (also called *fetch on write*)—The block is loaded, followed by the write-hit actions above. This is similar to a read miss.
- *No write allocate* (also called *write around*)—The block is modified in the lower level and not loaded into the cache.

While either write-miss policy could be used with write through or write back, generally write-back caches use write allocate (hoping that subsequent writes to that block will be captured by the cache) and write-through caches often use no write allocate (since subsequent writes to that block will still have to go to memory).

An Example Cache: The VAX-11/780 Cache

To give substance to these ideas, Figure 8.11 shows the organization of the cache on the VAX-11/780. The cache contains 8192 bytes of data in 8-byte blocks with two-way–set-associative placement, random replacement, write through with a one-word write buffer, and no write allocate on a write miss.

Let's trace a cache hit through the steps of a hit as labeled in Figure 8.11. (The five steps are shown as circled numbers.) The address coming into the cache is divided into two fields: the 29-bit block-frame address and 3-bit block offset. The block-frame address is further divided into an address tag and cache index. Step 1 shows this division.

The cache index selects the set to be tested to see if the block is in the cache. (A set is one block from each bank in Figure 8.11.) The size of the index depends on cache size, block size, and set associativity. In this case, a 9-bit index results:

$$\frac{\text{Blocks}}{\text{Bank}} = \frac{\text{Cache size}}{\text{Block size} * \text{Set associativity}} = \frac{8192}{8 * 2} = 512 = 2^9$$

In a two-way–set-associative cache, the index is sent to both banks. This is step 2.

After reading an address tag from each bank, the tag portion of the block-frame address is compared to the tags. This is step 3 in the figure. To be sure the tag contains valid information, the valid bit must be set, or the results of the comparison are ignored.

Assuming one of the tags does match, a 2:1 multiplexer (step 4) is set to select the block from the matching set. Why can't both tags match? It is the job of the replacement algorithm to make sure that an address appears in only one block. To reduce the hit time, the data is read at the same time as the address tags; thus, by the time the block multiplexer is ready, the data is also ready.

This step is needed in set-associative caches, but it can be omitted from direct-mapped caches since there is no selection to be made. The multiplexer used in this step can be on the critical timing path, endangering the clock cycle time of the CPU. (The example on pages 418–419 and the fallacy on page 481 explore the trade-off of lower miss rates and higher clock cycle time.)

In the final step the word is sent to the CPU. All five steps occur within a single CPU clock cycle.

What happens on a miss? The cache sends a stall signal to the CPU telling it to wait, and two words (eight bytes) are read from memory. That takes 6 clock cycles on the VAX-11/780 (ignoring bus interference). When the data arrives, the cache must pick a block to replace; the VAX-11/780 selects one of the two blocks at random. Replacing a block means updating the data, the address tag, and the valid bit. Once this is done, the cache goes through a regular hit cycle and returns the data to the CPU.

Writes are more complicated in the VAX-11/780, as they are in any cache. If the word to be written is in the cache, the first four steps are the same. The next step is to write the data in the block, then write the changed-data portion into the

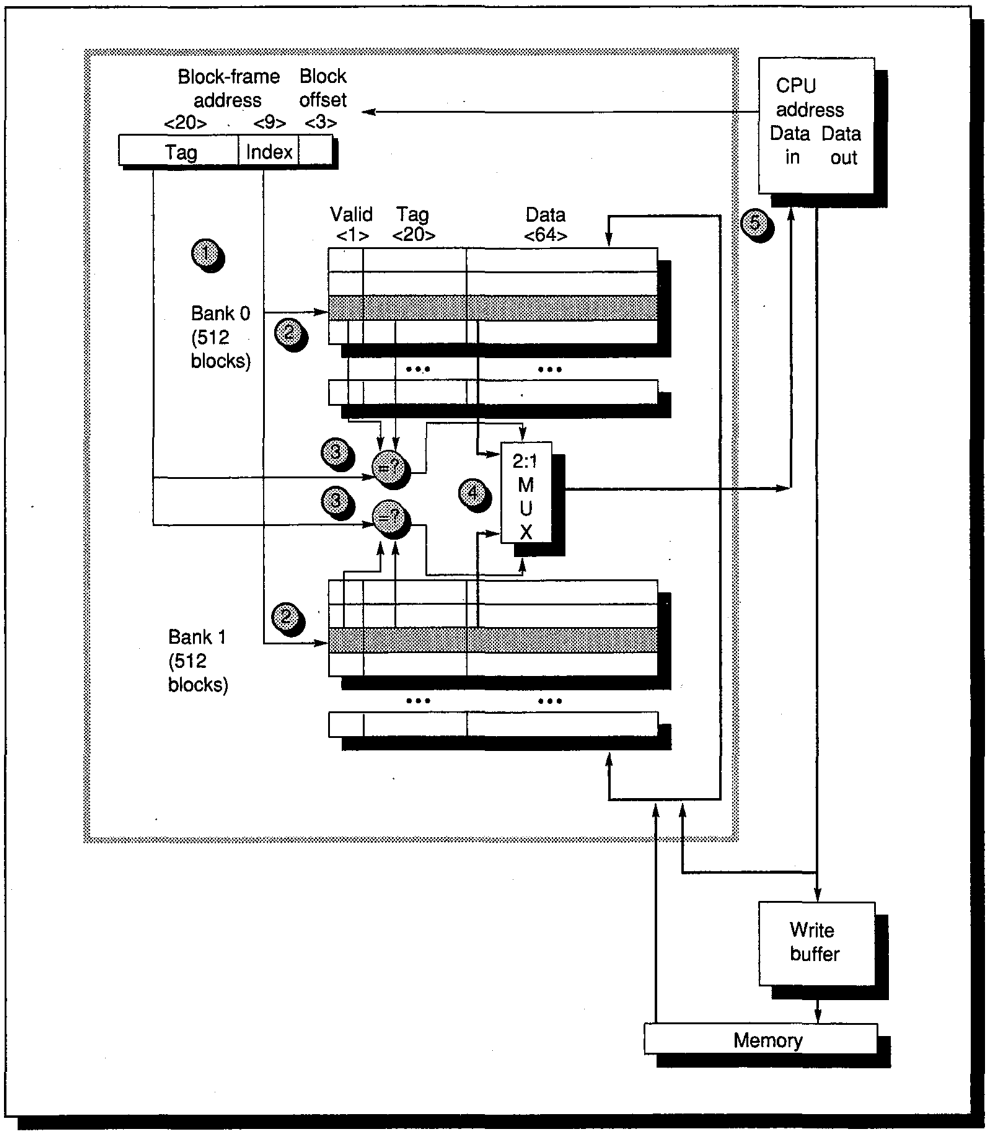

FIGURE 8.11 The organization of the VAX-11/780 cache. The 8-KB cache is two-way set associative with 8-byte blocks. It has 512 sets with two blocks per set; the set is selected by the 9-bit index. The five steps of a read hit, shown as circled numbers in order of occurrence, label this organization. The line from memory to the cache is used on a miss to load the cache. Multiplexing as found in step 4 is not needed in a direct-mapped cache. Note that the offset is connected to chip select of the data SRAMs to allow the proper words to be sent to the 2:1 multiplexer.

cache. The VAX-11/780 uses no write allocate. Consequently, on a write miss the CPU writes "around" the cache to lower-level memory and does not affect the cache.

Since this is a write-through cache, the process isn't yet over. The word is also sent to a one-word write buffer. If the write buffer is empty, the word and full address are written in the buffer, and we are finished. The CPU continues working while the write buffer writes the word to memory. If the buffer is full, the cache (and CPU) must wait until the buffer is empty.

Cache Performance

CPU time can be divided into the clock cycles the CPU spends executing the program and the clock cycles the CPU spends waiting for the memory system. Thus,

$$\text{CPU time} = (\text{CPU-execution clock cycles} + \text{Memory-stall clock cycles}) * \text{Clock cycle time}$$

To simplify evaluation of cache alternatives, sometimes designers assume that all memory stalls are due to the cache. This is true for many machines; on machines where this is not true, the cache still dominates stalls that are not exclusively due to the cache. We use this simplifying assumption here, but it is important to account for **all** memory stalls when calculating final performance!

The formula above raises the question whether the clock cycles for a cache access should be considered part of CPU-execution clock cycles or part of memory-stall clock cycles. While either convention is defensible, the most widely accepted is to include hit clock cycles in CPU-execution clock cycles.

Memory-stall clock cycles can then be defined in terms of the number of memory accesses per program, miss penalty (in clock cycles), and miss rate for reads and writes:

$$\text{Memory-stall clock cycles} = \frac{\text{Reads}}{\text{Program}} * \text{Read miss rate} * \text{Read miss penalty}$$

$$+ \frac{\text{Writes}}{\text{Program}} * \text{Write miss rate} * \text{Write miss penalty}$$

We simplify the complete formula by combining the reads and writes together:

$$\text{Memory-stall clock cycles} = \frac{\text{Memory accessess}}{\text{Program}} * \text{Miss rate} * \text{Miss penalty}$$

Factoring instruction count (IC) from execution time and memory stall cycles, we now get a CPU-time formula that includes memory accesses per instruction, miss rate, and miss penalty:

$$\text{CPU time} = \text{IC} * \left(\text{CPI}_{\text{Execution}} + \frac{\text{Memory accesses}}{\text{Instruction}} * \text{Miss rate} * \text{Miss penalty}\right) * \text{Clock cycle time}$$

Some designers prefer measuring miss rate as *misses per instruction* rather than misses per memory reference:

$$\frac{\text{Misses}}{\text{Instruction}} = \frac{\text{Memory accesses}}{\text{Instruction}} * \text{Miss rate}$$

The advantage of this measure is that it is independent of the hardware implementation. For example, the VAX-11/780 instruction unit can make repeated references to a single byte (see Section 8.7), which can artificially reduce the miss rate if measured as misses per memory reference rather than per instruction executed. The drawback is that this measure is architecture dependent, thus it is most popular with architects working with a single computer family. They then use this version of the CPU-time formula:

$$\text{CPU time} = \text{IC} * \left(\text{CPI}_{\text{Execution}} + \frac{\text{Misses}}{\text{Instruction}} * \text{Miss penalty}\right) * \text{Clock cycle time}$$

We can now explore the consequences of caches on performance.

Example

Let's use the VAX-11/780 as a first example. The cache miss penalty is 6 clock cycles, and all instructions normally take 8.5 clock cycles (ignoring memory stalls). Assume the miss rate is 11%, and there is an average of 3.0 memory references per instruction. What is the impact on performance when behavior of the cache is included?

Answer

$$\text{CPU time} = \text{IC} * \left(\text{CPI}_{\text{Execution}} + \frac{\text{Memory-stall clock cycles}}{\text{Instruction}}\right) * \text{Clock cycle time}$$

The performance, including cache misses, is

$$\text{CPU time}_{\text{with cache}} = \text{IC} * (8.5 + 3.0 * 11\% * 6) * \text{Clock cycle time}$$

$$= \text{Instruction count} * 10.5 * \text{Clock cycle time}$$

The clock cycle time and instruction count are the same, with or without a cache, so CPU time increases with CPI from 8.5 to 10.5. Hence, the impact of the memory hierarchy is to stretch the CPU time by 24%.

Example

Let's now calculate the impact on performance when behavior of the cache is included on a machine with a lower CPI. Assume that the cache miss penalty is 10 clock cycles and, on average, instructions take 1.5 clock cycles; the miss rate is 11%, and there is an average of 1.4 memory references per instruction.

Answer

$$\text{CPU time} = \text{IC} * \left(\text{CPI}_{\text{Execution}} + \frac{\text{Memory-stall clock cycles}}{\text{Instruction}}\right) * \text{Clock cycle time}$$

Making the same assumptions as in the previous example on cache hits, the performance, including cache misses, is

$$\text{CPU time}_{\text{with cache}} = \text{IC} * (1.5 + 1.4*11\%*10) * \text{Clock cycle time}$$

$$= \text{Instruction count}*3.0*\text{Clock cycle time}$$

The clock cycle time and instruction count are the same, with or without a cache, so CPU time increases with CPI from 1.5 to 3.0. Including cache behavior doubles execution time.

As these examples illustrate, cache-behavior penalties range from significant to enormous. Furthermore, cache misses have a double-barreled impact on a CPU with a low CPI and a fast clock:

1. The lower the CPI, the more pronounced the impact is.
2. Independent of the CPU, main memories have similar memory-access times, since they are built from the same memory chips. When calculating CPI, the cache miss penalty is measured in **CPU** clock cycles needed for a miss. Therefore, a higher CPU clock rate leads to a larger miss penalty, even if main memories are the same speed.

The importance of the cache for CPUs with low CPI and high clock rates is thus greater; and, consequently, greater is the danger of neglecting cache behavior in assessing performance of such machines.

While minimizing average memory-access time is a reasonable goal and we will use it in much of this chapter, keep in mind that the final goal is to reduce CPU execution time.

Example

What is the impact of two different cache organizations on the performance of a CPU? Assume that the CPI is normally 1.5 with a clock cycle time of 20 ns, that there are 1.3 memory references per instruction, and that the size of both caches is 64 KB. One cache is direct mapped and the other is two-way set associative. Since the speed of the CPU is tied directly to the speed of the caches, assume the CPU clock cycle time must be stretched 8.5% to accommodate the selection multiplexer of the set-associative cache (step 4 in Figure 8.11 on page 415.) To the first approximation, the cache miss penalty is 200 ns for either cache organization. (In practice it must be rounded up or down to an integer number of clock cycles.) First, calculate the average memory-access time, and then CPU performance.

Answer

Figure 8.12 on page 421 shows that the miss rate of a direct-mapped 64-KB cache is 3.9% and the miss rate for a two-way–set-associative cache of the same size is 3.0%. Average memory-access time is

$$\text{Average memory-access time} = \text{Hit time} + \text{Miss rate} * \text{Miss penalty}$$

Thus, the time for each organization is

$$\text{Average memory-access time}_{1\text{-way}} = 20 + .039*200 = 27.8 \text{ ns}$$

$$\text{Average memory-access time}_{2\text{-way}} = 20*1.085 + .030*200 = 27.7 \text{ ns}$$

The average memory-access time is better for the two-way–set-associative cache.

CPU performance is

$$\text{CPU time} = \text{IC} * \left(\text{CPI}_{\text{Execution}} + \frac{\text{Misses}}{\text{Instruction}} * \text{Miss penalty}\right) * \text{Clock cycle time}$$

$$= \text{IC} * \left(\text{CPI}_{\text{Execution}} * \text{Clock cycle time} + \frac{\text{Memory accesses}}{\text{Instruction}} * \text{Miss rate} * \text{Miss penalty} * \text{Clock cycle time}\right)$$

Substituting 200ns for (Miss penalty * Clock cycle time), the performance of each cache organization is

$$\text{CPU time}_{1\text{-way}} = \text{IC}*(1.5*20 + 1.3*0.039*200) = 40.1*\text{IC}$$

$$\text{CPU time}_{2\text{-way}} = \text{IC}*(1.5*20*1.085 + 1.3*0.030*200) = 40.4*\text{IC}$$

and relative performance is

$$\frac{\text{CPU time}_{2\text{-way}}}{\text{CPU time}_{1\text{-way}}} = \frac{40.4 * \text{Instruction count}}{40.1 * \text{Instruction count}}$$

In contrast to the results of average access-time comparison, the direct-mapped cache leads to slightly better performance. Since CPU time is our bottom-line evaluation (and direct mapped is simpler to build), the preferred cache is direct mapped in this example. (See the fallacy on page 481 for more on this kind of trade-off.)

The Three Sources of Cache Misses: Compulsory, Capacity, and Conflicts

An intuitive model of cache behavior attributes all misses to one of three sources:

- *Compulsory*—The first access to a block is not in the cache, so the block must be brought into the cache. These are also called *cold start misses* or *first reference misses.*
- *Capacity*—If the cache cannot contain all the blocks needed during execution of a program, capacity misses will occur due to blocks being discarded and later retrieved.

- *Conflict*—If the block-placement strategy is set associative or direct mapped, conflict misses (in addition to compulsory and capacity misses) will occur because a block can be discarded and later retrieved if too many blocks map to its set. These are also called *collision misses.*

Figure 8.12 shows the relative frequency of cache misses, broken down by the "three Cs." To show the benefit of associativity, conflict misses are divided into misses caused by each decrease in associativity. The categories are labeled *n*-way, meaning the misses caused by going to the lower level of associativity from the next one above. Here are the four categories:

8-way: from fully associative (no conflicts) to 8-way associative

4-way: from 8-way associative to 4-way associative

2-way: from 4-way associative to 2-way associative

1-way: from 2-way associative to 1-way associative (direct mapped)

Figure 8.13 (page 422) presents the same data graphically. The top graph shows absolute miss rates; the bottom graph plots percentage of all the misses by cache size.

Having identified the three Cs, what can a computer designer do about them? Conceptually, conflicts are the easiest: Fully associative placement avoids all conflict misses. Associativity is expensive in hardware, however, and may slow access time (see the example above or the second fallacy in Section 8.10), leading to lower overall performance. There is little to be done about capacity except to buy larger memory chips. If the upper-level memory is much smaller than what is needed for a program, and a significant percentage of the time is spent moving data between two levels in the hierarchy, the memory hierarchy is said to *thrash.* Because so many replacements are required, thrashing means the machine runs close to the speed of the lower-level memory, or maybe even slower due to the miss overhead. Making blocks larger reduces the number of compulsory misses, but it can increase conflict misses.

The three C's give insight into the cause of misses, but this simple model has its limits. For example, increasing cache size reduces conflict misses as well as capacity misses, since a larger cache spreads out references. Thus, a miss might move from one category to the other as parameters change. Three C's ignore replacement policy, since it is difficult to model and since, in general, it is of less significance. In specific circumstances the replacement policy can actually lead to anomalous behavior, such as poorer miss rates for larger associativity, which is directly contradictory to the three C's model.

Cache size	Degree associative	Total miss rate	Miss-rate components (relative percent) (*Sum = 100% of total miss rate*) Compulsory		Capacity		Conflict	
1 KB	1-way	0.191	0.009	5%	0.141	73%	0.042	22%
1 KB	2-way	0.161	0.009	6%	0.141	87%	0.012	7%
1 KB	4-way	0.152	0.009	6%	0.141	92%	0.003	2%
1 KB	8-way	0.149	0.009	6%	0.141	94%	0.000	0%
2 KB	1-way	0.148	0.009	6%	0.103	70%	0.036	24%
2 KB	2-way	0.122	0.009	7%	0.103	84%	0.010	8%
2 KB	4-way	0.115	0.009	8%	0.103	90%	0.003	2%
2 KB	8-way	0.113	0.009	8%	0.103	91%	0.001	1%
4 KB	1-way	0.109	0.009	8%	0.073	67%	0.027	25%
4 KB	2-way	0.095	0.009	9%	0.073	77%	0.013	14%
4 KB	4-way	0.087	0.009	10%	0.073	84%	0.005	6%
4 KB	8-way	0.084	0.009	11%	0.073	87%	0.002	3%
8 KB	1-way	0.087	0.009	10%	0.052	60%	0.026	30%
8 KB	2-way	0.069	0.009	13%	0.052	75%	0.008	12%
8 KB	4-way	0.065	0.009	14%	0.052	80%	0.004	6%
8 KB	8-way	0.063	0.009	14%	0.052	83%	0.002	3%
16 KB	1-way	0.066	0.009	14%	0.038	57%	0.019	29%
16 KB	2-way	0.054	0.009	17%	0.038	70%	0.007	13%
16 KB	4-way	0.049	0.009	18%	0.038	76%	0.003	6%
16 KB	8-way	0.048	0.009	19%	0.038	78%	0.001	3%
32 KB	1-way	0.050	0.009	18%	0.028	55%	0.013	27%
32 KB	2-way	0.041	0.009	22%	0.028	68%	0.004	11%
32 KB	4-way	0.038	0.009	23%	0.028	73%	0.001	4%
32 KB	8-way	0.038	0.009	24%	0.028	74%	0.001	2%
64 KB	1-way	0.039	0.009	23%	0.019	50%	0.011	27%
64 KB	2-way	0.030	0.009	30%	0.019	65%	0.002	5%
64 KB	4-way	0.028	0.009	32%	0.019	68%	0.000	0%
64 KB	8-way	0.028	0.009	32%	0.019	68%	0.000	0%
128 KB	1-way	0.026	0.009	34%	0.004	16%	0.013	50%
128 KB	2-way	0.020	0.009	46%	0.004	21%	0.006	33%
128 KB	4-way	0.016	0.009	55%	0.004	25%	0.003	20%
128 KB	8-way	0.015	0.009	59%	0.004	27%	0.002	14%

FIGURE 8.12 Total miss rate for each size cache and percentage of each according to the "three Cs." Compulsory misses are independent of cache size, while capacity misses decrease as capacity increases. Hill [1987] measured this trace using 32-byte blocks and LRU replacement. It was generated on a VAX-11 running Ultrix by mixing three systems' traces, using a multiprogramming workload and three user traces. The total length was just over a million addresses; the largest piece of data referenced during the trace was 221 KB. Figure 8.13 (page 422) shows the same information graphically. Note that the 2:1 cache rule of thumb (inside front cover) is supported by the statistics in this table: a direct-mapped cache of size N has about the same miss rate as a 2-way–set-associative cache of size N/2.

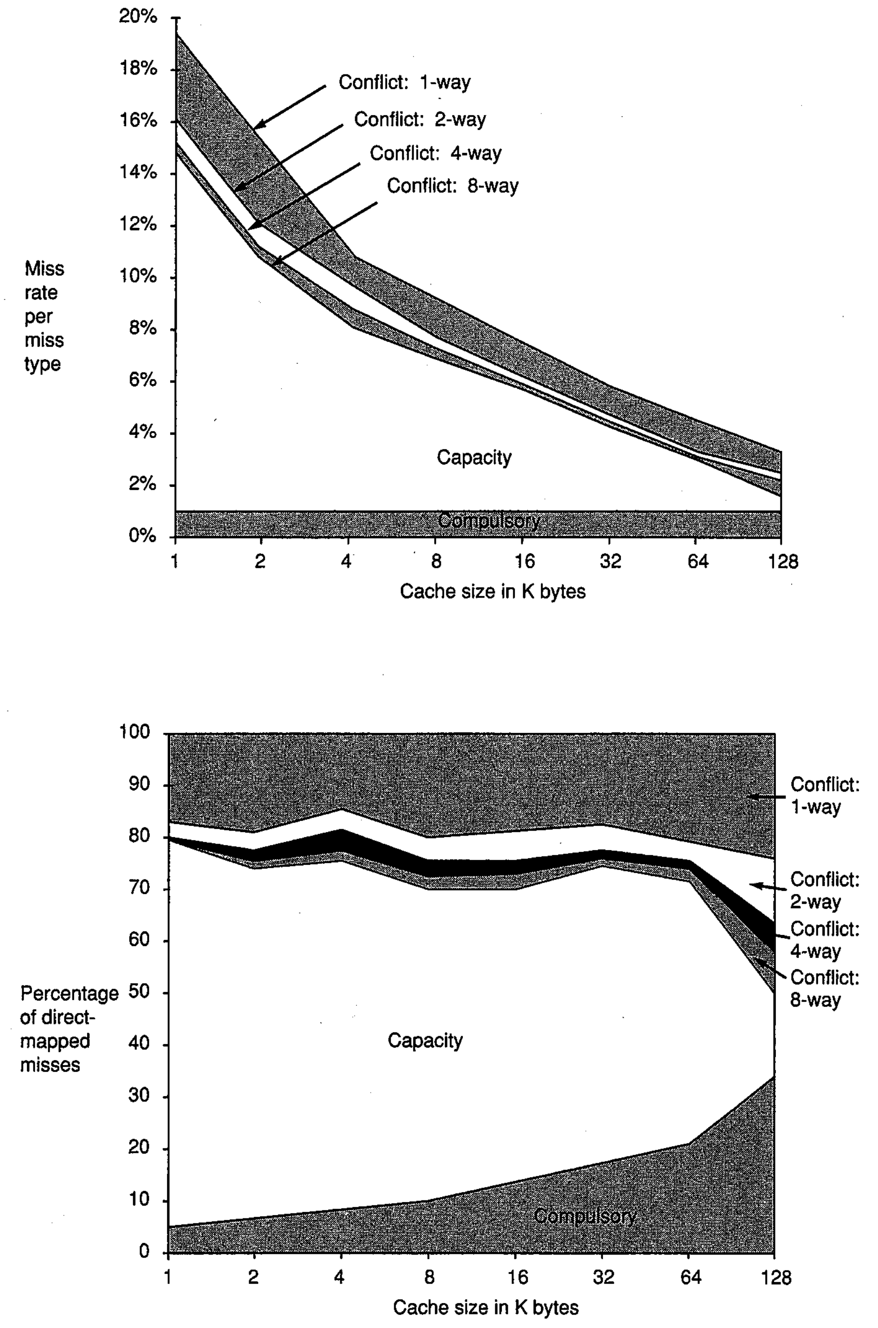

FIGURE 8.13 Total miss rate (top) and distribution of miss rate (bottom) for each size cache according to three Cs for the data in Figure 8.12 (page 421). The top diagram is the actual miss rates, while the bottom diagram is scaled to the direct-mapped miss ratio.

Choices for Block Sizes in Caches

Figures 8.3 and 8.4 (page 406) showed the abstract tradeoff of block size versus miss rate and memory-access time. Figures 8.14 and 8.15 (page 424) show the specific numbers for a set of programs and cache sizes. Larger block sizes reduce compulsory misses, as the principle of spatial locality suggests. At the same time, larger blocks also reduce the number of blocks in the cache, increasing conflict misses.

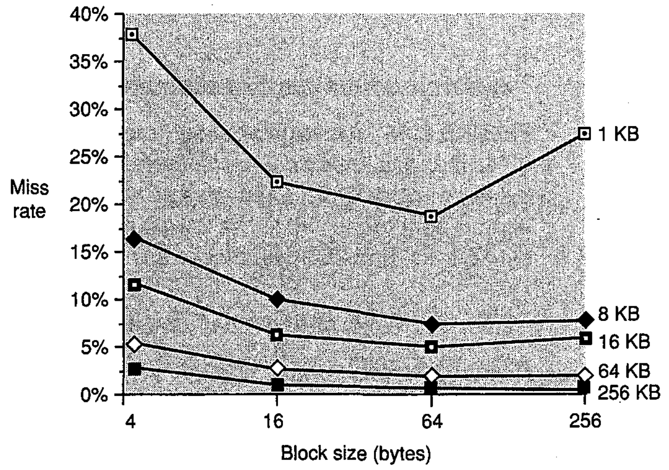

FIGURE 8.14 Miss rate versus block size. Note that for a 1-KB cache, 256-byte blocks have a higher miss rate than either 16- or 64-byte blocks. (The smallest block is 4 bytes.) In this particular example, the cache would have to be 256 KB in order for increasing block size to always result in decreased misses. This data was collected for a direct-mapped cache using one of the VAX traces containing user and operating system code, which is distributed with this book (SAVE0).

Instruction-Only or Data-Only Caches Versus Unified Caches

Unlike other levels of the memory hierarchy, caches are sometimes divided into instruction-only and data-only caches. Caches that can contain either instructions or data are *unified* caches, or *mixed* caches. The CPU knows whether it is issuing an instruction address or a data address, so there can be separate ports for both, thereby doubling the bandwidth between the cache and the CPU. (Section 6.4 in Chapter 6 shows the advantages of dual memory ports for pipelined execution.) Separate caches also offers the opportunity of optimizing each cache separately: different capacities, block sizes, and associativities may lead to better performance. Splitting thus affects the cost and performance far beyond what is indicated by the change in miss rates. We limit our discussion to that point now simply to show how miss rates for instructions differ from miss rates for data.

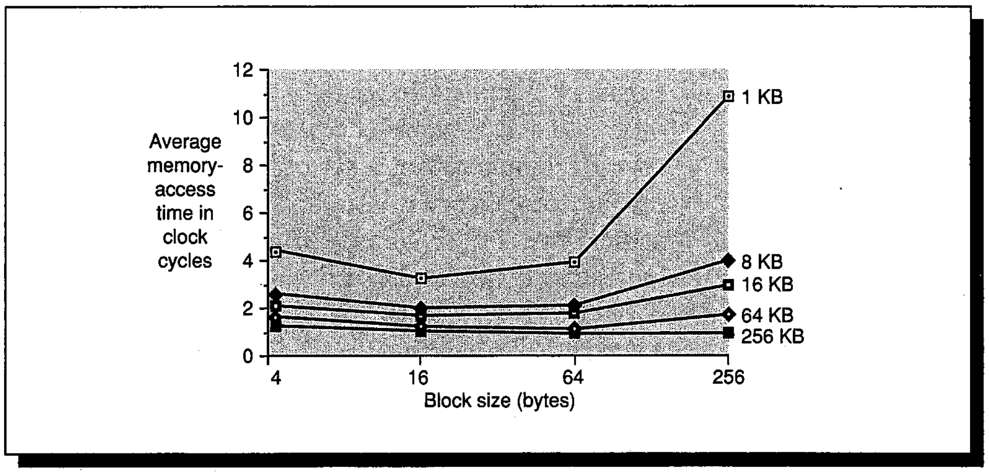

FIGURE 8.15 Average access time versus block size using the miss rates in Figure 8.14. This assumes an 8–clock-cycle latency and that the memory and bus can transfer 4 bytes per clock cycle. On a miss all the blocks are loaded into the cache before the requested word is sent to the CPU. The lowest average memory-access time is either for 16-byte or 64-byte blocks, and 256-byte blocks are better than 4-byte blocks only for the largest cache.

Figure 8.16 shows that instruction-only caches have lower miss rates than data-only caches. Separating instructions and data removes misses due to conflicts between instruction blocks and data blocks, but the split also fixes the cache space devoted to each type. A fair comparison of separate instruction and data caches to unified caches requires the total cache size to be the same. Therefore, a separate 1-KB instruction cache and 1-KB data cache should be compared to a unified 2-KB cache. Calculating the average miss rate with separate instruction-only and data-only caches necessitates knowing the percentage of memory references to each cache.

Size	Instruction only	Data only	Unified
0.25 KB	22.2%	26.8%	28.6%
0.50 KB	17.9%	20.9%	23.9%
1 KB	14.3%	16.0%	19.0%
2 KB	11.6%	11.8%	14.9%
4 KB	8.6%	8.7%	11.2%
8 KB	5.8%	6.8%	8.3%
16 KB	3.6%	5.3%	5.9%
32 KB	2.2%	4.0%	4.3%
64 KB	1.4%	2.8%	2.9%
128 KB	1.0%	2.1%	1.9%
256 KB	0.9%	1.9%	1.6%

FIGURE 8.16 Miss rates for instruction-only, data-only, and unified caches of different sizes. The data are for a 2-way–associative cache using LRU replacement with 16-byte blocks for an average of user/system traces on the VAX-11 and system traces on the IBM 370 [Hill 1987]. The percentage of instruction references in these traces is about 53%.

Example Which has the lower miss rate: a 16-KB instruction cache with a 16-KB data cache or a 32-KB unified cache? Assume 53% of the references are instructions.

Answer As stated in the legend of Figure 8.16, 53% of the memory accesses are instruction references. Thus, the overall miss rate for the split caches is

$$53\% * 3.6\% + 47\% * 5.3\% = 4.4\%$$

A 32-KB unified cache has a slightly lower miss rate of 4.3%.

8.4 Main Memory

... the one single development that put computers on their feet was the invention of a reliable form of memory, namely, the core memory. ... Its cost was reasonable, it was reliable and, because it was reliable, it could in due course be made large.

Maurice Wilkes, *Memoirs of a Computer Pioneer* (1985, p. 209)

Provided there is only one level of cache, main memory is the next level down in the hierarchy. Main memory satisfies the demands of caches and vector units, and serves as the I/O interface as it is the destination of input as well as the source for output. Unlike caches, performance measures of main memory emphasize both latency and bandwidth. Generally, main memory latency (which affects the cache miss penalty) is the primary concern of the cache, while main-memory bandwidth is the primary concern of I/O and vector units. As cache blocks grow from 4-8 bytes to 64–256 bytes, main memory bandwidth becomes important to caches as well. The relationship of main memory and I/O is discussed in Chapter 9.

Memory latency is traditionally quoted using two measures—access time and cycle time. *Access time* is the time between when a read is requested and when the desired word arrives, while *cycle time* is the minimum time between requests to memory. In the 1970s, as DRAMs grew in capacity the cost of a package with all the necessary address lines became an issue. The solution was to multiplex the address lines, thereby cutting the number of address pins in half. The top half of the address comes first, during the *row-access strobe*, or RAS. This is followed by the second half of the address during the *column-access strobe*, or CAS. These names come from the internal chip organization, for the memory is organized as a rectangular matrix addressed by rows and columns.

An additional requirement of DRAMs derives from the property signified by its first letter, D, for dynamic. Every DRAM must have every row accessed within a certain time window, such as 2 milliseconds, or the information in the DRAM can be lost. This requirement means that the memory system is occasionally unavailable because it is sending a signal telling every chip to refresh. The cost of a refresh is typically a full memory access (RAS and CAS) for each row of the DRAM. Since the memory matrix in a DRAM is likely to be square, the number of steps in a refresh is usually the square root of the DRAM capacity.

In contrast to DRAMs are SRAMs—the first letter standing for "static." The dynamic nature of the circuits for DRAM require data to be written back after being read, hence the difference between the access time and the cycle time and also the need to refresh. SRAMs use more circuits per bit to prevent the information from being disturbed when read. Thus, unlike DRAMs, there is no difference between access time and cycle time and there is no need to refresh SRAM. In DRAM designs the emphasis is on capacity, while SRAM designs are concerned with both capacity **and** speed. (Because of this concern, SRAM address lines are not multiplexed.) For memories designed in comparable technologies, the capacity of DRAMs is roughly 16 times that of SRAMs, and the cycle time of SRAMs is 8 to 16 times faster than DRAMs.

The main memory of virtually every computer sold in the last decade is composed of semiconductor DRAMs (and virtually all caches use SRAM). Amdahl suggested a rule of thumb that memory capacity should grow linearly with CPU speed to keep a balanced system (see Section 1.4), and CPU designers rely on DRAMs to supply that demand: they expect a four-fold improvement in capacity every three years. Unfortunately, the performance of DRAMs is growing at a much slower rate. Figure 8.17 shows a performance improvement in row-access time of about 22% per generation, or 7% per year. As noted in Chapter 1, CPU performance improved 18% to 35% per year prior to 1985, and since that time has jumped to 50% to 100% per year. Figure 8.18 plots these optimistic and pessimistic CPU performance projections against the steady 7% performance improvement in DRAM speeds.

		Row access (RAS)		Column	
Year of introduction	**Chip size**	**Slowest DRAM**	**Fastest DRAM**	**access (CAS)**	**Cycle time**
1980	64 Kbit	180 ns	150 ns	75 ns	250 ns
1983	256 Kbit	150 ns	120 ns	50 ns	220 ns
1986	1 Mbit	120 ns	100 ns	25 ns	190 ns
1989	4 Mbit	100 ns	80 ns	20 ns	165 ns
1992?	16 Mbit	≈85 ns	≈65 ns	≈15 ns	≈140 ns

FIGURE 8.17 Times of fast and slow DRAMs with each generation. The improvement by a factor of two in column access accompanied the switch from NMOS DRAMs to CMOS DRAMs. With three years per generation, the performance improvement of row access time is about 7% per year. Data in the last row represent predicted performance for 16-Mbit DRAMs, which are not yet available.

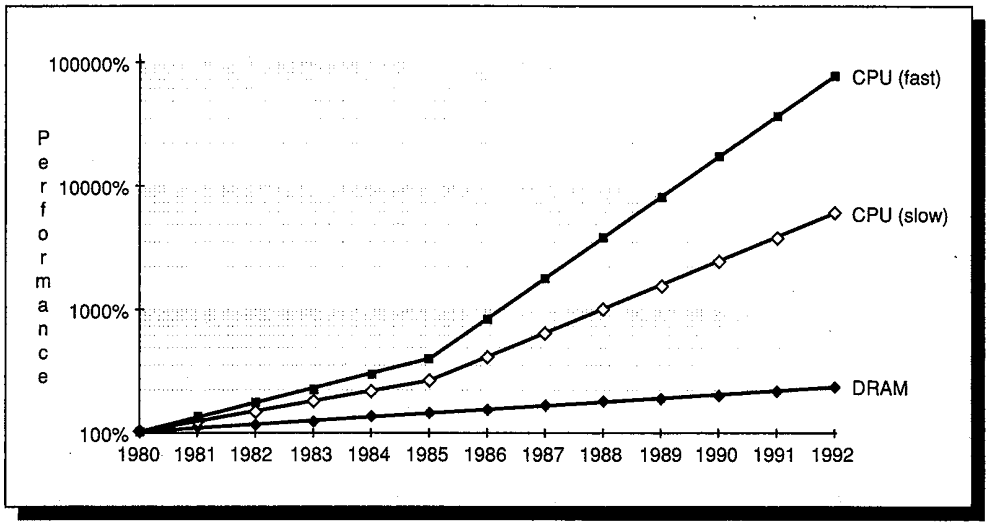

FIGURE 8.18 Starting with 1980 performance as a baseline, the performance of DRAMs and CPUs are plotted over time. The DRAM baseline is 64 KB in 1980, with three years to the next generation. The slow CPU line assumes a 19% improvement per year until 1985 and a 50% improvement thereafter. The fast CPU line assumes a 26% performance improvement between 1980 and 1985 and 100% per year thereafter. Note that the vertical axis must be on a logarithmic scale to record the size of the CPU–DRAM performance gap.

The CPU–DRAM performance gap is clearly a problem on the horizon—Amdahl's Law warns us what will happen if we ignore one portion of the computation while trying to speed up the rest. Section 8.8 will describe what can be done with cache organization to reduce this performance gap, but simply making caches larger cannot eliminate it. Innovative organizations of main memory are needed as well. In the rest of this section we will examine techniques for organizing memory to improve performance, including techniques especially for DRAMs.

Organizations for Improving Main Memory Performance

While it is generally easier to improve memory bandwidth with new organizations than it is to reduce latency, a bandwidth improvement does allow cache-block size to increase without a corresponding increase in the miss penalty.

Let's illustrate these organizations with the case of satisfying a cache miss. Assume the performance of the basic memory organization is

1 clock cycle to send the address

6 clock cycles for the access time per word

1 clock cycle to send a word of data

Given a cache block of four words, the miss penalty is 32 clock cycles, with a memory bandwidth of one-half byte per clock cycle.

Figure 8.19 shows some of the options to faster memory systems. The simplest approach to increasing memory bandwidth, then, is to make the memory wider.

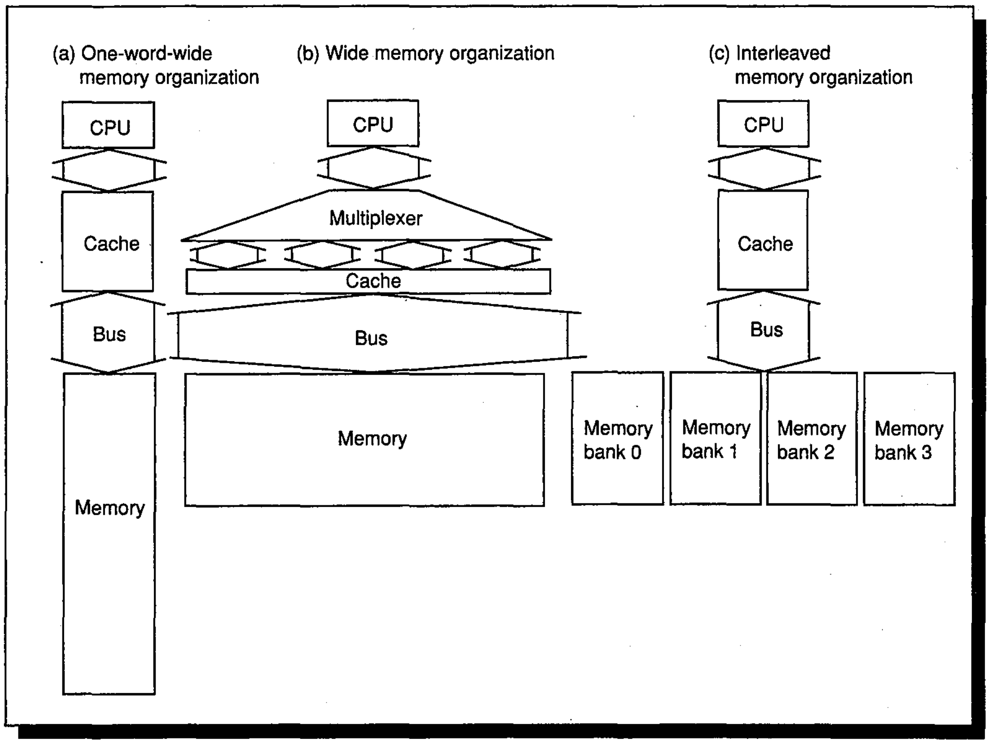

FIGURE 8.19 Three examples of bus width, memory width, and memory interleaving to achieve higher memory bandwidth. (a) is the simplest design, with everything the width of one word; (b) shows a wider memory, bus, and cache; while (c) shows a narrow bus and cache with an interleaved memory.

Wider Main Memory

Caches are often organized with a width of one word because most CPU accesses are that size. Main memory, in turn, is one word wide to match the width of the cache. Doubling or quadrupling the width of the memory will therefore double or quadruple the memory bandwidth. With a main memory width of two words the miss penalty in our example would drop from 4*8 or 32 clock cycles to 2*8 or 16 clock cycles. At four words wide the miss penalty is just 1*8 clock cycles. The bandwidth is then one byte per clock cycle at two words wide and two bytes per clock cycle when the memory is four words wide.

There is cost in the wider bus. The CPU will still access the cache a word at a time, so there now needs to be a multiplexer between the cache and the CPU—and that multiplexer may be on the critical timing path. (If the cache is faster than the bus, however, the multiplexer can be placed between the cache and the bus.) Another drawback is that since main memory is traditionally expansible by the customer, the minimum increment is doubled or quadrupled. Finally, memories with error correction have difficulties with writes to a portion of the protected block (e.g., a write of a byte); the rest of the data must be read so that the new error correction code can be calculated and stored when the data is written. If the error correction is done over the full width, the wider memory will increase the frequency of such "read-modify-write" sequences because more writes become partial block writes. Many designs of wider memory have separate error correction every 32 bits since most writes are that size. One example of wider main memory was a computer whose cache, bus, and memory were all 512 bits wide.

Interleaved Memory

Memory chips can be organized in banks to read or write multiple words at a time rather than a single word. The banks are one word wide so that the width of the bus and the cache need not change, but sending addresses to several banks permits them all to read simultaneously. For example, sending an address to four banks (with access times shown on page 427) yields a miss penalty of 1+6+4*1 or 11 clock cycles, giving a bandwidth of about 1.5 bytes per clock cycle. Banks are also valuable on writes. While back-to-back writes would normally have to wait for earlier writes to finish, banks allow one clock cycle for each write, provided the writes are not destined to the same bank.

The mapping of addresses to banks affects the behavior of the memory system. The example above assumes the addresses of the four banks are interleaved at the word level—bank 0 has all words whose address modulo 4 is 0, bank 1 has all words whose address modulo 4 is 1, and so on. This mapping is referred to as the *interleaving factor*; *interleaved memory* normally means banks of memory that are word interleaved. This optimizes sequential memory accesses. A cache-read miss is an ideal match to word-interleaved memory, as the words in a block are read sequentially. Write-back caches make writes as well as reads sequential, getting even more efficiency from interleaved memory.

Example

What can interleaving and a wide memory buy? Consider the following description of a machine and its cache performance:

Block size = 1 word

Memory bus width = 1 word

Miss rate = 15%

Memory accesses per instruction = 1.2

Cache miss penalty = 8 cycles (as above)

Average cycles per instruction (ignoring cache misses) = 2

If we change the block size to two words, the miss rate falls to 10%, and a four-word block has a miss rate of 5%. What is the improvement in performance of interleaving two ways and four ways versus doubling the width of memory and the bus, assuming the access times on page 427.

Answer The CPI for the base machine using one-word blocks is

$$2 + (1.2*15\%*8) = 3.44$$

Since the clock cycle time and instruction count won't change in this example, we can calculate performance improvement by just comparing CPI.

Increasing the block size to two words gives the following options:

32-bit bus and memory, no interleaving	= 2 + (1.2*10%*2*8)	= 3.92
32-bit bus and memory, interleaving	= 2 + (1.2*10%*(1+6+2))	= 3.08
64-bit bus and memory, no interleaving	= 2 + (1.2*10%*1*8)	= 2.96

Thus, doubling the block size slows down the straightforward implementation (3.92 versus 3.44), while interleaving or wider memory is 12% or 16% faster, respectively. If we increase the block size to four, the following is obtained:

32-bit bus and memory, no interleaving	= 2 + (1.2*5%*4*8)	= 3.92
32-bit bus and memory, interleaving	= 2 + (1.2*5%*(1+6+4))	= 2.66
64-bit bus and memory, no interleaving	= 2 + (1.2*5%*2*8)	= 2.96

Again, the larger block hurts performance for the simple case, although the interleaved 32-bit memory is now fastest—29% versus 16% for the wider memory and bus.

The original motivation for memory banks was interleaving sequential accesses. A further reason is to allow multiple independent accesses. Multiple memory controllers allow banks (or sets of word-interleaved banks) to operate independently. For example, an input device may use one controller and its memory, the cache may use another, and a vector unit may use a third. To reduce the chances of conflicts many banks are needed; the NEC SX/3, for instance, has up to 128 banks.

As capacity per memory chip increases, there are fewer chips in the same-sized memory system, making multiple banks much more expensive. For example, a 16-MB main memory takes 512 memory chips of 256 K (262,144) x 1 bits, easily organized into 16 banks of 32 memory chips. But it takes only 32 4-M (4,194,304) x 1-bit memory chips for 16 MB, making one bank the limit. This is the main disadvantage of interleaved memory banks. Even though the

Amdahl/Case rule of thumb for balanced computer systems recommends increasing memory capacity with increasing CPU performance, the 60% growth in DRAM capacity exceeded the rate of increase in CPU performance in the past (page 17 of Chapter 1). If the rate of increase of CPU speeds seen in the late 1980s can be maintained (Figure 8.18, page 427) and these systems follow the Amdahl/Case rule of thumb, then the number of chips may not be reduced.

A second disadvantage of interleaving is again the difficulty of main memory expansion. Since memory-control hardware will likely need equal-sized banks, doubling the main memory will probably be the minimum increment.

DRAM-Specific Interleaving for Improving Main Memory Performance

DRAM access times are divided into row access and column access. DRAMs buffer a row of bits inside the DRAM for the column access. This row is usually the square root of the DRAM size—1024 bits for 1 Mbit, 2048 for 4 Mbits, and so on. All DRAMs come with optional timing signals that allow repeated accesses to the buffer without a row-access time. There are three versions for this optimization:

- *Nibble mode*—The DRAM can supply three extra bits from sequential locations for every row access.
- *Page mode*—The buffer acts like a SRAM; by changing column address, random bits can be accessed in the buffer until the next row access or refresh time.
- *Static column*—Very similar to page mode, except that it's not necessary to hit the column-access strobe line every time the column address changes; this option has been nicknamed SCRAM, for static column DRAM.

Starting with the 1-Mbit DRAMs, most dies can perform any of the three options, with the optimization selected at the time the die is packaged by choosing which pads to wire up. These operations change the definition of cycle time for DRAMs. Figure 8.20 (page 432) shows the traditional cycle time plus the fastest speed between accesses in the optimized mode.

The advantage of these optimizations is that they use the circuitry already on the DRAMs, adding little cost to the system while achieving almost a fourfold improvement in bandwidth. For example, nibble mode was designed to take advantage of the same program behavior as interleaved memory. The chip reads four bits at a time internally, supplying four bits externally in the time of four optimized cycles. Unless the bus transfer time is faster than the optimized cycle time, the cost of four-way interleaved memory is only more complicated timing control. Page mode and static column could also be used to get even higher interleaving with slightly more complex control. DRAMs also tend to have weak tristate buffers, implying traditional interleaving with more memory chips must include buffer chips for each memory bank.

Chip size	Row access Slowest DRAM	Row access Fastest DRAM	Column access	Cycle time	Optimized time nibble, page, static column
64 Kbits	180 ns	150 ns	75 ns	250 ns	150 ns
256 Kbits	150 ns	120 ns	50 ns	220 ns	100 ns
1 Mbits	120 ns	100 ns	25 ns	190 ns	50 ns
4 Mbits	100 ns	80 ns	20 ns	165 ns	40 ns
16 Mbits	≈85 ns	≈65 ns	≈15 ns	≈140 ns	≈30 ns

FIGURE 8.20 DRAM cycle time for the optimized accesses. This is Figure 8.17 (page 426) with a column added to show the optimized cycle time for the three modes. Starting with the 1-Mbit DRAM, optimized cycle time is about four times faster than unoptimized cycle time. It is so much faster that page mode was renamed *fast page mode*. The optimized cycle time is the same no matter which of the 3 optimized modes is selected.

Thus, the authors expect that most main memory systems in the future will use such techniques to reduce the CPU–DRAM performance gap. Unlike traditional interleaved memories, there are no disadvantages using these DRAM modes as DRAMs scale upward in capacity, nor is there the problem of the minimum expansion increment in main memory.

One possibility that recently arrived is DRAMs that do not multiplex the address lines. At the cost of a larger package, a full random access falls between a row-access time and a column-access time in Figure 8.20. If unencoded DRAMs can stay close to the price per bit of the high volume encoded DRAMs, the computer architect will have another option in his bag of tricks for memory design.

8.5 Virtual Memory

... a system has been devised to make the core drum combination appear to the programmer as a single level store, the requisite transfers taking place automatically.

Kilburn et al. [1962]

At any instant in time computers are running multiple processes, each with its own address space. (Processes are described in the next section.) It would be too expensive to dedicate a full-address-space worth of memory for each process, especially since many processes use only a small part of their address space. Hence, there must be a means of sharing a smaller amount of physical memory between many processes. One way to do this, *virtual memory*, divides physical memory into blocks and allocates them to different processes. Inherent in such an approach must be a *protection* scheme that restricts a process to the blocks belonging just to that process. Most forms of virtual memory also reduce the time to start a program, since not all code and data need be in physical memory before a program can begin.

While virtual memory is essential for current computers, sharing is not the reason virtual memory was invented. In former days if a program became too large for physical memory, it was up to the programmer to make it fit. Programmers divided programs into pieces and then identified the pieces that were mutually exclusive. These *overlays* were loaded or unloaded under user program control during execution, with the programmer ensuring that the program never tried to access more physical main memory in the machine. As one can well imagine, this responsibility eroded programmer productivity. Virtual memory, invented to relieve programmers of this burden, automatically managed the two levels of the memory hierarchy represented by main memory and secondary storage.

In addition to sharing protected memory space and automatically managing the memory hierarchy, virtual memory also simplifies loading the program for execution. Called *relocation*, this procedure allows the same program to run in any location in physical memory. (Prior to the popularity of virtual memory, machines would include a relocation register just for that purpose.) An alternative to a hardware solution would be software that changed all addresses in a program each time it was run.

Several general memory-hierarchy terms from Section 8.3 apply to virtual memory, while some other terms are different. *Page* or *segment* is used for block, and *page fault*, or *address fault*, is used for miss. With virtual memory, the CPU produces *virtual addresses* that are translated by a combination of hardware and software to *physical addresses*, which can be used to access main memory. This process is called *memory mapping* or *address translation*. Today, the two memory hierarchy levels controlled by virtual memory are DRAMs and magnetic disks. Figure 8.21 shows a typical range of memory hierarchy parameters for virtual memory.

Block (page) size	512 – 8192 bytes
Hit time	1–10 clock cycles
Miss penalty	100,000 – 600,000 clock cycles
(Access time)	(100,000–500,000 clock cycles)
(Transfer time)	(10,000–100,000 clock cycles)
Miss rate	0.00001%–0.001%
Main memory size	4 MB – 2048 MB

FIGURE 8.21 Typical ranges of parameters for virtual memory. These figures, contrasted with the values for caches in Figure 8.5 (page 408), represent increases of 10 to 100,000 times.

There are further differences between caches and virtual memory beyond those quantitative ones seen by comparing Figure 8.21 (page 433) to Figure 8.5 (page 408):

- Replacement on cache misses is primarily controlled by hardware, while virtual memory replacement is primarily controlled by the operating system; the longer miss penalty means the operating system can afford to get involved and spend more time deciding what to replace.
- The size of the processor address determines the size of virtual memory, but the cache size is normally independent of the processor address.
- In addition to acting as the lower-level memory for main memory in the hierarchy, secondary storage is also used for the file system that is not normally part of the address space; most of secondary storage is in fact taken up by the file system.

Virtual memory encompasses several related techniques. Virtual memory systems can be categorized into two classes: those with fixed-size blocks, called *pages*, and those with variable size blocks, called *segments*. Pages are typically fixed at 512 to 8192 bytes, while segment size varies. The largest segment supported on any machine ranges from 2^{16} bytes up to 2^{32} bytes; the smallest segment is one byte.

The decision to use paged virtual memory versus segmented virtual memory affects the CPU. Paged addressing has a single, fixed-size address divided into page number and offset within a page, analogous to cache addressing. A single address does not work for segmented addresses; the variable size of segments requires one word for a segment number and one word for an offset within a segment, for a total of two words. An unsegmented address space is simpler for the compiler.

The pros and cons of these two approaches have been well documented in operating systems textbooks; these are summarized in Figure 8.22. Because of the replacement problem (the third line of the figure), few machines today use pure segmentation. Some machines use a hybrid approach, called *paged segments*, in which a segment is an integral number of pages. This simplifies replacement because memory need not be contiguous, and the full segments need not be in main memory.

We are now ready to answer the four memory-hierarchy questions for virtual memory.

Q1: Where Can a Block Be Placed in Main Memory?

The miss penalty for virtual memory involves access to a rotating magnetic storage device and is therefore quite high. Given the choice of lower miss rates or a simpler placement algorithm, operating systems designers always pick lower miss rates because of the horrendous cost of a miss. Thus, operating systems allow blocks to be placed anywhere in main memory. According to the terminology in Figure 8.6 (page 409), this strategy would be labeled fully associative.

Q2: How Is a Block Found If It Is in Main Memory?

Both paging and segmentation rely on a data structure that is indexed by the page or segment number. This data structure contains the physical address of the block. For paging, the offset is simply concatenated to this physical page address (see Figure 8.23, page 436). For segmentation, the offset is added to the segment's physical address to obtain the final virtual address.

	Page	Segment
Words per address	One	Two (segment and offset)
Programmer visible?	Invisible to application programmer	May be visible to application programmer
Replacing a block	Trivial (all blocks are the same size)	Hard (must find contiguous, variable-size, unused portion of main memory)
Memory use inefficiency	Internal fragmentation (unused portion of page)	*External fragmentation* (unused pieces of main memory)
Efficient disk traffic	Yes (adjust page size to balance access time and transfer time)	Not always (small segments may transfer just a few bytes)

FIGURE 8.22 Paging versus segmentation. Both can waste memory, depending on the block size and how well the segments fit together in main memory. Programming languages with unrestricted pointers require both the segment and the address to be passed. A hybrid approach, called *paged segments*, shoots for the best of both worlds: segments are composed of pages, so replacing a block is easy, yet a segment may be treated as a logical unit.

This data structure containing the physical page addresses usually takes the form of a *page table*. Indexed by the virtual page number, the size of the table is the number of pages in the virtual-address space. Given a 28-bit virtual address, 4 KB pages, and 4 bytes per page-table entry, the size of the page table would be 256 KB. To reduce the size of this data structure, some machines apply a hashing function to the virtual address so that the data structure need only be the size of the number of **physical** pages in main memory; this number would be much smaller than the number of virtual pages. Such a structure is called an *inverted page table*. Using the example above, a 64-MB physical memory would only need 64 KB (4*64 MB/4 KB) for an inverted page table.

To reduce address translation time, computers use a cache dedicated to these address translations, called a translation-lookaside buffer, or simply translation buffer. They are described in more detail shortly.

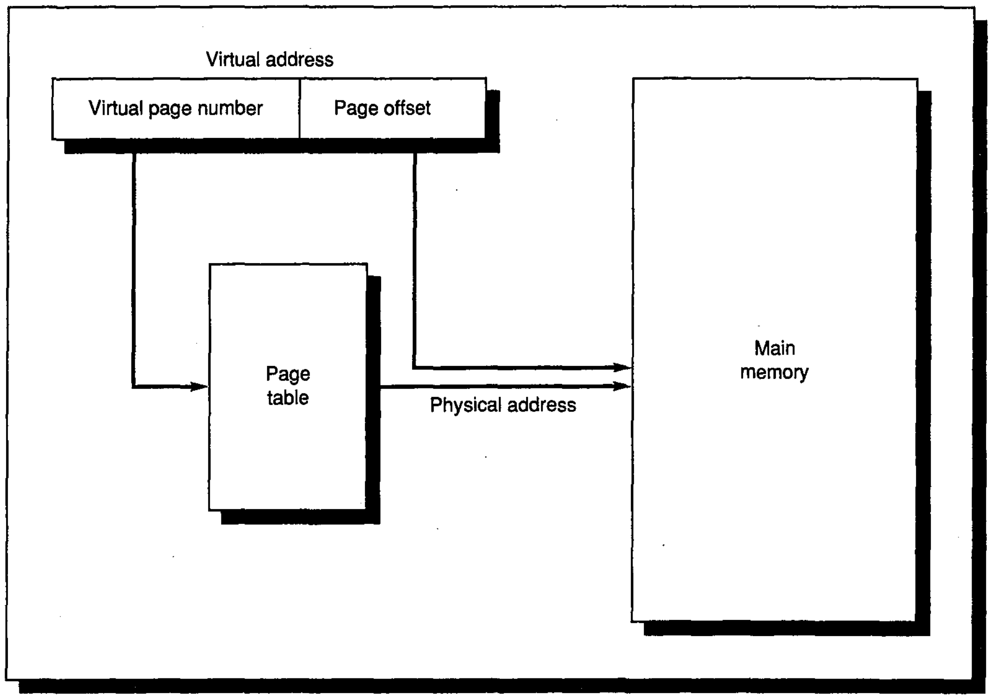

FIGURE 8.23 The mapping of a virtual address to a physical address via a page table.

Q3: Which Block Should Be Replaced on a Virtual Memory Miss?

As mentioned above, the overriding operating system guideline is minimizing page faults. Consistent with this, almost all operating systems try to replace the least-recently used (LRU) block, because that is the one least likely to be needed. To help the operating system estimate LRU, many machines provide a *use bit* or *reference bit*, which is set whenever a page is accessed. The operating system periodically clears the use bits and later records them so it can determine which pages were touched during a particular time period. By keeping track in this way, the operating system can select a page that is among the least-recently referenced.

Q4: What Happens on a Write?

The level below main memory contains rotating magnetic disks that take hundreds of thousands of clock cycles to access. Because of the great discrepancy in access time, no one has yet built a virtual memory operating system that can write through main memory straight to disk on every store by the CPU. (This remark should not be interpreted as an opportunity to become famous by being the first to build one!) Thus, the write strategy is always write back. Since the cost of an unnecessary access to the next-lower level is so high, virtual memory systems include a dirty bit so that the only blocks written to disk are those that have been altered since they were loaded from the disk.

Selecting a Page Size

The most obvious architectural parameter is the page size. Choosing the page is a question of balancing forces that favor a larger page size versus those favoring a smaller size. The following favor a larger size:

- The size of the page table is inversely proportional to the page size; memory (or other resources used for the memory map) can therefore be saved by making the pages bigger.
- Transferring larger pages to or from secondary storage, possibly over a network, is more efficient than transferring smaller pages.

(The larger page size may also help in address translation of cache addresses; see Section 8.8.)

The main motivation for a smaller page size is conserving storage. A small page size will result in less wasted storage when a contiguous region of virtual memory is not equal in size to a multiple of the page size. The term for this unused memory in a page is *internal fragmentation.* Assuming that each process has three primary segments (text, heap, and stack), the average wasted storage per process will be 1.5 times the page size. This is negligible for machines with megabytes of memory and page sizes in the range of 2 KB to 8 KB. Of course, when the page sizes become very large (more than 32 KB), lots of storage (both main and secondary) may be wasted, as well as I/O bandwidth. A final concern is process start-up time; many processes are small, so larger page sizes would lengthen the time to invoke a process.

Techniques for Fast Address Translation

Page tables are usually so large that they are stored in main memory and often paged themselves. This means that every memory access takes at least twice as long, with one memory access to obtain the physical address and a second access to get the data. This cost is far too dear.

One remedy is to remember the last translation, so that the mapping process is skipped if the current address refers to the same page as the last one. A more general solution is to again rely on the principle of locality; if the references have locality, then the address translations for the references must also have locality. By keeping these address translations in a special cache, a memory access rarely requires a second access to translate the data. This special address translation cache is referred to as a *translation-lookaside buffer* or TLB, also called a "translation buffer," or TB. A TLB entry is like a cache entry where the tag holds portions of the virtual address and the data portion holds a physical page-frame number, protection field, use bit, and dirty bit. To change the physical page-frame number or protection of an entry in the page table the operating system must make sure the old entry is not in the TLB; otherwise, the system won't behave properly. Note that this dirty bit means the corresponding **page** is dirty, not that the address translation in the TLB is dirty nor that a particular block in the data cache is dirty. Figure 8.24 shows typical parameters for TLBs.

Block size	4 – 8 bytes (1 page-table entry)
Hit time	1 clock cycle
Miss penalty	10 – 30 clock cycles
Miss rate	0.1% – 2%
TLB size	32 – 8192 bytes

FIGURE 8.24 Typical values of key memory-hierarchy parameters for TLBs. TLBs are simply caches for the virtual-to-physical address translations found in the page tables.

One architectural challenge stems from the difficulty of combining caches with virtual memory. The virtual address must first go through the TLB **before** the physical address can access the cache, meaning that the cache hit time must be stretched to allow for address translation (or the pipeline could be stretched as in Chapter 6). One way to reduce hit time is to access the cache with the page offset, the portion of the virtual address that does not need to be translated. While the cache address tags are being read, the virtual portion of the address (the page-frame address) is sent to the TLB to be translated. The address comparison is then between the physical address from the TLB and the cache tag. Since the TLB is usually smaller and faster than the cache-address-tag memory, simultaneous TLB reading need not slow down cache hit times. The drawback with this scheme is that a direct-mapped cache can be no bigger than a page. Another option, virtually addressed caches, is discussed in Section 8.8.

8.6 Protection and Examples of Virtual Memory

The invention of multiprogramming led to new demands for protection and sharing between programs. These are closely tied to virtual memory in computers today, and so we cover the topic here along with two examples of virtual memory.

Multiprogramming lead to the concept of a *process*. Metaphorically, a process is a program's breathing air and living space; that is, a running program plus any state needed to continue running the program. Timesharing means sharing the CPU and memory with several users at the same time to give the appearance that every user has his own machine. Thus, at any instant it must be possible to switch from one process to another. This is called a *process switch* or *context switch*. Figure 8.25 shows the frequency of these switches on the VAX 8700.

Instructions between process switches	19,353
Clock cycles between process switches	170,113
Time between process switches	7.7 ms

FIGURE 8.25 Frequency of process switches on VAX 8700 for timesharing workload. Most switching occurs on interrupts caused by I/O events or by the interval timer (see Figure 5.10, page 216). Since neither the latency of the I/O device nor the timer is affected by the speed of the CPU clock, faster machines generally execute more clock cycles and instructions between process switches.

A process must operate correctly whether it executes continuously from start to finish or is interrupted repeatedly and switched with other processes. The responsibility for maintaining correct process behavior is shared by the computer designer, who must ensure that the CPU portion of the process state can be saved and restored, and the operating system designer, who must guarantee that processes do not interfere with each others' computations. The safest way to protect the state of one process from another would be to copy the current information to disk. But a process switch would then take seconds—far too long for a timesharing environment. The problem is solved by operating systems partitioning main memory so that several different processes have their state in memory at the same time. This means that the operating system designer needs help from the computer designer to provide protection so that one process cannot modify another. Besides protection, the computers also provide for sharing of code and data between processes, to allow communication between processes or to save memory by reducing the number of copies of identical information.

Protecting Processes

The simplest protection mechanism is a pair of registers that checks every address to be sure that it falls between the two limits traditionally called *base* and *bound*. An address is valid if

$$\text{Base} \le \text{Address} \le \text{Bound}$$

In some systems the address is considered an unsigned number that is always added to the base, so the valid test is just

$$(\text{Base} + \text{Address}) \le \text{Bound}$$

For user processes to be protected from each other, they can't change the base and bounds registers, yet the operating system must be able to change the registers so that it can switch processes. Hence, the computer designer has three more responsibilities in helping the operating system designer protect processes from each other:

1. Provide at least two modes indicating whether the running process is a user process or an operating system process, sometimes called a *kernel* process, a *supervisor* process or an *executive* process.

2. Provide a portion of the CPU state that a user process can use but not write. This includes the base/bound registers, a user/supervisor mode bit(s), and the interrupt enable/disable bit. Users are prevented from writing this state because the operating system cannot control user processes if users can change the address-range checks, disable interrupts, or give themselves supervisor privileges.

3. Provide mechanisms whereby the CPU can go from user mode to supervisor mode and vice versa. The first direction is typically accomplished by a *system call*, implemented as a special instruction that transfers control to a dedicated location in supervisor code space. The PC from the point of the system call is saved, and the CPU is placed in supervisor mode. The return to user mode is like a subroutine return that restores the previous user/supervisor mode.

Base and bound constitute the minimum protection system. Virtual memory provides an alternative to this simple model. As we have seen, the CPU address must go through a mapping from virtual to physical address. This provides the opportunity for the hardware to check further for errors in the program or to protect processes from each other. The simplest way of doing this is to add access permission flags to each page or segment. For example, since few programs today intentionally modify their own code, an operating system can detect accidental writes to code by offering read-only protection to pages. This can be extended by adding a user/kernel bit to prevent a user program from trying to access pages that belong to the kernel. As long as the CPU provides a read/write signal and a user/kernel signal, it is easy for the address translation hardware to detect stray memory accesses before they can do damage. As seen in Section 5.6 of Chapter 5, such reckless behavior interrupts the CPU. Obviously, user programs cannot be allowed to modify the page table.

Protection can be escalated, depending on the apprehension of the computer designer or the purchaser. Rings added to the CPU-protection structure expand memory-access protection from two levels (user and kernel) to many more. Like a military classification system of top secret, secret, classified, and unclassified, concentric *rings* of security levels allow the most trusted to access anything, the second most trusted to access everything except the innermost level, and so on down to "civilian" programs which are the least trusted and, hence, have the most limited range of accesses. There may also be restrictions on the entrance point between the levels. The 80286 protection structure, which uses rings, is described later in this section. It is not clear today whether rings are an improvement on the simple system of user and kernel modes.

As the designer's apprehension escalates to trepidation, these simple rings may not suffice. The fact that a program in the inner sanctum can access anything calls for a new classification system. Instead of a military model, the analogy of this next model is to keys and locks: A program can't unlock access to the data unless it has the key. For these keys, or *capabilities*, to be useful, the hardware and operating system must be able to explicitly pass them from one program to another without allowing a program itself to forge them. Such checking requires a great deal of hardware support.

A Paged Virtual Memory Example: VAX-11 Memory Management and the VAX-11/780 TLB

The VAX architecture uses a combination of segmentation and paging. This combination provides protection while minimizing page-table size. The address space is first divided into two segments: process (bit 31 = 0) and system (bit 31=1). Every process has its own private space and shares system space with every other process. The process address space is further subdivided into two regions called P0 and P1, using bit 30 to distinguish them. Area P0 (bit 30 = 0) grows from address 0 upward while P1 (bit 30 = 1) grows downward to 0. Figure 8.26 shows the layout of P0 and P1. The two segments can grow until one exceeds its 2^{30} address-space size and its virtual memory is exhausted. Many systems today use some such combination of predivided segments and paging. The approach provides many advantages: Segmentation divides system and process address space and conserves page-table space, while paging provides virtual memory, relocation, and protection.

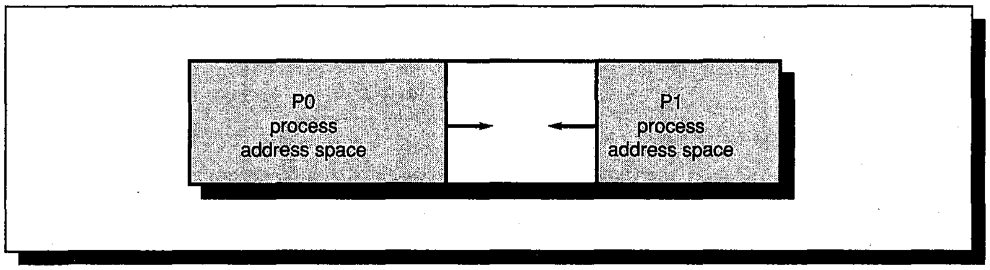

FIGURE 8.26 The organization of P0 and P1 in the VAX. This is the process half of the address space, selected with a 0 in bit 31 of a virtual address. Bit 30 of the address divides P0 and P1. Operating systems put the text and heap areas into P0 and a downward growing stack into P1.

To conserve page-table space, each of the three regions—P0 process, P1 process, and system—is provided with a pair of base-bound registers that indicate the start and limit of the page table for each region. The alternative would be to have a single page table that covers the full address space, independent of the program's actual size. The small size of the VAX pages—512 bytes, yielding large page tables—makes such conservation especially important.

Figure 8.27 (page 442) shows the mapping of a VAX address. The two most-significant bits of an address select which segment or base-bound–register pair to use in selecting a page table and checking the reference. A one in the first bit selects the system page table, whose base and length are found respectively in the system base register and in the system length register. A zero in the first bit of an address (as in the figure) selects page table P0 or P1, found by the P0 or P1 base registers and checked by the P0 or P1 limit (bound) registers. The P0 and P1 page tables are in the system-space virtual memory, while the system page table is in physical memory.

This offers an interesting way to conserve physical memory. Since the P0 and P1 page tables are also in virtual memory, this means the page tables can be paged. Just as some code and data can remain on disk during program execution, the page-table translation entries for that code and data can remain on disk until they are used. This is especially important for programs whose memory size varies dynamically during execution, as page tables can be increased as P0 or P1 space grows. In the worst case, then, a process page fault can result in a second page fault bringing in the missing piece of the process page table needed to complete the address translation. What prevents all pages tables from being

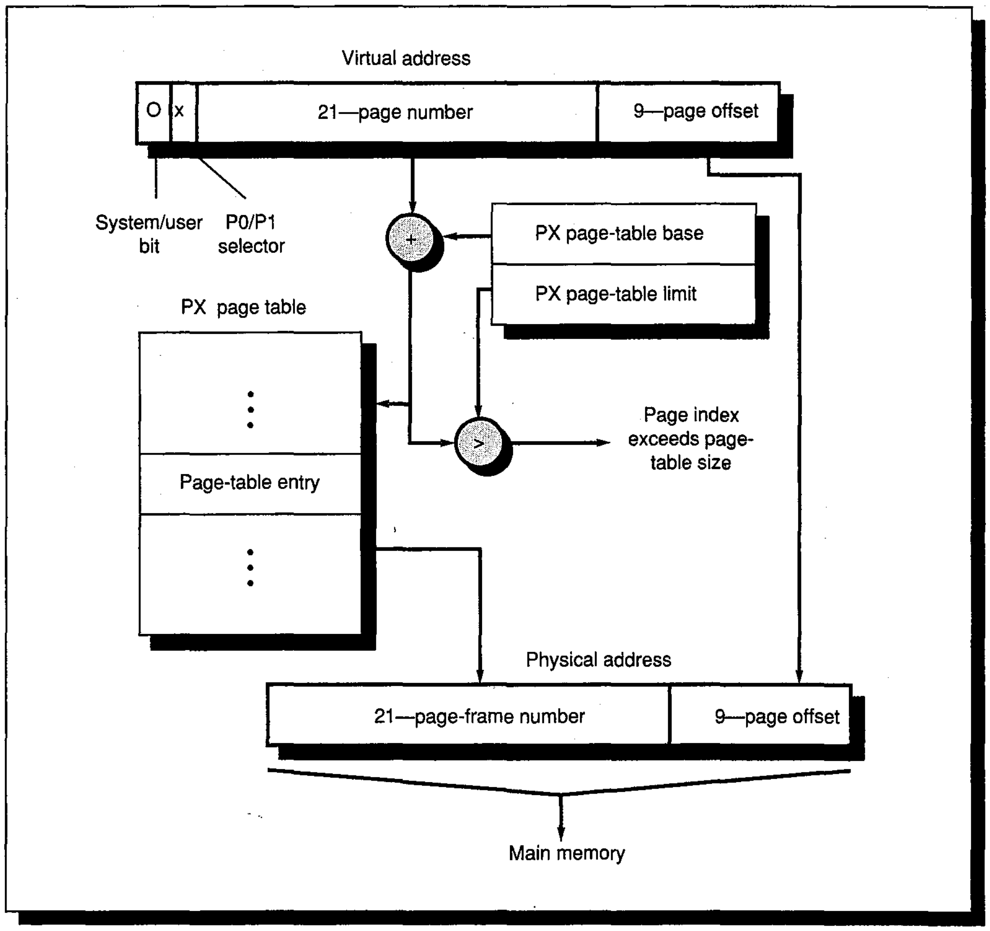

FIGURE 8.27 The mapping of a VAX virtual address. PX refers to either P0 or P1.

migrated to secondary storage? Some system page tables are loaded into physical memory when the operating system is booted and are prevented from migrating to disk. Thus, eventually a series of faults must cross an address stored in the system page table that is "frozen" into main memory.

While this explains translation of legal addresses, what prevents the user from creating illegal address translations and getting into mischief? The page tables themselves are protected from being written to by user programs. Thus, the user can try any virtual address, but by controlling the page-table entries the operating system controls what physical memory is accessed. Sharing of memory between processes is accomplished by having a page-table entry in each address space point to the same physical-memory page.

A *page-table entry* (PTE) on the VAX is straightforward. Other than the physical page-frame number these are the only architecture-defined fields:

M—the *modify bit* indicating the page is dirty

V—the *valid bit* indicating this PTE has a valid address

PROT—four protection bits

Note that there is no reference or use bit. Hence, a page-replacement algorithm such as LRU must rely on the modify bit or some software technique to measure usage. Rather than simply a kernel/user protection structure, the VAX uses a four-level structure consisting of kernel, executive, supervisor, and user. The four protection bits in the PTE contain 16 encodings of selected combinations of no access, read-only access, and read-write access, with the four security levels. For example, 1001 means read-write access for kernel and executive-level processes, read access for supervisor-level processes, and no access for user-level processes. To further isolate these four levels, each has its own stack and its own copy of the stack pointer (R15).

The first implementation of this architecture was the VAX-11/780, which employs a TLB to reduce address-translation time. Figure 8.28 shows the key parameters of this TLB.

Block size	1 PTE (4 bytes)
Hit time	1 clock cycle
Miss penalty (average)	22 clock cycles
Miss rate	1% – 2%
Cache size	128 PTEs (512 bytes)
Block selection	Random
Write strategy	(Not applicable)
Block placement	2-way set associative

FIGURE 8.28 Memory hierarchy parameters of the VAX-11/780 TLB.

Figure 8.29 shows the VAX-11/780 TLB organization, with each step of a translation labeled. The TLB uses two-way–set-associative placement; thus, the translation begins (steps 1 and 2) by sending a portion of the virtual address ("index") to both sets to select the two tags that are to be compared. Of course, the tag must be marked valid to allow a match. At the same time, the type of memory access is checked for a violation (also in step 2) against protection information in the TLB.

For reasons similar to those in the cache case, there is no need to include the 9 bits of the VAX page offset in the TLB; nor is there reason to include the 6 address bits to index the TLB. The remaining bits are used in the comparison (step 3). The matching address tag sends the corresponding physical address through the multiplexer (step 4). The page offset is then combined with the physical page frame to form a full physical address (step 5).

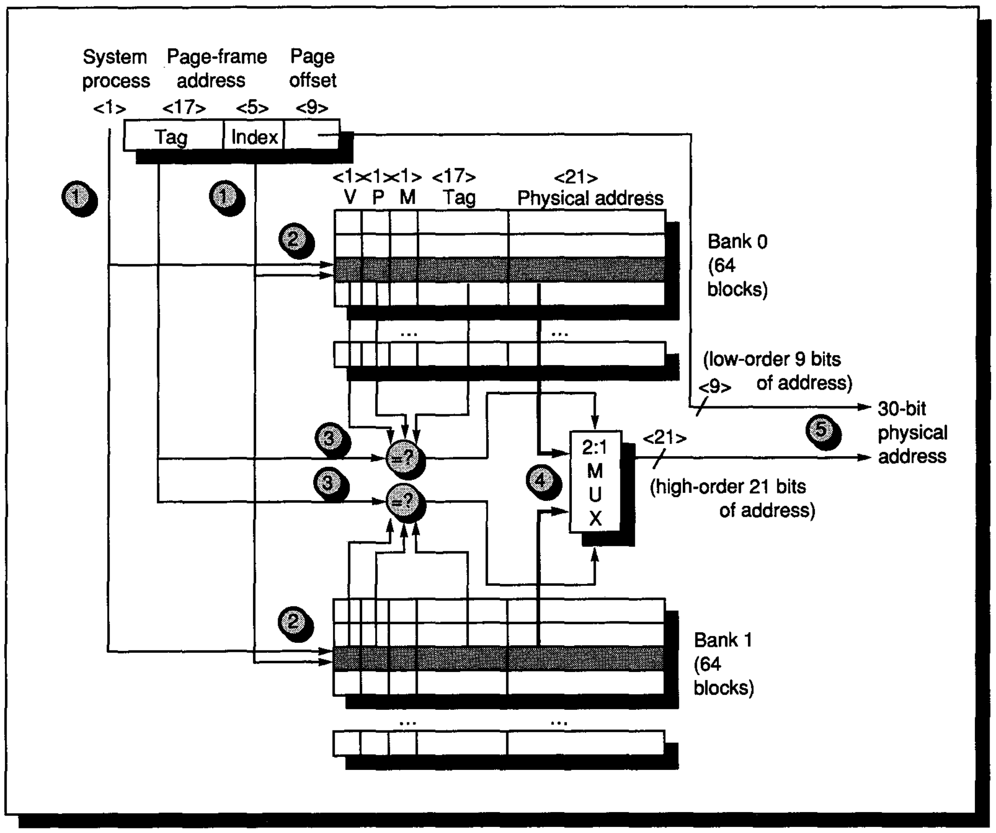

FIGURE 8.29 Operation of the VAX-11/780 TLB during address translation. The five steps of a TLB hit are shown as circled numbers.

There is one unusual feature of the VAX-11/780 TLB: The TLB is further subdivided to make sure the process portion of the address occupies no more than 50% of the TLB entries. The top 32 entries of each bank are reserved for system space, and the bottom 32 are reserved for process space. The most significant bit of the address is used to select the appropriate half of the TLB (step 1). Since the system portion of the address space is the same for all processes, a process switch invalidates only the lower 32 entries of each bank for the VAX-11/780 TLB. This restriction had two goals. The first was to reduce the process-switch time by reducing the number of TLB entries that had to be invalidated; the second was to improve performance by preventing the system or user process from throwing out the other's translations when process switches were frequent. Splitting the TLB will usually lead to higher overall TLB miss rate, but may reduce the peak TLB miss rate in heavily process-switching environments.

A Segmented Virtual Memory Example: Protection in the Intel 80286/80386

The second system is the most dangerous system a man ever designs. . . . The general tendency is to over-design the second system, using all the ideas and frills that were cautiously sidetracked on the first one.

F. P. Brooks, Jr., *The Mythical Man-Month* (1975)

The original 8086 used segments for addressing, yet it provided nothing for virtual memory or for protection. Segments had base registers but no bound registers and no access checks; and before a segment register could be loaded the corresponding segment had to be in physical memory. Intel's dedication to virtual memory and protection is evident in subsequent models, with a few fields extended to support larger addresses.

Like the VAX, the 80286 has four levels of protection. The innermost level (0) corresponds to VAX kernel mode, and the outermost level (3) corresponds to VAX user mode. The 80286 also follows the VAX by having separate stacks for each level to avoid security breaches between the levels. There are also data structures analogous to VAX page tables that contain the physical addresses for segments, as well as a list of checks to be made on translated addresses.

The Intel designers did not stop there. The 80286 divides the address space, allowing both the operating system and the user access to the full space. The 80286 user can call an operating system routine in this space and even pass parameters to it retaining full protection. This is not a trivial action, since the stack for the operating system is different from the user's stack. Moreover, the 80286 allows the operating system to maintain the protection level of the called routine for the parameters that are passed to it. This potential loophole in protection is prevented by not allowing the user to ask the operating system to access something indirectly that he would not have been able to access himself. Such security loopholes are called *Trojan horses*.

The 80286 designers were guided by the principle of trusting the operating system as little as possible, while supporting sharing and protection. As an example of the use of such protected sharing, suppose a payroll program writes checks and also updates the year-to-date information on total salary and benefits payments. Thus, we want to give the program the ability to read the salary and year-to-date information and modify the year-to-date information but not the salary. We shall see the mechanism to support such features shortly. In the rest of this section we will look at the big picture of the 80286 protection and examine its motivation. Readers interested in the detailed picture can find it in a comprehensive book by Crawford and Gelsinger [1987].

Adding Bounds Checking and Memory Mapping

The first step in enhancing the 80286 was getting the segmented addressing to check bounds as well as supply a base. Rather than a base address, as in the 8086, segment registers in the 80286 contain an index to a virtual memory data structure called a *descriptor table*. Descriptor tables play the role of page tables in the VAX. On the 80286 the equivalent of a page-table entry is a *segment descriptor*. It contains fields found in PTEs:

A *present bit*—equivalent to the PTE valid bit, used to indicate this is a valid translation

A *base field*—equivalent to a page-frame address, containing the physical address of the first byte of the segment

An *access bit*—like the reference bit or use bit in some architectures that is helpful for replacement algorithms

An *attributes field*—like the protection field in the VAX PTE, which specifies the valid operations and protection levels for operations that use this segment

There is also a *limit field*, not found in paged systems, which establishes the upper bound of valid offsets for this segment. Figure 8.30 shows examples of 80286 segment descriptors.

Adding Sharing and Protection

The Intel designers' next step was to provide for protected sharing. Like the VAX, half of the address space is shared by all processes and half is unique to each process, called *global address space* and *local address space*, respectively. Each half is given a descriptor table with the appropriate name. A descriptor pointing to a shared segment is placed in the global-descriptor table, while a descriptor for a private segment is placed in the local-descriptor table.

A program loads an 80286 segment register with an index to the table **and** a bit saying which table it desires. The operation is checked according to the attributes in the descriptor, the physical address being formed by adding the offset in the CPU to the base in the descriptor, provided the offset is less than the limit field. Unlike the encoding of operations and levels in the VAX PTE, every segment descriptor has a separate two-bit field to give the legal access level of this segment. A violation occurs only if the program tries to use a segment with a lower protection level in the segment descriptor.

We can now show how to invoke the payroll program to update the year-to-date information without allowing it to update salaries. The program could be given a descriptor to the information that has the writable field clear, meaning it can read but not write the data. A trusted program can then be supplied that will only write the year-to-date information and is given a descriptor with the writable field set (Figure 8.30). The payroll program invokes the trusted code using a code-segment descriptor with the conforming field set (Figure 8.30). This means the called program takes on the privilege level of the code being called rather than the privilege level of the caller. Hence, the payroll program can read the salaries and call a trusted program to update the year-to-date totals, yet the payroll program cannot modify the salaries. If a Trojan horse exists in this system, to be effective it must be located in the trusted code whose only job is to update the year-to-date information. The argument for this style of protection is that limiting the scope of the vulnerability enhances security.

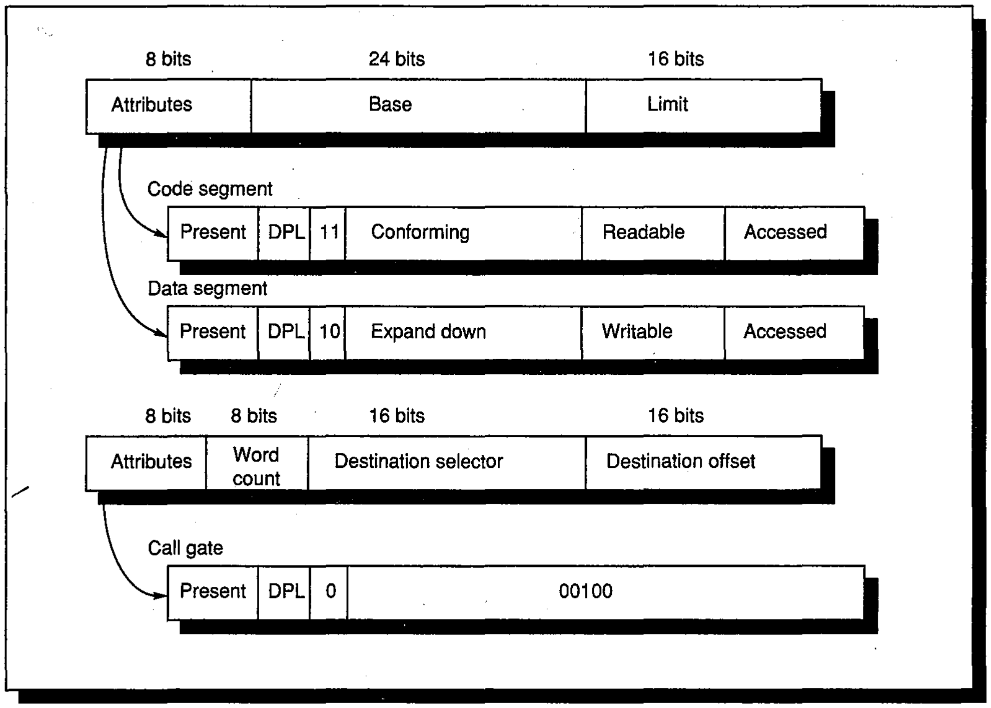

FIGURE 8.30 The 80286 segment descriptors are all 48 bits long and are distinguished by bits in the attributes field. *Base, limit, present, readable,* and *writable* are all self-explanatory. DPL means *descriptor privilege level*—this is checked against the code privilege level to see if the access will be allowed. *Conforming* says the code takes on the privilege level of the code being called rather than the privilege level of the caller; it is used for library routines. The *expand-down field* flips the check to let the base field be the high-water mark and the limit field be the low-water mark. As one might expect, this is used for stack segments that grow down. *Word count* controls the number of words copied from the current stack to the new stack on a call gate. The other two fields of the call-gate descriptor, *destination selector* and *destination offset*, select the descriptor of the destination of the call and the offset into it. There are many more than these three segment descriptors in the 80286. The principal change in the 80386 was to lengthen the base by eight bits and the limit by four bits.

Adding Safe Calls from User to OS Gates and Inheriting Protection Level for Parameters

Allowing the user to jump into the operating system is a bold step. How, then, can a hardware designer increase the chances of a safe system without trusting the operating system or any other piece of code? The 80286 approach is to restrict where the user can enter a piece of code, to safely place parameters on the proper stack, and to make sure the user parameters don't get the protection level of the called code.

To restrict entry into others' code, the 80286 provides a special segment descriptor, or *call gate*, identified by a bit in the attributes field. Unlike other descriptors, call gates are full physical addresses of an object in memory; the offset supplied by the CPU is ignored. As stated above, their purpose is to prevent the user from randomly jumping anywhere into a protected or more- privileged code segment. In our programming example, this means the only place the payroll program can invoke the trusted code is at the proper boundary. This is needed to make conforming segments work as intended.

What happens if caller and callee are "mutually suspicious," so that neither trusts each other? The solution is found in the word-count field in the bottom descriptor in Figure 8.30 (page 447). When a call instruction invokes a call-gate descriptor, the descriptor will copy the number of words specified in the descriptor from the local stack onto the stack corresponding to the level of this segment. This allows the user to pass parameters by first pushing them onto the local stack. The hardware then safely transfers them onto the correct stack. A return from a call gate will pop the parameters off both stacks and copy any return values to the proper stack.

This still leaves open the potential loophole of having the operating system use the user's address, passed as parameters, with the operating system's security level, instead of with the user's level. The 80286 solves this problem by dedicating two bits in every CPU segment register to the *requested protection level.* When an operating system routine is invoked, it can execute an instruction that sets this two-bit field in all address parameters with the protection level of the user that called the routine. Thus, when these address parameters are loaded into the segment registers, they will set the requested protection level to the proper value. The 80286 hardware then uses the requested protection level to prevent any foolishness: No segment can be accessed from the system routine using those parameters if it has a more-privileged protection level than requested.

Summary: Protection on the VAX Versus the 80286

If the 80286 protection model looks harder to build than the VAX model, that's because it is. This effort must be especially frustrating for the 80286 engineers, since most customers just use the 80286 as a fast 8086 and don't exploit the elaborate protection mechanism. Also, the fact that the protection model is a mismatch for the simple paging protection of UNIX means it will be used only by someone writing an operating system specially for this computer. OS/2 from Microsoft is the best candidate, but only time will tell whether the performance cost of such protection is justified for a personal-computer operating system. Two questions remain: Will the considerable protection-engineering effort, which must be borne by each generation of the 80x86 family, be put to good use, and will it prove any safer in practice than a paging system?

8.7 More Optimizations Based on Program Behavior

Making the frequent case fast is the inspiration for almost all inventions aimed at improving performance. In this section are two more examples of hardware optimized to program behavior. The first fetches instructions before they are needed, and the second avoids saving registers to memory on procedure calls.

Instruction-Prefetch Buffers

Many machines use an *instruction-prefetch buffer* to take advantage of the normal sequential execution of instructions. Typically, an instruction buffer contains two to eight sequential instructions; as each instruction is consumed by the CPU, a subsequent instruction word is prefetched. Prefetching only makes sense if the memory system can deliver instructions much faster than the CPU can consume them; otherwise the buffer cannot get ahead of the CPU. This can be accomplished by having a wider path that fetches more than one instruction at a time, or by simply having a faster memory system than the CPU. The drawback to instruction buffers is that they increase memory traffic by requesting words of instructions that may never be needed by the CPU, as is the case when a branch is taken. Instruction-prefetch buffers are also useful for aligning variable-sized instructions.

The 8-byte instruction-prefetch buffer (IB) of the VAX-11/780, shown in Figure 8.31 (page 450), will serve as an example. The opcode of the current instruction is in the high-order byte of the IB; as pieces of the instruction are consumed, the whole buffer is shifted to the left by the appropriate amount. The left-most byte can correspond to any byte address, while the rest of the bytes in the IB must be sequential. The Vs in the figure represent a valid bit per byte of the instruction buffer and indicate the sequential bytes that contain valid instructions.

The IB tries to stay ahead of the PC. Whenever at least one byte is free in the IB, a read is requested for an aligned 32-bit word that contains that byte; only 32-bit words are prefetched from the memory. When the 32-bit prefetched word arrives, the IB loads as much of it as it has space for. A 32-bit instruction word therefore takes between one and four fetches from memory, depending on luck.

When the PC changes due to a branch or interrupt, the IB may have prefetched one or two unneeded instructions. The PC change causes all the valid bits to be turned off, and the IB is reloaded. Section 8.9 examines the performance impact of the IB.

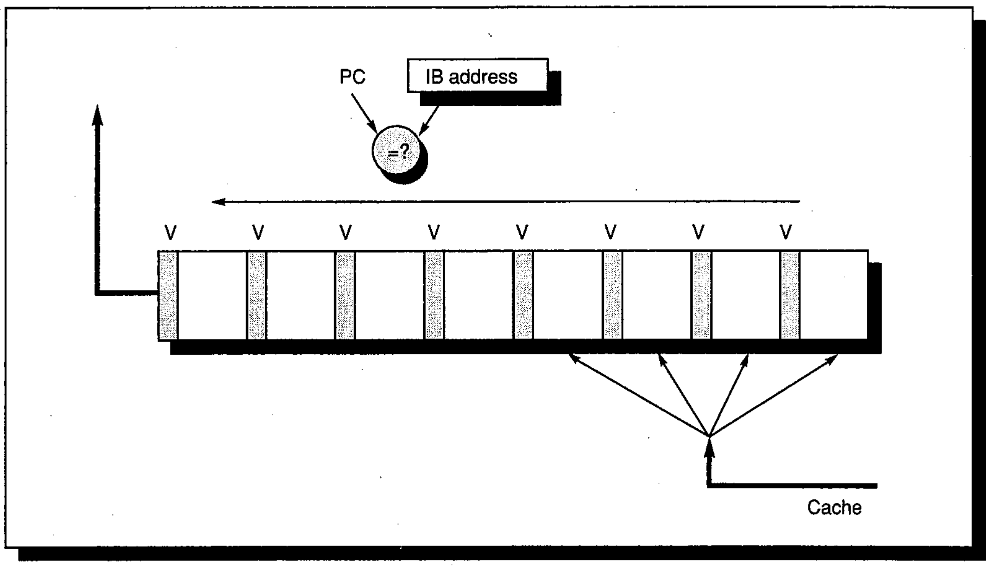

FIGURE 8.31 The VAX-11/780 instruction-prefetch buffer. Every byte has a valid bit to determine the number of consecutive bytes that have valid instructions. The instruction decoder can read the top four bytes of the buffer in a single clock cycle.

Registers and Register Windows

Figures 3.28 and 3.29 (pages 117–118) in Chapter 3 show that saving registers on procedure calls and restoring them on returns can account for 5% to 40% of the data memory references. As an alternative, several banks of registers can be used, with a new one allocated on each call. Although this could limit the depth of procedure calls, the limitation is avoided by operating the banks as a circular buffer, providing unlimited depth. This technique has been termed *register windows*.

Figure 8.32 shows the essence of the idea. On the x axis is time, measured in procedure calls or returns; on the y axis is the depth or nesting of procedure calls. Each call moves down the y axis, and each return moves up. The boxes show memory being accessed to save some of the buffer, either when it is full and is followed by a call (*window overflow*) or when it is empty and is followed by a return (*window underflow*). The figure shows eight window overflows and two window underflows during this section of program execution. Over the life of the program the number of overflows and underflows will equalize.

One might well ask what the trade-off is between buffer size and overflows or underflows. Figure 8.33 shows the shape of the curve for several programs written in several programming languages. The knee of the curve seems to be six to eight banks. While this holds for most programs, the optimization is based on

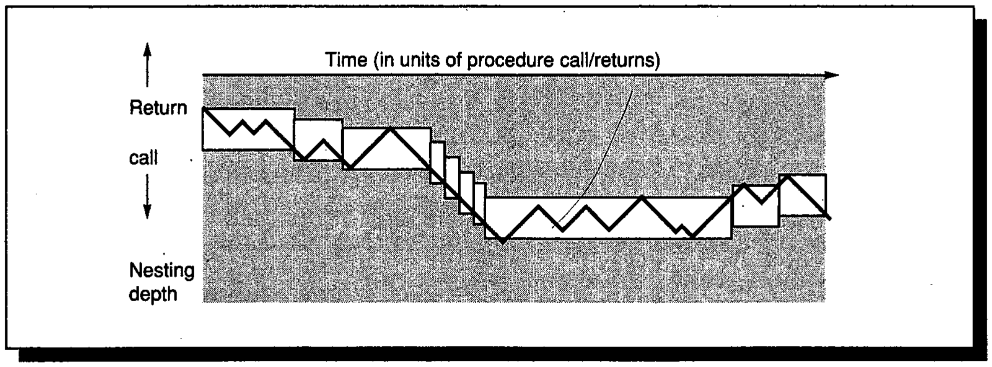

FIGURE 8.32 Change in procedure nesting depth over time. The boxes show procedure calls and returns inside the buffer before a window overflow or underflow. The program starts with three calls, a return, a call, a return, three calls, and then a window overflow.

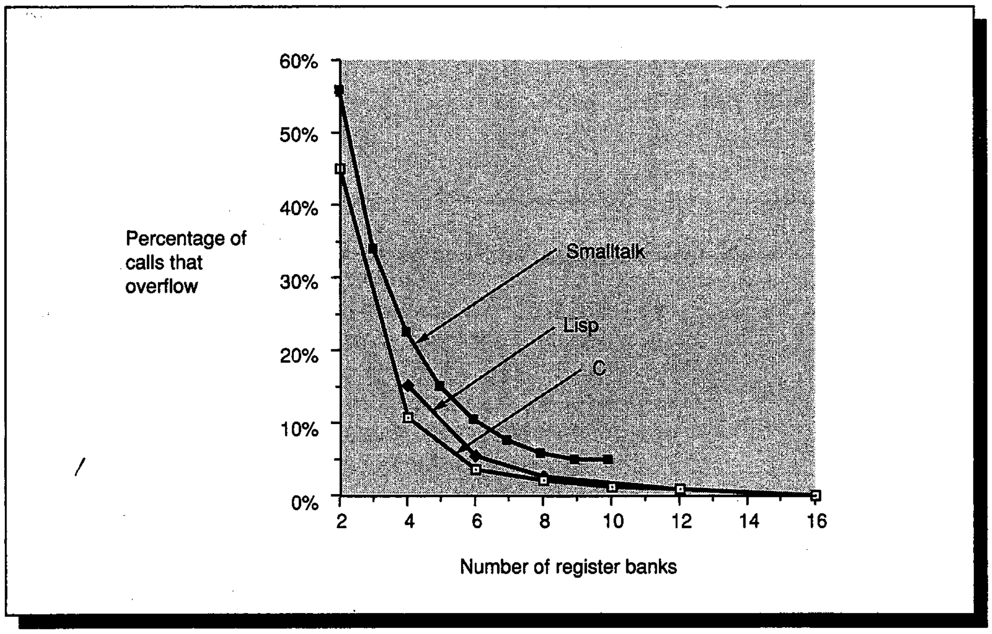

FIGURE 8.33 Number of banks or windows of registers versus overflow rate for several programs in C, LISP, and Smalltalk. The programs measured for C include a C compiler, a Pascal interpreter, troff, a sort program, and a few UNIX utilities [Halbert and Kessler 1980].The LISP measurements include a circuit simulator, a theorem prover, and several small LISP benchmarks [Taylor et al. 1986]. The Smalltalk programs come from the Smalltalk macro benchmarks [McCall 1983] which include a compiler, browser, and decompiler [Blakken 1983 and Ungar 1987].

program-specific patterns of calls and returns that might be quite different in some other programs. The worst case for register windows would be hundreds of calls followed by hundreds of returns. This would make Figure 8.32 look like seismograph output during an earthquake, and the performance impact would be just as devastating!

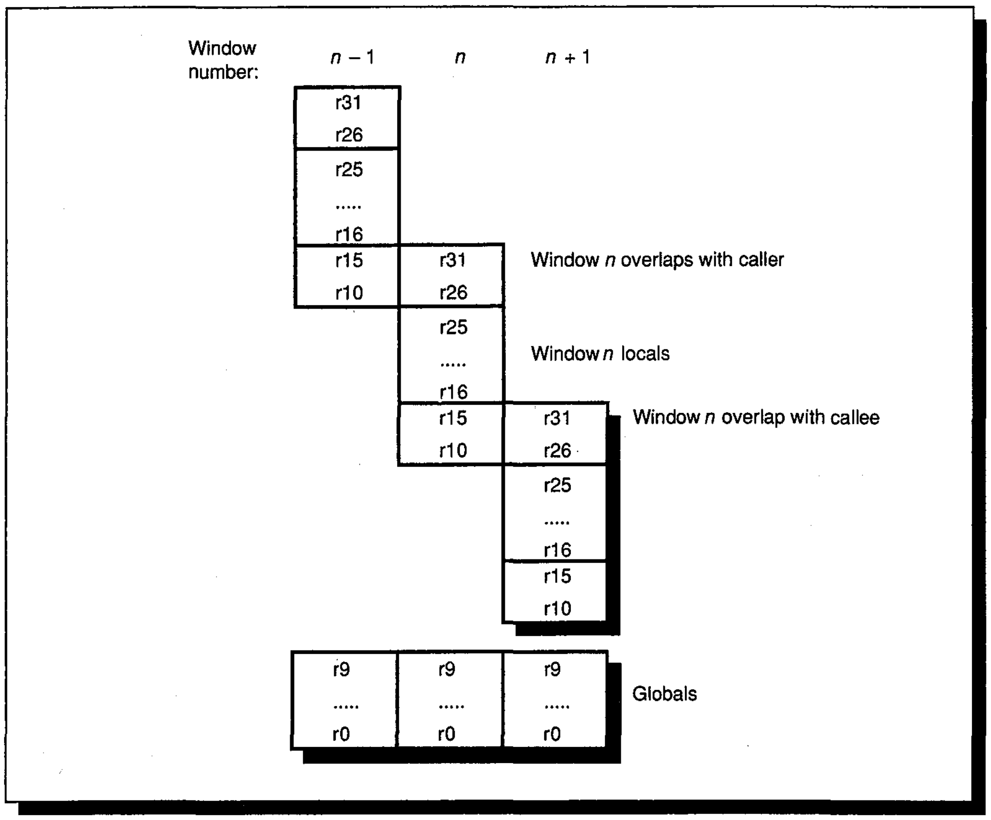

FIGURE 8.34 Parameters can be passed in registers if there are common registers between two banks or windows. This scheme divides registers into globals, which don't change on a procedure call, and locals, which do change. By having an overlap between locals for adjacent procedure calls and renumbering the registers on a call, the outgoing parameters of the caller become the incoming parameters of the callee. For example, a value placed in register 15 before a call is in register 31 after the call.

The difficulty of passing parameters in registers presents a drawback: If each procedure has its own unique set of registers, then nothing is common. This can be overcome by overlapping the register banks or windows such that there is a common area in which to pass parameters. Figure 8.34 shows one such design. Six registers overlap each window, with R15 to R10 of the caller's registers becoming R31 to R26 after the call. Ten registers are not included in the windows, so there are 16 (32 – 10 – 6) registers per window even though each procedure sees 32 registers at a time.

From Figure 8.33 we can estimate the percentage of calls that overflow the windows or returns that underflow them, but to understand the impact on performance we must know the cost an overflow or underflow. With an overlapping register design, like the one on SPARC, the cost is saving 16 registers on an overflow (or restoring 16 registers on an underflow) plus the cost of interrupt. On the Sun 4 today it takes about 60 clock cycles for an overflow or underflow.

The Pros and Cons of Register Windows

Depending on the application, programming language, and user practices, the compiler can close the gap between machines with and without register windows. Most machines, for example, have separate floating-point registers, which means that floating-point-intensive programs will be unaffected by register windows. Also, many data references are to objects that cannot be allocated in registers, like arrays or structures (see Figures 3.28 and 3.29 on pages 117–118 of Chapter 3).

An optimization called *interprocedural register allocation* allows more intelligent allocation of registers across procedure boundaries. Unfortunately, interprocedural register allocation works best when procedures are compiled or linked at the same time. Long compilation and link time do not match the emphasis on a rapid debug-edit-compile cycle in current dynamic languages like LISP and Smalltalk. Interprocedural register allocation is not generally applicable to object-oriented languages like Objective C and Smalltalk because in the dynamic equivalent of a procedure call the compiler doesn't know which procedure will be invoked on such calls. Register windows also simplify some compiler decisions, since there is no extra cost in using a register that will not be saved or restored separately.

	GCC	TeX
Percentage of DLX instructions call or return	1.8%	3.6%
Registers stored per call	2.3	3.2
Loads DLX	3,928,710	2,811,545
Loads SPARC	3,313,317	2,736,979
Ratio loads DLX / SPARC	1.20	1.03
Stores DLX	2,037,226	1,974,078
Stores SPARC	1,246,538	1,401,186
Ratio stores DLX / SPARC	1.60	1.41

FIGURE 8.35 Benefits of register windows on loads and stores for non–floating-point programs. The first row shows the percentage of DLX instructions executed that are calls or returns. The second row shows the average number of register saves and restores per call on the DLX architecture with optimization level O2. The following rows show the total number of loads and stores for each optimization and for the SPARC architecture, which has register windows. The data below includes the loads and stores due to window overflow and window underflow. GCC executes about 20% more loads and 60% more stores on DLX than on a machine with register windows, while TeX executes about 3% more loads and 41% more stores. These savings correspond to about 7% of the instruction count for GCC and 5% for TeX. How this translates into memory-system performance depends on the details of the rest of the memory hierarchy. Interprocedural register allocation closes this gap. For example, using O3 optimization on TeX reduces the number of DLX loads by 5% to 2,671,631 and the number of stores by 10% to 1,791,831. Note that the inputs for these programs were not the same as those used in Chapters 2 or 4. (Spice was not included because register windows offer no benefit for floating-point programs.)

The danger of register windows is that the larger number of registers could slow down the clock rate. So far, this has not been the case for commercial machines. The SPARC architecture (with register windows) and the MIPS R2000 architecture (without) are contemporary machines built in several technologies. The SPARC clock rate has not been slower than MIPS for implementations in similar technologies, probably because cache-access times dominate register-access times in implementations to date of either architecture. A second concern is the impact of register windows on process-switch time. Sun Microsystems has found that UNIX operating system vagaries dominate process-switch time, and less than 20% of the process-switch time is spent on saving or restoring registers. Figure 8.35 (page 453) compares some measures of the benefits of register windows on our benchmark programs.

8.8 Advanced Topics—Improving Cache-Memory Performance

This section covers advanced topics in cache memories, going through new ideas at a much quicker pace than previous sections. The central points of this chapter are not lost if this section is skipped; in fact, the Putting It All Together section that follows is independent of this material.

The increasing gap between CPU and main memory speeds has attracted the attention of many architects. After making some easy decisions in the beginning, the architect faces a threefold dilemma when attempting to further reduce average access time:

- Increasing block size doesn't improve average access time; the lower miss rate doesn't offset the higher miss penalty.
- Making the cache bigger would make it slower, jeopardizing the CPU clock rate.
- Making the cache more associative would also make it slower, again jeopardizing the CPU clock rate.

Moreover, the miss rate calculated from user programs paints too rosy a picture. Figure 8.36 shows the real cache miss rate for a running program, including the operating system code invoked by the programs. This reveals the average access time to be worse than expected.

This section covers a plethora of techniques for improving cache performance: subblock placement, write buffers, out-of-order fetching, virtually addressed caches, two-level caches, and issues relating to cache coherency. The cache-coherency sections include an example of the stale-data problem, a survey of coherency alternatives, an example cache protocol, a synchronization algorithm used in cache coherent multiprocessors, a timeline showing multiprocessor synchronization, and comments about the impact of memory consistency on parallel processors.

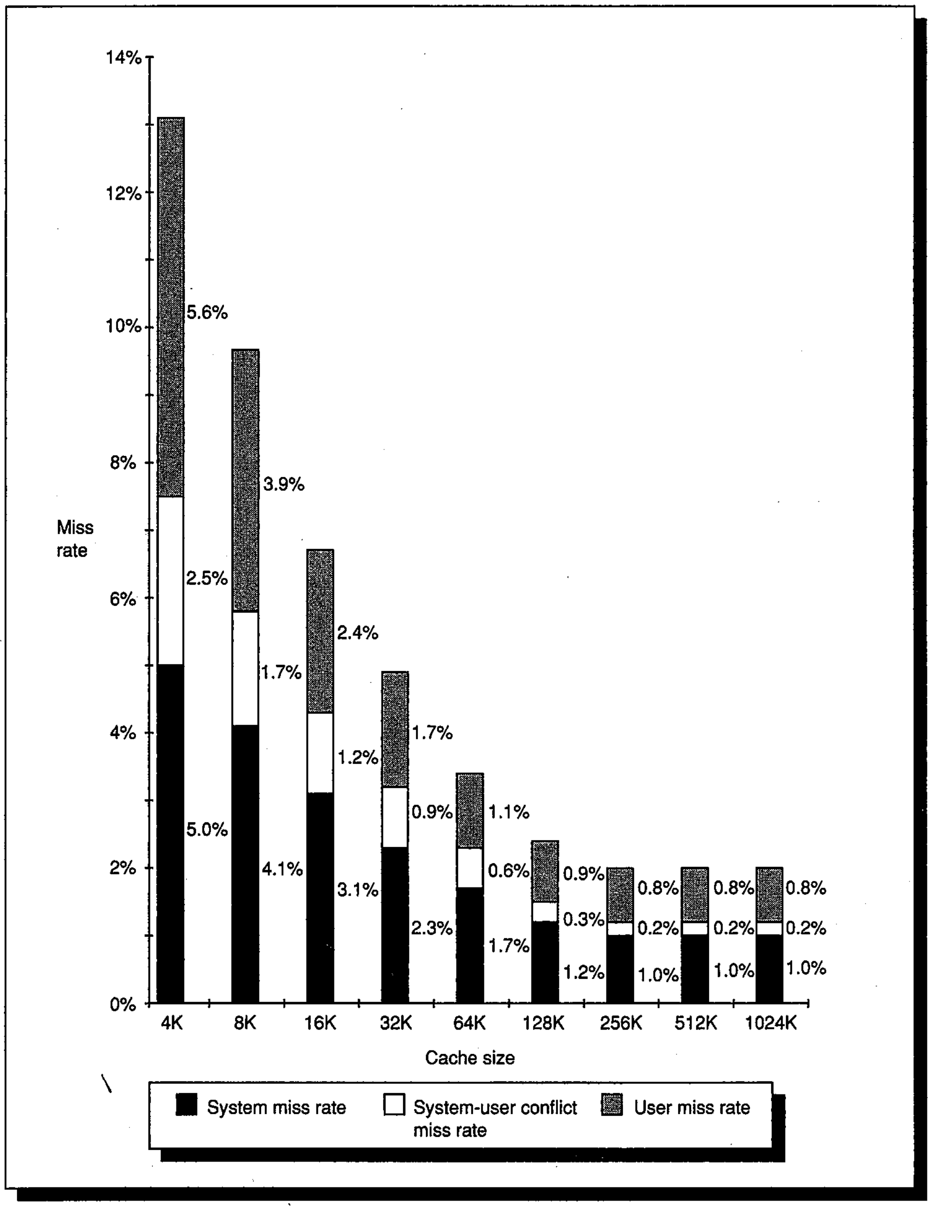

FIGURE 8.36 The miss rate of a program, including the operating system code it invokes, versus cache size. The top category is what would be measured from a user trace; the bottom category is the miss rate for the operating system code; and the middle category is the miss rate due to conflicts between the user code and system code. Agarwal [1987] collected these statistics for the Ultrix operating system running on a VAX, assuming direct-mapped caches with a block size of 16 bytes.

Reducing Hit Times—Making Writes Faster

As mentioned before, writes usually take more than one clock cycle because the tag must be checked before writing the data. There are two ways to do faster writes.

The first, used on the VAX 8800, pipelines the writes for a write-through cache. Tags and data are split so that they can be addressed independently. As usual, the cache compares the tag with the current write address. The difference is that the memory access during this comparison uses the address and data from the **previous** write. Therefore, writes can be performed back to back at one per clock cycle because the CPU does not have to wait for the write to the cache if the first stage is a hit. The 8800 pipeline does not affect read hits—the second stage of the write occurs during the first stage of the next write or during a cache miss.

Another way of reducing writes to one clock cycle involves caches that must be direct mapped, using a technique known as *subblock placement*. Like the VAX-11/780 instruction buffer, there is a valid bit on units smaller than the full block, called *subblocks*. The valid bits specify some parts of the block as valid and some parts as invalid. A match of the tag doesn't mean the word is necessarily in the cache, as the valid bits for that word must also be on. Figure 8.37 gives an example. Note that for caches with subblock placement a block can no longer be defined as the minimum unit transferred between cache and memory. For such caches a block is defined as the unit of information associated with an address tag.

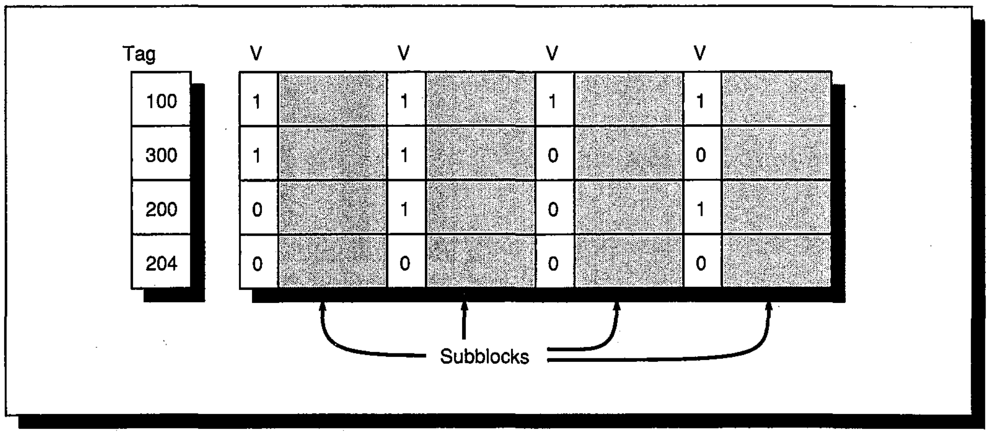

FIGURE 8.37 In this example there are four subblocks per block. In the first block (top) all the valid bits are on, equivalent to the valid bit being on for a block in a normal cache. In the last block (bottom), the opposite is true; no valid bits are on. In the second block, locations 300 and 301 are valid and will be hits, while locations 302 and 303 will be misses. For the third block, locations 201 and 203 are hits. If, instead of this organization, there were 16 blocks the size of the subblock, 16 tags would be needed instead of 4.

Subblock placement was invented to reduce the long miss penalty of large blocks (since only a part of a large block need be read) and to reduce the tag storage for small caches. It can also help write hits by **always** writing the word (no matter what happens with the tag match), turning the valid bit on, and then sending the word to memory. Let's look at the cases to see why this trick works:

- *Tag match and valid bit already set.* Writing the block was the proper action, and nothing was lost by setting the valid bit on again.
- *Tag match and valid bit not set.* The tag match means that this is the proper block; writing the data into the block makes it appropriate to turn the valid bit on.
- *Tag mismatch.* This is a miss and will modify the data portion of the block. However, as this is a write-through cache, no harm was done; memory still has an up-to-date copy of the old value. Only the tag to the address of the write need be changed because the valid bit has already been set. If the block size is one word and the store instruction is writing one word, then the write is complete. When the block is larger than a word or if the instruction is a byte or halfword store, then either the rest of the valid bits are turned off (allocating the subblock without fetching the rest of the block) or memory is requested to send the missing part of the block (write allocate).

This trick isn't possible with a write-back cache because the only valid copy of the data may be in the block, and it could be overwritten before checking the tag.

Reducing Miss Penalty—Making Write Misses Faster

Now that we have seen how to make write hits faster, let's look at write misses. With a write-through cache the most important improvement is a write buffer (page 416) of the proper size (see the fallacy on page 482 in Section 8.10). Write buffers, however, do complicate things in that they might have the updated value of a location needed on a read miss.

Example

Look at this code sequence:

```
SW  512(R0),R3   ; M[512]←R3 (cache index 0)
LW  R1,1024(R0)  ; R1←M[1024] (cache index 0)
LW  R2,512(R0)   ; R2←M[512] (cache index 0)
```

Assume a direct-mapped cache that maps 512 and 1024 to the same block, and a four-word write buffer. Will R3 always equal R2?

Answer

Let's follow the cache to see the danger. The data in R3 is placed into the write buffer after the store. The following load uses the same cache index and is therefore a miss. We then try to load the data from location 512 into register R2; this also results in a miss. If the write buffer hasn't completed writing to location 512 in memory, the read of location 512 will put the old, wrong value into the cache block, and then into R2. Without proper precautions, R3 would not be equal to R2!

The simplest way out of this dilemma is for the read miss to wait until the write buffer is empty. However, a write buffer of a few words in a write-through cache will almost always have data in the buffer on a miss, thereby increasing the read miss penalty. The designers of the MIPS M/1000 estimated that waiting for a four-word buffer to empty would have increased the average read miss penalty by 50%. The alternative is to check the contents of the write buffer on a read miss, and if there are no conflicts and the memory system is available, let the read miss continue.

The cost of writes in a write-back cache can also be reduced. By just adding a full block buffer to store a dirty block, the read can happen first. After the new data is loaded into the block, the CPU continues execution. The buffer then writes in parallel with the CPU. Similar to the situation above, if a read miss occurs the CPU can stall until the buffer is empty.

Reducing Miss Penalty—Making Read Misses Faster

Making writes faster is helpful, but it is reads that dominate cache accesses. The strategy to making read misses faster is to be impatient: Don't wait for the full block to be loaded before sending the requested word to the CPU. Here are two specific strategies:

- *Early restart*—As soon as the requested word of the block arrives, send it to the CPU and let the CPU continue execution.
- *Out-of-order fetch*—Request the missed word first from memory and send it to the CPU as soon as it arrives; let the CPU continue execution while filling the rest of the words in the block. Out-of-order fetch is also called *wrapped* fetch.

Alas, these read tricks are not as important as they sound. Spatial locality—the reason for big blocks in the first place—dictates that the next cache request is likely to be to the same block. Also, handling another request while trying to fill the rest of a block quickly gets complicated.

A more subtle reason why out-of-order fetch will not be as rewarding as one might think is that not all the words of a block have an equal likelihood of being accessed first. With a 16-word block in an instruction cache, for example, the average block entry point is 2.8 words from the left-most byte. If entries were evenly distributed, the average would be 8 words. The high-order word is the most likely one, due to sequential accesses from prior blocks on instruction fetches and sequentially stepping through arrays for data caches.

For pipelined machines that allow out-of-order completion using a scoreboard or Tomasulo-style control (Section 6.7 of Chapter 6), the CPU need not stall on a cache miss, offering another way to reduce memory stalls. Spatial locality suggests this optimization (called a *lock-up free cache*) may be limited in practice, since again the next reference is likely to be to the same block.

Making Cache Hits Faster—Virtually Addressed Caches

Miss penalty is an important part of average access time, but hit time affects both the average access time and the clock rate of the CPU. Helping the hit time may therefore help everything. A solution mentioned earlier is to use the physical part of the address to index the cache while sending the virtual address through the TLB. The limitation is that a direct-mapped cache can be no bigger than the page size. To allow large cache sizes with the 4-KB pages in the System/370, IBM uses high associativity so that they can still access the cache with a physical index. The IBM 3033, for example, is 16-way set associative, even though studies show there is little benefit to miss rates above 4-way set associativity.

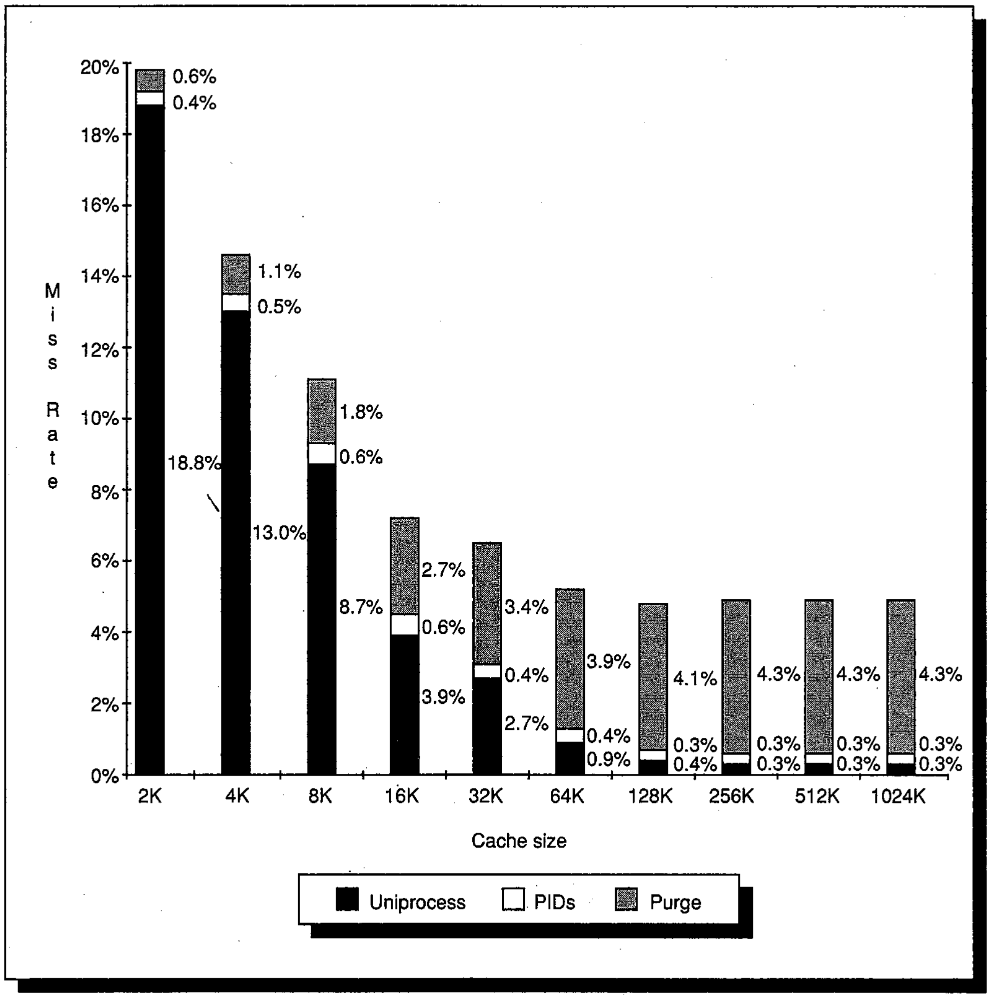

FIGURE 8.38 Miss rate versus cache size of a program measured three ways: without process switches (uniprocess), with process switches using a process-identifier tag (PIDs), and with process switches but without PIDs (purge). PIDs increase the uniprocess absolute miss rate by 0.3 to 0.6 and save 0.6 to 4.3 over purging. Agarwal [1987] collected these statistics for the Ultrix operating system running on a VAX, assuming direct-mapped caches with a block size of 16 bytes.

One scheme for fast cache hits without this size restriction is go to a more heavily pipelined memory access where the TLB is just one step of the pipeline. The TLB is a distinct unit that is smaller than the cache, and thus easily pipelined. This scheme doesn't change memory latency, but relies on the efficiency of the CPU pipeline to achieve higher memory bandwidth.

Another alternative is to match on virtual addresses directly. Such caches are termed *virtual caches*. This eliminates the TLB translation time from a cache hit. Why doesn't everyone build virtually addressed caches? One reason is that every time a process is switched, the virtual addresses refer to different physical addresses, requiring the cache to be flushed. Figure 8.38 (page 459) shows the impact on miss rates of this flushing. One solution is to increase the width of the cache-address tag with a *process-identifier tag* (PID). If the operating system assigns these tags to processes, it only need flush the cache when a PID is recycled (the PID provides protection). Figure 8.38 shows that improvement.

Another reason why virtual caches are not more universally adopted has to do with operating systems and user programs that use two different virtual addresses for the same physical address. These duplicate addresses, called *synonyms* or *aliases*, could result in two copies of the same data in a virtual cache; if one is modified, the other will have the wrong value. With a physical cache this wouldn't happen, since the accesses would first be translated to the same physical cache block. There are hardware schemes, called *anti-aliasing*, that can guarantee every cache block a unique physical address, but software can make this much easier by forcing aliases to share some address bits. The version of UNIX from Sun Microsystems, for example, requires all aliases to be identical in the last 18 bits of their addresses. Thus, a direct-mapped cache that is 2^{18} (256K) bytes or smaller can never have duplicate physical addresses for blocks. This requirement also simplifies anti-aliasing hardware for larger caches or for set-associative caches. (Of course, the best software solution from the hardware designers perspective is to do away with aliases!)

The final area of concern with virtual addresses is I/O. I/O typically uses physical addresses and thus would require mapping to virtual addresses to interact with a virtual cache. (The impact of I/O on caches is further discussed below.)

Reducing Miss Penalty—Two-Level Caches

Let's return our attention to miss penalty. CPUs are getting faster and main memories are getting larger, but slower relative to the faster CPUs. The question facing the architect is: Should I make the cache faster to keep pace with the speed of CPUs, or make the cache larger to overcome the widening gap between the CPU and main memory? One answer is: Both. By adding another level of cache between the original cache and memory, the first-level cache can be small enough to match the clock cycle time of the CPU while the second-level cache can be large enough to capture many accesses that would go to main memory.

Definitions for a second level of cache are not always straightforward. Let's start with the definition of *average memory-access time* for a two-level cache. Using the subscripts L1 and L2 to refer respectively to a first-level and a second-level cache, the original formula is

$$\text{Average memory-access time} = \text{Hit time}_{L1} + \text{Miss rate}_{L1} * \text{Miss penalty}_{L1}$$

and

$$\text{Miss penalty}_{L1} = \text{Hit time}_{L2} + \text{Miss rate}_{L2} * \text{Miss penalty}_{L2}$$

so

$$\text{Average memory-access time} = \text{Hit time}_{L1} + \text{Miss rate}_{L1} * (\text{Hit time}_{L2} + \text{Miss rate}_{L2} * \text{Miss penalty}_{L2})$$

In this formula, the success of the second-level miss rate is measured on the left-overs from the first-level cache. To avoid ambiguity, these terms are adopted here for a two-level cache system:

- *Local miss rate*—The number of misses in the cache divided by the total number of memory accesses to this cache; this is miss rate$_{L2}$ above.
- *Global miss rate*—The number of misses in the cache divided by the total number of memory accesses generated by the CPU; using the terms above, this is miss rate$_{L1}$ * miss rate$_{L2}$.

Example Suppose that in 1000 memory references there are 40 misses in the first-level cache and 20 misses in the second-level cache. What are the various miss rates?

Answer The miss rate for the first-level cache is 40/1000 or 4%. The local miss rate for the second-level cache is 20/40 or 50%. The global miss rate of the second-level cache is 20/1000 or 2%.

Figure 8.39 (page 462) and Figure 8.40 (page 463) show how miss rates and relative execution time change with the size of a second-level cache. Figure 8.41 (page 463) shows typical parameters of second-level caches.

With these definitions in place, we can consider the parameters of second-level caches. The foremost difference between the two levels is that the speed of the first-level cache affects the clock rate of the CPU, while the speed of the second-level cache only affects the miss penalty of the first-level cache. Thus, we can consider many alternatives in the second-level cache that would be ill chosen for the first-level cache. There is but one consideration for the design of the second-level cache: Will it lower the average memory-access–time portion of the CPI?

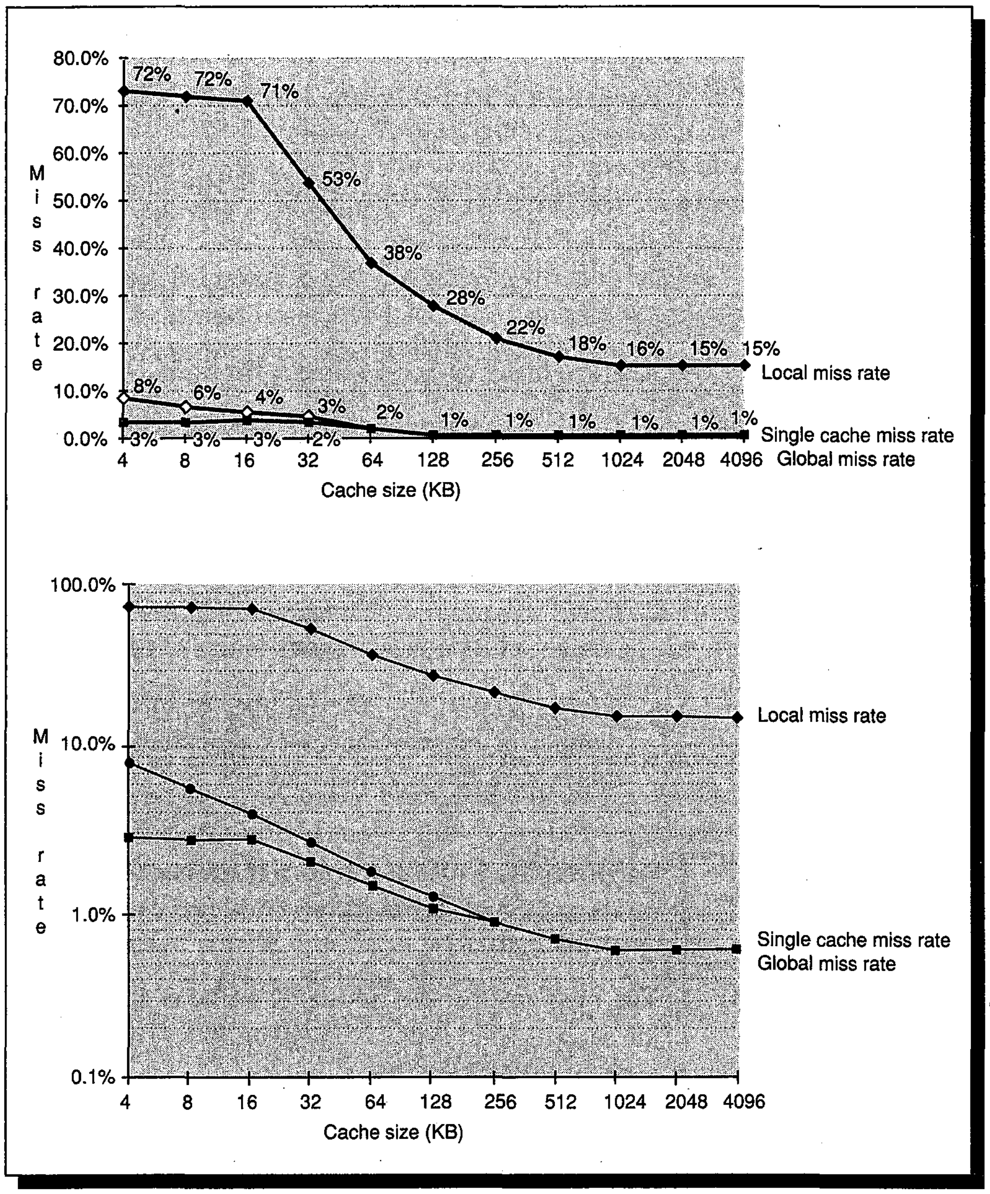

FIGURE 8.39 Miss rates versus cache size. The top graph shows the results plotted on a linear scale as we have done with earlier figures, while the bottom graph shows the results plotted on a log scale. As miss rates shrink the log scale makes the differences easier to follow. The miss rate of a single-level cache versus size is plotted against the local miss rate and global miss rate of a second-level cache using a 32-KB first-level cache. Second-level caches **smaller** than the 32-KB first level have high miss rates (at least for similar block sizes), as this figure illustrates. After 256 KB the single cache and global miss rates are virtually identical. Przybylski [1990] collected these data using traces available with this book: four traces from the VAX system and user programs and four user programs from the MIPS R2000 that were randomly interleaved to duplicate the effect of process switches.

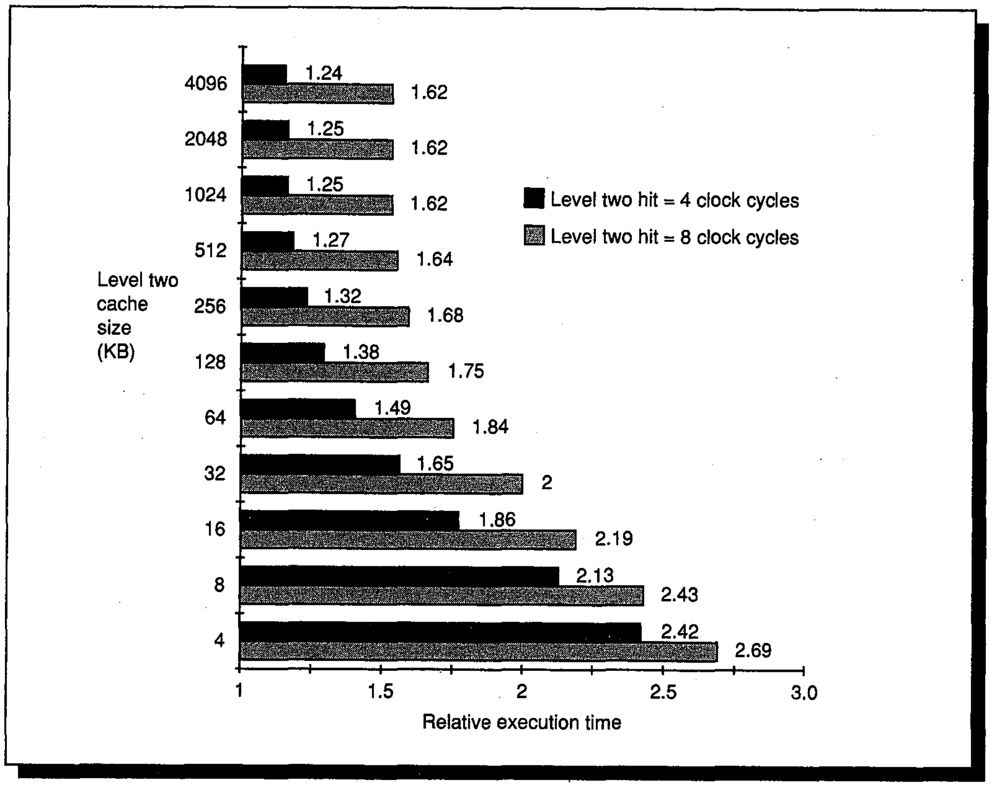

FIGURE 8.40 Relative execution time by second-level–cache size. Przybylski [1990] collected these data using a 32-KB, first-level, write-back cache, varying the size of the second-level cache. The two bars are for different clock cycles for a level two cache hit. The reference execution time of 1.00 is for a 4096-KB, second-level cache with a one-clock-cycle latency on a second-level hit. He used four traces from the VAX system and user programs (available with this book) and four user programs from the MIPS R2000 that were randomly interleaved to duplicate the effect of process switches.

Block (line) size	32 – 256 bytes
Hit time	4 – 10 clock cycles
Miss penalty	30 – 80 clock cycles
(Access time)	(14 – 18 clock cycles)
(Transfer time)	(16 – 64 clock cycles)
Local miss rate	15% – 30%
Cache size	256 KB – 4 MB

FIGURE 8.41 Typical values of key memory-hierarchy parameters for second-level caches.

The initial choice for second-level caches is size. Since everything in the first-level cache is likely to be in the second-level cache, the second-level cache should be bigger. If second-level caches are just a little bigger, the local miss rate will be high. This observation inspires design of huge second-level caches—the size of main memory in recent computers! If the second-level cache is much larger than the first-level cache, then the global miss rate is about the same as a single-level cache of the same size (see Figure 8.39, page 462). Large size means that the second-level cache may have practically no capacity misses, leaving compulsory and a few conflict misses for our attention. One question is whether set associativity makes more sense for second-level caches.

Example

Given the data below, what is the impact of second-level–cache associativity on the miss penalty?

- Two-way set associativity increases hit time by 10% of a CPU clock cycle
- Hit time$_{L2}$ for direct mapped = 4 clock cycles
- Local miss rate$_{L2}$ for direct mapped = 25%
- Local miss rate$_{L2}$ for two-way set associative = 20%
- Miss penalty$_{L2}$ = 30 clock cycles

Answer

For a direct-mapped, second-level cache, the first-level–cache miss penalty is

$$\text{Miss penalty}_{L1} = 4 + 25\% * 30 = 11.5 \text{ clock cycles}$$

Adding the cost of associativity increases the hit cost only 0.1 clock cycles, making the new first-level–cache miss penalty

$$\text{Miss penalty}_{L1} = 4.1 + 20\% * 30 = 10.1 \text{ clock cycles}$$

In reality, second-level caches are almost always synchronized with the first-level cache and CPU. Accordingly, the second-level hit time must be an integral number of clock cycles. If we are lucky, we can shave the second-level hit time to four cycles; if not, we can round up to five cycles. Either choice is an improvement over the direct-mapped, second-level cache:

$$\text{Miss penalty}_{L1} = 4 + 20\% * 30 = 10.0 \text{ clock cycles}$$

$$\text{Miss penalty}_{L1} = 5 + 20\% * 30 = 11.0 \text{ clock cycles}$$

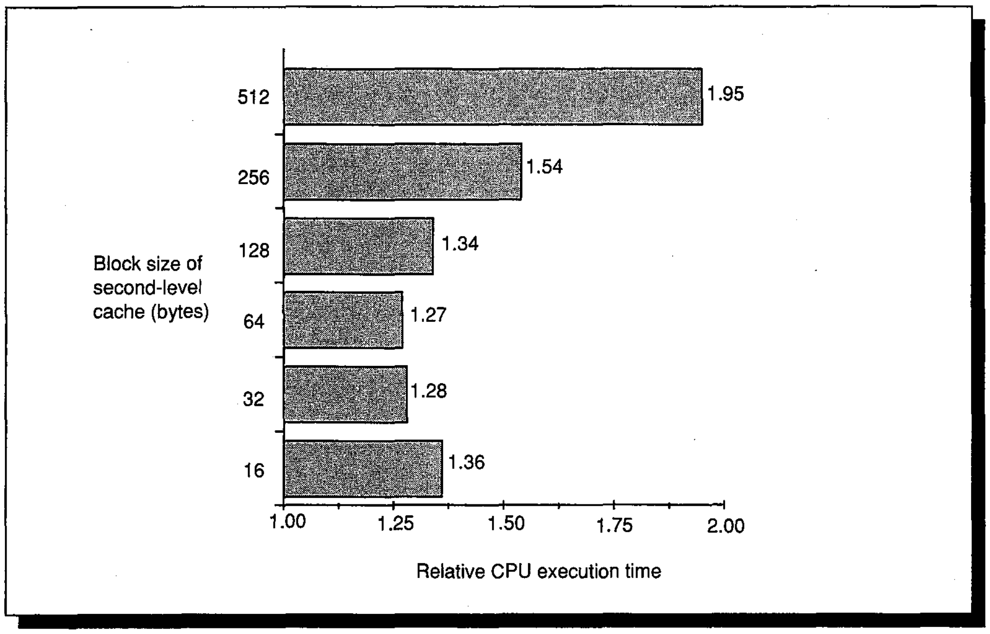

FIGURE 8.42 Relative execution time by block size for a two-level cache. Przybylski [1990] collected these data using a 512-KB second-level cache. He used four traces from the VAX system and user programs (available with this book) and four user programs from the MIPS R2000 that were randomly interleaved to duplicate the effect of process switches.

Higher associativity is worth considering because it has small impact on the second-level hit time and because so much of the average access time is due to misses. However, for these very large caches the benefits of associativity diminish because larger size has eliminated many conflict misses.

As long as spatial locality holds there may be a benefit in increasing block size. Increasing block size can increase conflict misses with small caches since there may not be enough places to put data, therefore increasing miss rate. Because this is not an issue in large, second-level caches, and because memory-access time is relatively longer, larger block sizes are popular. Figure 8.42 shows the variation in execution time as the second-level block size changes.

One final consideration concerns whether all data in the first-level cache is always in the second-level cache. If so, the second-level cache is said to have the *multilevel inclusion property*. Inclusion is desirable because consistency between I/O and caches (or between caches in a multiprocessor) can be determined just by checking the second-level cache.

The drawback to this natural inclusion is that the lower average memory-access times can suggest smaller blocks for the smaller first-level cache and larger blocks for the larger second-level cache. Inclusion can still be maintained in this case with a little extra work on a second-level miss: The second-level cache must invalidate all first-level blocks that map onto the second-level block to be replaced, causing a slightly higher first-level miss rate.

Reducing Miss Rate by Reducing Cache Flushes—I/O

Although there is little more that can improve CPU execution time, there are issues in cache design to improve system performance, particularly for input/output. Because of caches, data can be found in memory or in the cache. As long as the CPU is the sole device changing or reading the data and the cache stands between the CPU and memory, there is little danger in the CPU seeing the old or *stale* copy. I/O means the opportunity exists for other devices to cause copies to be inconsistent or for other devices to read the stale copies. Figure 8.43 illustrates the problem. This is generally referred to as the *cache-coherency* problem.

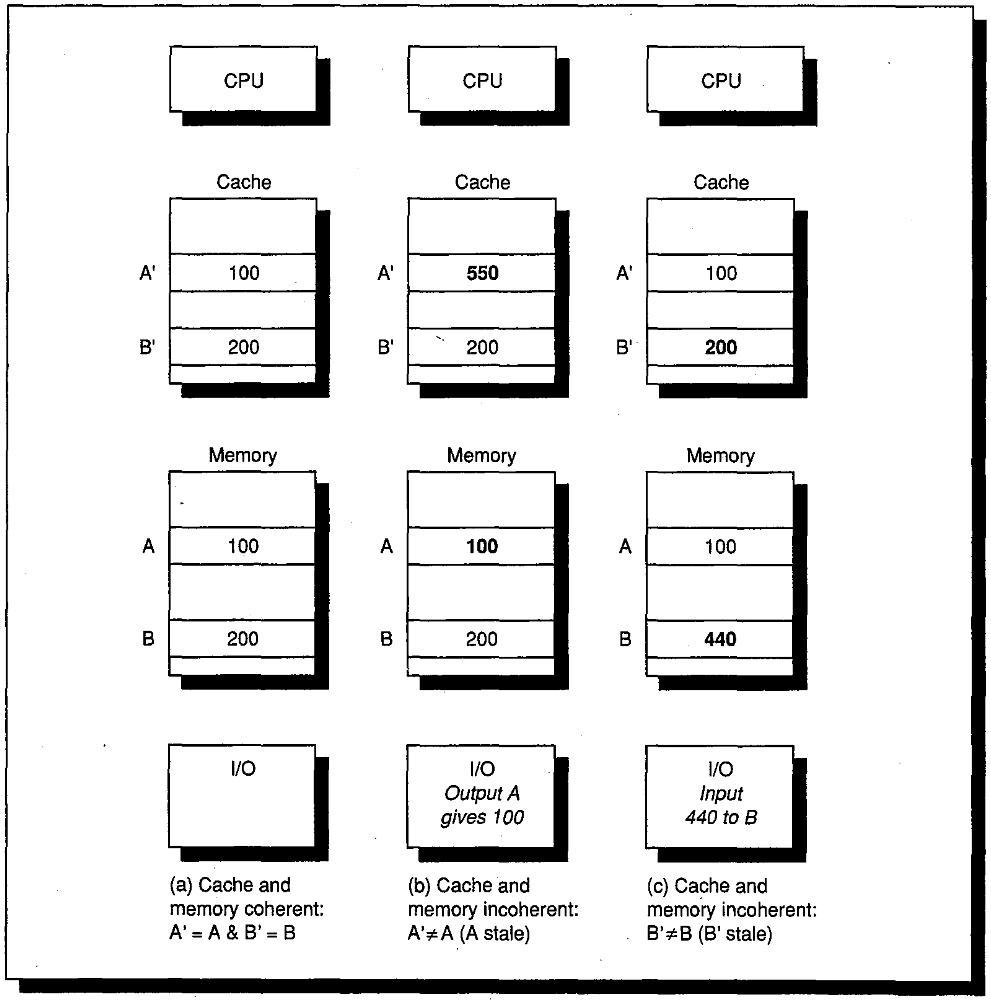
CPU
CPU
CPU
Cache
Cache
Cache
A'
100
B'
200
A'
550
B'
200
A'
100
B'
200
Memory
Memory
Memory
A
100
B
200
A
100
B
200
A
100
B
440
I/O
I/O
Output A
gives 100
I/O
Input
440 to B
(a) Cache and memory coherent: A' = A & B' = B
(b) Cache and memory incoherent: A'≠A (A stale)
(c) Cache and memory incoherent: B'≠B (B' stale)

FIGURE 8.43 The cache-coherency problem. A' and B' refer to the cached copies of A and B in memory. (a) shows cache and main memory in a coherent state. In (b) we assume a write-back cache when the CPU writes 550 into A. Now A' has the value but the value in memory has the old, stale value of 100. If an output used the value of A from memory, it would get the stale data. In (c) the I/O system inputs 440 into the memory copy of B, so now B' in the cache has the old, stale data.

The question is this: Where does the I/O occur in the computer—between the I/O device and the cache or between the I/O device and main memory? If input puts data into the cache and output reads data from the cache, both I/O and the CPU see the same data, and the problem is solved. The difficulty in this approach is that it interferes with the CPU. I/O competing with the CPU for cache access will cause the CPU to stall for I/O. Input will also interfere with the cache by displacing some information with the new data that is unlikely to be accessed by the CPU soon. For example, on a page fault the CPU may need to access a few words in a page, but a program is not likely to access every word of the page if it were loaded into the cache.

The goal for the I/O system in a computer with a cache is to prevent the stale-data problem while interfering with the CPU as little as possible. Many systems, therefore, prefer that I/O occur directly to main memory, acting as an I/O buffer. If a write-through cache is used, then memory has an up-to-date copy of the information, and there is no stale-data issue for output. (This is the reason many machines use write through.) Input requires some extra work. The software solution is to guarantee that no blocks of the I/O buffer designated for input are in the cache. In one approach, a buffer page is marked as noncacheable; the operating system always inputs to such a page. In another approach, the operating system flushes the buffer addresses from the cache after the input occurs. A hardware solution is to check the I/O addresses on input to see if they are in the cache. If so, the cache entries are invalidated to avoid stale data. All these approaches can also be used for output with write-back caches. More about this is found in the next chapter.

Reducing Bus Traffic—Multiprocessor Cache Coherency

The cache-coherency problem applies to multiprocessors as well as I/O. Unlike I/O, where multiple data copies is a rare event—one to be avoided whenever possible—a program running on multiple processors will want to have copies of the same data in several caches. Performance of a multiprocessor program depends on the performance of the system when sharing data. The protocols to maintain coherency for multiple processors are called *cache-coherency protocols*. There are two classes of protocols followed to maintain cache coherency:

- *Directory based*—The information about one block of physical memory is kept in just one location.
- *Snooping*—Every cache that has a copy of the data from a block of physical memory also has a copy of the information about it. These caches are usually on a shared-memory bus, and all cache controllers monitor or *snoop* on the bus to determine whether or not they have a copy of the shared block.

In directory-based protocols there is logically a single directory that keeps the state of every block in main memory. Information in the directory can include which caches have copies of the block, whether it is dirty, and so on. Of course directory entries can be distributed so that different requests can go to different memories, thereby reducing contention. However, they retain the characteristic that the sharing status of a block is always in a single known location.

Snooping protocols became popular with multiprocessors using microprocessors and caches on a shared memory because they can use a preexisting physical connection: the bus to memory. Snooping has an edge over directory protocols in that the coherency information is proportional to the number of blocks in a cache rather than the number of blocks in main memory. Directories, on the other hand, do not require a single bus going to all caches and, hence, may scale to more processors.

The coherency problem is for a processor to have exclusive access to write an object and to have the most recent copy when reading an object. Thus, both directory-based and snooping protocols must locate all the caches that share the object to be written. The consequence of a write to shared data is either to invalidate all other copies or to broadcast the write to the shared copies. Because of write-back caches, coherency protocols must also help read misses determine who has the most up-to-date value.

For the remainder of this section we concentrate on snooping caches; the same ideas apply to directory-based caches except the state of the caches is tracked differently, and caches are involved only if the directory says they have a copy of a block whose status must change.

Sharing information is added to the status bits already in a cache block for snooping protocols, and that information is used in monitoring bus activities. On a read miss all caches check to see if they have a copy of the requested block and take the appropriate action, such as supplying the data to the cache that missed. Similarly, on a write all caches check to see if they have a copy and then act, perhaps invalidating their copy or changing their copy to the new value.

Since every bus transaction checks cache-address tags, one might assume that it interferes with the CPU. It would, were it not for duplicating the address-tag portion of the cache (not the whole cache) to get an extra read port for snooping. This way, snooping interferes with the CPU's access to the cache only when there is a coherency problem (although on a miss with snooping the CPU must arbitrate with the bus to change the snoop tags as well as the normal tags). When a coherency operation occurs in the cache the CPU will likely stall, since the cache is unavailable. In multilevel caches, if the coherency check can be limited to the lower cache because of multilevel inclusion, duplicating the address tags will probably not be necessary.

Snooping protocols are of two types, depending on what happens on a write:

- *Write invalidate*—The writing processor causes all copies in other caches to be invalidated before changing its local copy; it is then free to update the data until another processor asks for it. The writing processor issues an invalidation signal over the bus, and all caches check to see if they have a copy; if so, they must invalidate the block containing the word. Thus, this scheme allows multiple readers but only a single writer.

- *Write broadcast*—Rather than invalidate every block that is shared, the writing processor broadcasts the new data over the bus; all copies are then updated with the new value. This scheme continuously broadcasts writes to shared data while write invalidate deletes all other copies so that there is only one local copy for subsequent writes. Write-broadcast protocols usually allow blocks to be tagged as shared (broadcast) or private (local). One way to think of this protocol is it acts like a write-through cache for shared data (broadcasting to other caches) and a write-back cache for private data (the modified data leaves the cache only on a miss).

Most cache-based multiprocessors use write back caches because it reduces bus traffic and thereby allows more processors on a single bus. Write-back caches use either invalidation or broadcast, and numerous variations exist for both alternatives (see the next section). So far, there is no consensus on which is the superior scheme. Some programs have less coherency overhead with write invalidate, and some with write broadcast. A later section shows how synchronization can be implemented in coherency-based multiprocessors; the accesses for synchronization seem to favor write broadcast.

One early insight has been that block size plays an important role in cache coherency. Take, for example, the case of snooping on a second-level cache with a block size of eight words, and a single word is alternatively written and read by two processors. Whether write invalidation or write broadcast is used, the protocol that only broadcasts or sends a word has an advantage over a scheme that transfers the full block. Another concern of large blocks is called *false sharing*: two different shared variables are located in the same cache block, causing the block to be exchanged between processors even though the processors are accessing different variables. Compiler research is working to reduce cache miss rates by allocating data with high processor locality to the same blocks. Success in this field could increase the desirability of large blocks for multiprocessors.

Measurements to date indicate that shared data has lower spatial and temporal locality than observed for other types of data, independent of the coherency policy.

An Example Protocol

To illustrate the complexities of a cache-coherency protocol, Figure 8.44 (page 470) shows a finite-state transition diagram for a write-invalidation protocol based on write- back policy. The three states of the protocol are duplicated to represent transitions based on CPU actions, as opposed to transitions based on bus operations. This is done only for purposes of this figure; there is only one finite-state machine per cache, with stimuli coming either from the attached CPU or from the bus.

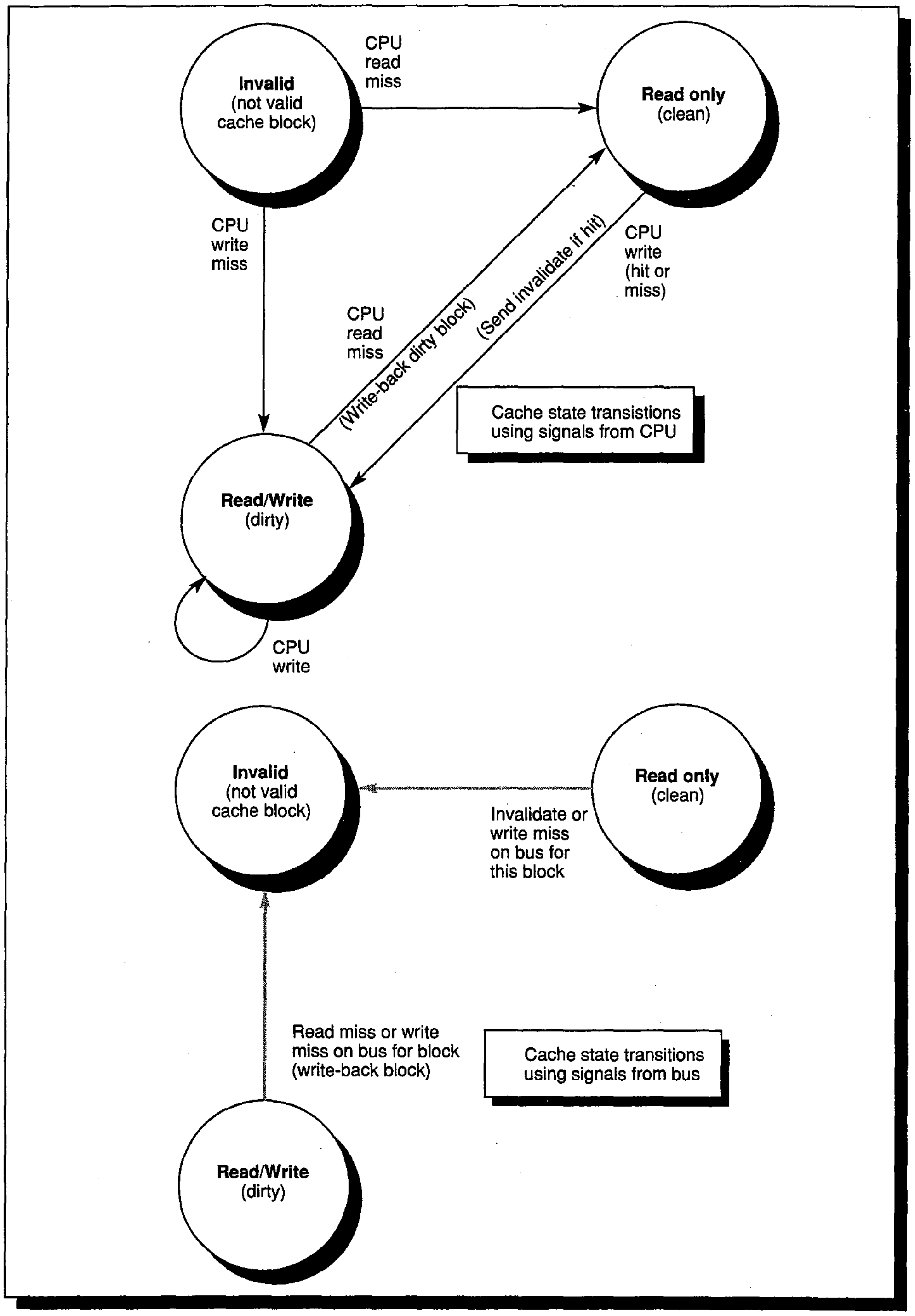
Invalid (not valid cache block)
CPU read miss
Read only (clean)
CPU write miss
CPU read miss
(Write-back dirty block)
(Send invalidate if hit)
CPU write (hit or miss)
Cache state transistions using signals from CPU
Read/Write (dirty)
CPU write
Invalid (not valid cache block)
Read only (clean)
Invalidate or write miss on bus for this block
Read miss or write miss on bus for block (write-back block)
Cache state transitions using signals from bus
Read/Write (dirty)

FIGURE 8.44 A write-invalidate, cache-coherency protocol. The upper part of the diagram shows state transitions based on actions of the CPU associated with this cache; the lower part shows transitions based on operations on the bus. There is only one state machine in a cache, although there are two represented here to clarify when a transition occurs. The black arrows and states would be in a normal cache, with the gray arrows added to get cache coherency. In contrast to what is shown here, some protocols call writes to clean data a "write miss," so that there is no separate signal for invalidation.

Transitions happen on read misses, write misses, or write hits; read hits do not change cache state. When the CPU has a read miss, it will change the state of that block to Read only and write back the old block if it was in the Read/Write state (dirty). All the caches snoop on the read miss to see if this block is in their cache. If one has a copy and it is in the Read/Write state, then the block is written to memory and that block is changed to the invalid state. (An optimization not shown in the figure would be to change the state of that block to Read only.) When a CPU writes into a block, that block goes to the Read/Write state. If the write was a hit, an invalidate signal goes out over the bus. Because caches monitor the bus, all check to see if they have a copy of that block; if they do, they invalidate it. If the write was a miss, all caches with copies go to the invalid state.

As you might imagine, there are many variations on cache coherency that are much more complicated than this simple model. The variations include whether or not the other caches try to supply the block if they have a copy, whether or not the block must be invalidated on a read miss, as well as write invalidate versus write broadcast as discussed above. Figure 8.45 summarizes several snooping cache-coherency protocols.

Name	Category	Memory-write policy	Unique feature
Write Once	Write invalidate	Write back after first write	
Synapse N+1	Write invalidate	Write back	Explicit memory ownership
Berkeley	Write invalidate	Write back	Owned shared state
Illinois	Write invalidate	Write back	Clean private state; can supply data from any cache with a clean copy
Firefly	Write broadcast	Write back for private, Write through for shared	Memory updated on broadcast
Dragon	Write broadcast	Write back for private, Write through for shared	Memory not updated on broadcast

FIGURE 8.45 Six snooping protocols summarized. Archibald and Baer [1986] use these names to describe the six protocols, and Eggers [1989] summarizes the similarities and differences as shown above. Figure 8.44 (page 470) is simpler than any of these protocols.

Synchronization Using Coherency

One of the major requirements of a shared-memory multiprocessor is being able to coordinate processes that are working on a common task. Typically, a programmer will use *lock variables* to synchronize the processes.

The difficulty for the architect of a multiprocessor is to provide a mechanism to decide which processor gets the lock and to provide the operation that locks a variable. Arbitration is easy for shared-bus multiprocessors, since the bus is the only path to memory: The processor that gets the bus locks out all other processors from memory. If the CPU and bus provide an atomic swap operation, programmers can create locks with the proper semantics. The adjective *atomic* is key, for it means that a processor can both read a location **and** set it to the locked value in the same bus operation, preventing any other processor from reading or writing memory.

Figure 8.46 shows a typical procedure for locking a variable using an atomic swap instruction. Assume that 0 means unlocked and 1 means locked. A processor first reads the lock variable to test its state. A processor keeps reading and testing until the value indicates that the lock is unlocked. The processor then races against all other processes that were similarly "spin waiting" to see who

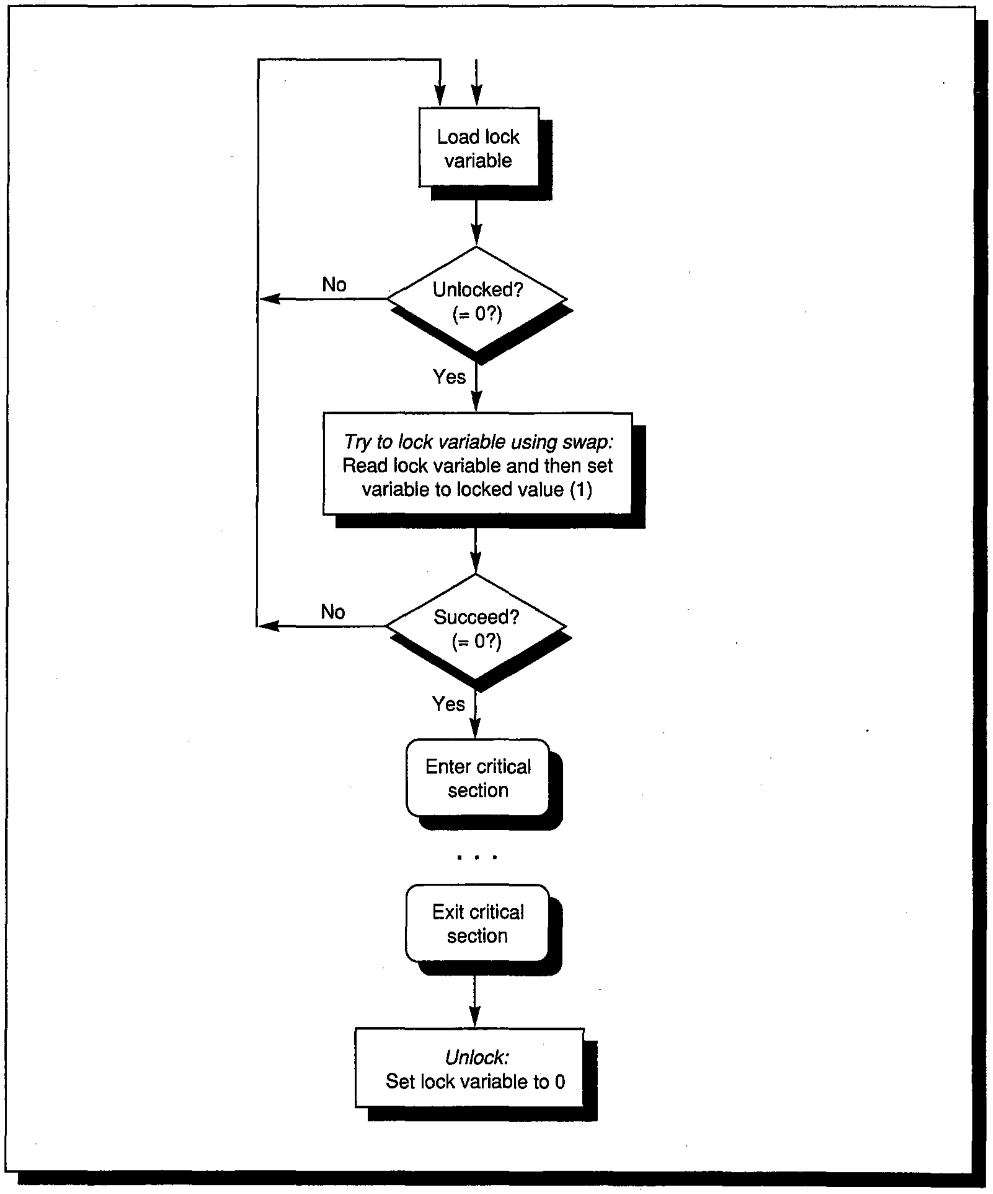
Load lock variable
No
Unlocked? (= 0?)
Yes
Try to lock variable using swap: Read lock variable and then set variable to locked value (1)
No
Succeed? (= 0?)
Yes
Enter critical section
. . .
Exit critical section
Unlock: Set lock variable to 0

FIGURE 8.46 Steps to acquire a lock to synchronize processes and then to release the lock on exit from the key section of code.

Step	Processor P0	Processor P1	Processor P2	Bus activity
1	Has lock	Spins, testing if lock = 0	Spins, testing if lock = 0	None
2	Set lock to 0 and 0 sent over bus			Write invalidate of lock variable from P0
3		Cache miss	Cache miss	Bus decides to service P2 cache miss
4		(Waits while bus busy)	Lock = 0	Cache miss for P2 satisfied
5		Lock = 0	Swap: read lock and set to 1	Cache miss for P1 satisfied
6		Swap: read lock and set to 1	Value from swap = 0 and 1 sent over bus	Write invalidate of lock variable from P2
7		Value from swap = 1 and 1 sent over bus	Enter critical section	Write invalidate of lock variable from P1
8		Spins, testing if lock = 0		None

FIGURE 8.47 Cache-coherency steps and bus traffic for three processors, P0, P1, and P2. This figure assumes write-invalidate coherency. P0 starts with the lock (step 1). P0 exits and unlocks the lock (step 2). P1 and P2 race to see which reads the unlocked value during the swap (steps 3-5). P2 wins and enters the critical section (steps 6 and 7), while P1 spins and waits (steps 7 and 8).

can lock the variable first. All processes use a swap instruction that reads the old value and stores a 1 into the lock variable. The single winner will see the 0, and the losers will see a 1 that was placed there by the winner. (The losers will continue to set the variable to the locked value, but that doesn't matter.) The winning processor executes the code after the lock and then stores a 0 into the lock variable when it exits, starting the race all over again. Testing the old value and then setting to a new value is why the atomic swap instruction is called *test and set* in some instruction sets.

Let's examine how the "spin lock" scheme of Figure 8.46 works with bus-based cache coherency. One advantage of this algorithm is that it allows processors to spin wait on a local copy of the lock in their caches. This reduces the amount of bus traffic versus lock algorithms that loop trying to perform a test and set. (Figure 8.47 shows the bus and cache operations for multiple processes trying to lock a variable.) Once the processor with the lock stores a 0 into the lock, all other caches see that store and invalidate their copy of the lock variable. They then get the new value for the lock of 0. (With write-broadcast cache coherency as on page 469, the caches would update their copy rather than first invalidate and then load from memory.) This new value starts the race to see who can set the lock first. The winner gets the bus and stores a 1 into the lock; the other caches replace their copy of the lock variable containing 0 with a 1. They read that the variable is already locked and must return to testing and spinning. This scheme has difficulty scaling up to many processors because of the communication traffic generated when the lock is released.

Models of Memory Consistency

When we introduce cache coherency to maintain the consistency of multiple copies of an object, we raise a new question: How consistent must the values seen by two processors be kept? The problem is best understood with an example: Here are two code segments from processes P1 and P2 shown side by side:

```
P1:     A = 0;                P2:     B = 0;
         .....                         .....
        A = 1;                        B = 1;
L1:     if (B == 0)  ...      L2:     if (A == 0) ...
```

Assume the processes are running on different processors, and that locations A and B are originally cached by both processors with the initial value of 0. If memory is always consistent, it will be impossible for **both** if statements (labeled L1 and L2) to evaluate their conditions as true (either A=1 or B=1). But suppose write invalidates have a delay, and the processor is allowed to continue during this delay, then it is possible that both P1 and P2 have not seen the invalidations for B and A (respectively) **before** they attempt to read the values. The question that is raised by this example is: How consistent a picture of memory must different processors see?

One approach, called sequential consistency, requires that the result of any execution is the same as if the accesses of each processor were kept in order and the accesses among different processors were arbitrarily interleaved. In this case, the apparent anomaly in the above example cannot occur. Implementing sequential consistency usually requires a processor to delay any memory access until all the invalidations caused by all previous writes are completed. Although this model presents a simple programming paradigm, it reduces potential performance, especially in a machine with a large number of processors, or long interconnect delays.

Alternative models provide a weaker model of memory consistency. For example, the programmer may be required to use synchronization instructions to order memory accesses to the same variable. Now, instead of delaying all accesses until invalidations complete, only synchronization accesses need to be delayed.

Whether programmers expect sequential consistency or some weaker form of consistency is still an open issue in 1990. The example above would work "correctly" with sequential consistency, but not with a weaker model. For weak consistency to produce the same results as sequential consistency, the program would have to be modified to include synchronization operations that order the accesses to variables A and B. It is natural to expect synchronization if you want processes to see the latest data independent of execution rates. Some machines choose to implement sequential consistency as the programming model, while others opt for a weaker consistency. In the future, as attempts are made to build larger multiprocessors, the issue of memory consistency will become increasingly performance critical.

8.9 Putting It All Together: The VAX-11/780 Memory Hierarchy

The challenge for the memory-hierarchy designer is in choosing parameters that work well together, not in inventing new techniques or simulating a cache in a well-understood configuration. A full example using the VAX-11/780 memory hierarchy is presented here in detail to illuminate the interactions. Although VAX-11/780 is not a very recent machine, measurements and design documentation are available on all aspects of its memory hierarchy. Figure 8.48 gives the overall picture.

Let's start with an instruction fetch just after a branch, when the instruction prefetch buffer is empty. The virtual address in the PC is first sent to the TLB. The most significant bit and the lower five bits of the page-frame address index an entry in each bank of the TLB. Including the most-significant bit, used to distinguish system space from process space, guarantees that half of each bank contains system translations and half contains process translations. The addresses in the tags are compared to see if the entry is a match to the page address requested by the TLB. If the valid bit of the entry is not set then there is no match no matter what the tag comparison says, and a miss is indicated.

If there is a match, the physical address is formed by concatenating the physical page-frame address of the TLB page-table entry with the page-offset portion of the address. To save time, the portion of the TLB containing the PTE is read at the same time as the tags, and a 2:1 multiplexer controlled by the tag-matching logic picks the proper PTE. While the address is being formed, the protection bits of the PTE are checked. Since this is an instruction fetch, there is no problem as long as the page can be read by a process at this level. If there are no protection violations, this physical address is sent to the cache.

At the same time the physical address is sent to the cache, two registers in the CPU instruction-prefetch buffer get the new values. The *virtual-instruction-buffer address* register (VIBA) is given the virtual page frame of the PC, and the *physical-instruction-buffer address* register (PIBA) is given the corresponding physical address. This trick, which was originally used in the first machine with virtual memory, avoids the instruction-prefetch buffer's accessing the TLB as long as the instructions are from the same page. The PIBA is actually given the PC address plus 4, so that it can begin prefetching the next instruction. It continues trying to prefetch ahead of the PC until a jump (a frequent occurrence in the VAX) or until the PIBA tries to cross a page boundary; in either case the VIBA and PIBA are no longer used for translating instruction addresses.

Meanwhile, the cache has just received the physical address of the instruction. With 8-byte blocks, a two-way–set-associative cache, and 512 blocks per set, nine bits of the address are needed to index both banks simultaneously. The partial addresses in the tags are compared with the corresponding bits of the physical PC address to see if there is a match. Of course, there are valid bits in each tag that must be turned on, or there can be no match.

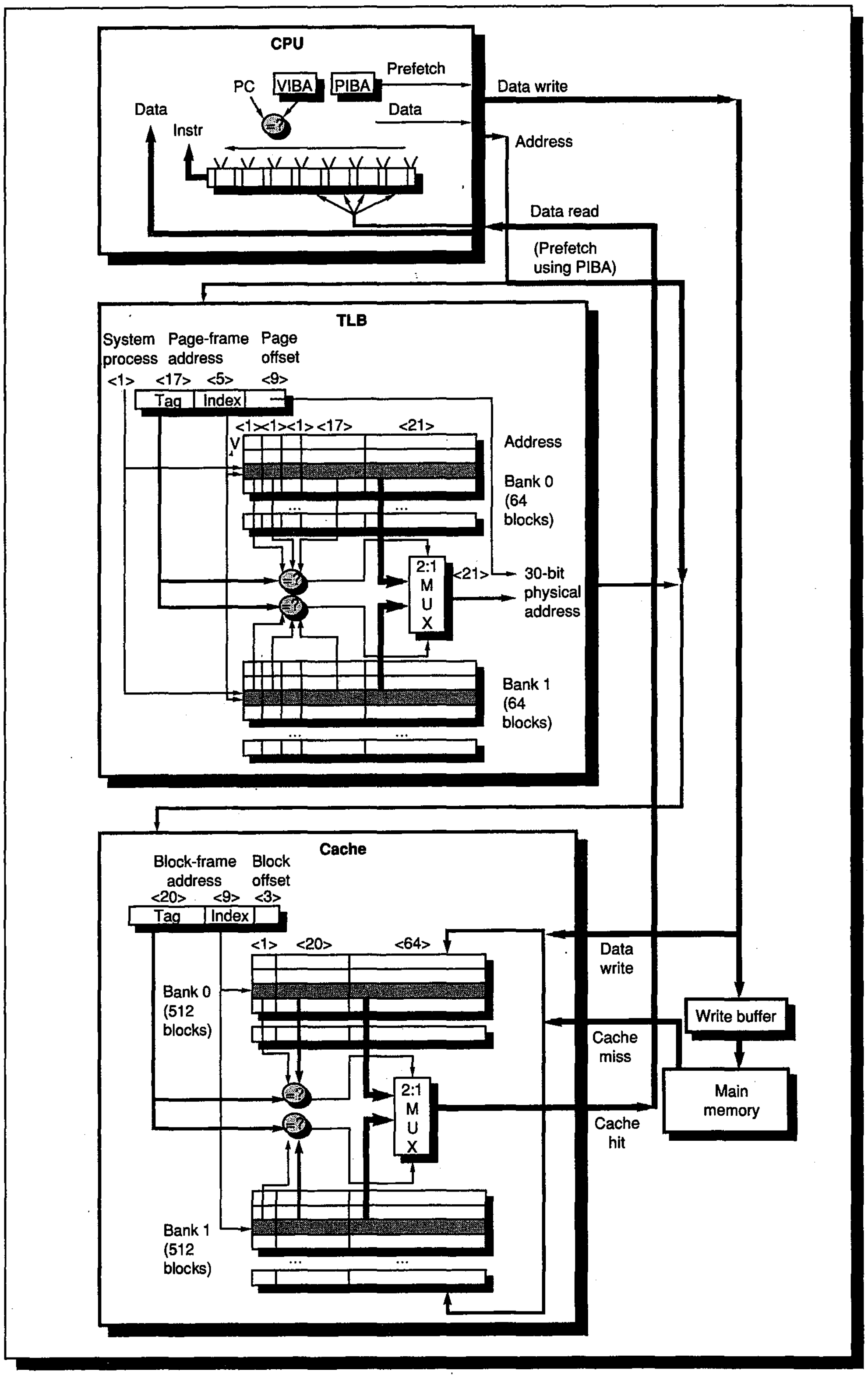

FIGURE 8.48 The overall picture of the VAX-11/780 memory hierarchy. Individual components can be seen in greater detail in Figures 8.11 (page 415), 8.29 (page 444), and 8.31 (page 450).

If there is a match, the lower bits of the physical PC address select the word from the cache block to be sent to the instruction-prefetch unit. Once again, reading data and tags together obviates any additional time delay.

When the word arrives at the prefetch unit, it is placed in the high-order four bytes of the buffer, and those bytes are marked valid. The PIBA immediately begins accessing the cache with the PC address plus 4 to prefetch the next word. As mentioned above, as long as the page-frame address in the PC matches the VIBA, the PIBA bypasses the TLB and goes directly to the cache.

Let's assume this instruction writes a register into memory. The first step will be to send the effective memory address to the TLB for translation. Since this is a write, the modify bit of the matching PTE must also be turned on; this results in a microcode-level trap of the instruction storing the register if the modify bit isn't set already, taking another clock cycle to write the new value in the TLB. The physical address is then sent to the cache. We then go through the same process as before (excluding the read), except that this time it takes an extra clock cycle to modify the portion of the block selected by the write and to write it back into the cache.

In a write-through cache the data must be written to main memory. To avoid the seven-cycle delay of main memory on every write, the VAX-11/780 uses a one-word write buffer. If the buffer is empty, the word is written and the CPU is given the signal to continue. If it is full, the CPU stalls until the buffer is empty.

How well does the 780 work? The bottom line in this evaluation is the percentage of time lost while the CPU is waiting for the memory hierarchy. In one timesharing workload the average number of clock cycles per 780 instruction is 10.6 clock cycles. The breakdown by category is

Compute: 7.3 clock cycles

Read: 0.8 clock cycles

Read stall: 1.0 clock cycles

Write: 0.4 clock cycles

Write stall: 0.4 clock cycles

Instruction-prefetch–buffer stall: 0.7 clock cycles

About 20% of the time the VAX-11/780 stalls while waiting for memory. When the base CPI is 8.5 (compute + read + write), 2.1 clock cycles for the memory hierarchy (read stall + write stall + prefetch stall) may be satisfactory, but it would devastate the performance of a machine with a CPI of 1 to 2.

Let's analyze each unit of the 780 memory hierarchy. An instruction-prefetch–buffer stall means that the buffer is empty, waiting for the cache to supply instructions because of a cache miss, a branch, too many data accesses (they have priority), not enough bytes to decode the instruction, or some combination of the above. The PIBA loadings due to branches versus page crossings vary with the benchmark, but branching is the cause 64% to 91% of the time (median = 76%). The prefetch unit references the cache 2.2 times on average per VAX instruction. The average instruction size is 3.8 bytes, making the effective size of the average prefetch just 1.7 bytes.

Example

Figure 3.33 in Chapter 3 (page 123) shows that the VAX executes many fewer bytes of instructions than DLX. This ignores the instruction-prefetch buffer. How much should we increase the instruction bytes fetched from the cache to include the effect of prefetching?

Answer

We can answer this in a couple of ways. Every prefetch access to the cache actually returns 4 bytes, and the average VAX instruction size is 3.8 bytes; the increase could therefore be

$$\frac{2.2 * 4}{3.8} = 2.32$$

since the prefetch unit references the cache 2.2 times per instruction. This suggests that the bytes fetched from the cache should be increased by 132%. Because the same code may be fetched multiple times by the prefetcher, however, the bandwidth between the cache and memory may not change since the prefetcher cannot cause cache misses.

The question can also be answered in terms of the number of bytes discarded because of a taken branch. About 25% of instructions change the PC on the VAX, and there could be from zero to eight bytes in the prefetch unit when a branch is taken. Assuming an optimistic two bytes, we get a 13% increase:

$$\frac{3.8 + (25\% * 2)}{3.8} = 1.13$$

Assuming six bytes, we get a 39% increase:

$$\frac{3.8 + (25\% * 6)}{3.8} = 1.39$$

While the variable size of VAX instructions does improve the bytes fetched in comparison to DLX, a fairer evaluation of the VAX would increase the bytes fetched from the cache by at least 13% to 39%.

With the instruction-prefetch buffer performing many translations via the PIBA and VIBA, how should TLB misses be measured? The TLB instruction and data-stream miss rates provide one definition:

$$\text{TLB instruction-stream miss rate} = \frac{\text{Misses caused by IB}}{\text{Reloadings of PIBA}}$$

$$\text{TLB data-stream miss rate} = \frac{\text{Misses}}{\text{Requests for 32-bit words of data}}$$

The data-stream definition means references to data objects larger than four bytes count as multiple accesses, as do accesses to unaligned data. Figure 8.49 shows the TLB miss rates.

TLB miss rates	Instruction stream	Data stream	Total
Process	0.7 %	0.6 %	0.7 %
System	15.4 %	5.4 %	7.2 %
Total	3.5 %	1.6 %	1.9 %

FIGURE 8.49 Miss rates for the VAX-11/780 TLB, ignoring the impact of instructions not translated by the TLB. This data was measured on a different timesharing workload than earlier VAX measurements [Clark and Emer 1985].

Overall references to the TLB after filtering by the PIBA are divided into 20% user instruction stream, 62% user data stream, 3% system instruction stream, and 15% system data stream. To account for the filtering of addresses by the PIBA optimization, TLB misses can also be counted as a rate per instruction executed, as in Figure 8.50.

TLB misses per 100 instructions	Instruction stream	Data stream	Total
Process	0.18	0.50	0.68
System	0.62	1.03	1.65
Total	0.80	1.53	2.33

FIGURE 8.50 Misses per hundred instructions for the VAX-11/780 TLB. Unlike Figure 8.49, this overall TLB evaluation accounts for the effect of the PIBA.

The VAX TLB spends on average 21.6 clock cycles on a miss (including 3.5 clock cycles for cache misses for some page-table entries), adding a total of 0.7 clock cycles per instruction for TLB misses to the average instruction. Thus, about a third of the memory-system stalls are due to TLB misses.

The same study by Emer and Clark [1984] showed a significant variation on cache miss rates:

- Data-stream, cache miss rates varied over the day from 12% to 25%, with a mean of 17%.
- Instruction-buffer–stream, cache miss rates varied from 4% to 13%, with a mean of 8%.
- The distribution of accesses to the cache from the CPU was instruction-prefetch–buffer–stream reads, 68%, data-stream reads, 20%, and data-stream writes, 12%. Calculated per instruction, there are about 2.2 references from the instruction-prefetch buffer, 0.8 data reads per instruction, and 0.4 data writes per instruction.

Example According to the VAX-11/780 Architecture Handbook, for the workload measured in 1978 the TLB miss rate was about 3%. What do the measurements say for the timesharing workload measured in 1984?

Answer Assuming just one memory reference to get the average VAX instruction of 3.8 bytes, the miss rate is 1%:

$$\frac{\dfrac{2.3 \text{ TLB misses}}{100 \text{ instructions}}}{\dfrac{1{+}0.8{+}0.4 \text{ references}}{\text{Instruction}}} = \frac{2.3}{100{*}2.2} = 0.01$$

Including the VIBA-PIBA, Figure 8.49 on page 479 shows a 1.9% miss rate.

Example According to the VAX-11/780 Architecture Handbook, for the workload measured in 1978 the cache miss rate was about 5%. What do the measurements say for the timesharing workload measured in 1984?

Answer The cache miss rate varies. The mean miss rate is

$$68\%{*}8\% + 20\%{*}17\% + 12\%{*}17\% = 11\%$$

In the best case, the answer is

$$68\%{*}4\% + 20\%{*}12\% + 12\%{*}12\% = 7\%$$

In the worst case,

$$68\%{*}13\% + 20\%{*}25\% + 12\%{*}25\% = 17\%$$

8.10 Fallacies and Pitfalls

As the most naturally quantitative of the computer architecture disciplines, memory hierarchy would seem to be less vulnerable to fallacies and pitfalls. Yet the authors were limited here not by lack of warnings, but by space.

Pitfall: Too small an address space.

Just five years after DEC and Carnegie-Mellon University collaborated to design the new PDP-11 computer family, it was apparent that their creation had a fatal flaw. An architecture announced by IBM six years **before** the PDP-11 is still thriving, with minor modifications, 25 years later. And the DEC VAX, criticized for including unnecessary functions, has sold 100,000 units since the PDP-11 went out of production. Why?

The fatal flaw of the PDP-11 was the size of its addresses as compared to the IBM 360 and the VAX. Address size limits the program length, since the size of a program and the amount of data needed by the program must be less than $2^{\text{address size}}$. The reason the address size is so hard to change is that it determines the minimum width of anything that can contain an address: PC, register, memory word, and effective-address arithmetic. If there is no plan to expand the address from the start, then the chances of successfully changing address size are so slim that it normally means the end of that computer family. Bell and Strecker [1976] put it like this:

There is only one mistake that can be made in computer design that is difficult to recover from—not having enough address bits for memory addressing and memory management. The PDP-11 followed the unbroken tradition of nearly every known computer. [p. 2]

A partial list of successful machines that eventually starved to death for lack of address bits includes the PDP-8, PDP-10, PDP-11, Intel 8080, Intel 8086, Intel 80186, Intel 80286, AMI 6502, Zilog Z80, CRAY-1, and CRAY X-MP.

Fallacy: Given the hardware resources, the computer designer who selects a set-associative cache over a direct-mapped cache of the same size will get a faster computer.

The question here is whether the extra logic of the set-associative cache affects the hit time, and therefore possibly the CPU clock rate. (See Figure 8.11.) If it does affect hit time, then the question is whether the advantage in lower miss rate offsets the slower hit time. In the mid-1980s many recognized this danger and selected direct-mapped placement; for example, the MIPS M/500, Sun 3/260, and VAX 8800. Hill [1988] makes an eloquent case for direct-mapped caches, including lower costs, faster hit times, and therefore smaller average access times for large, direct-mapped caches. Direct-mapped caches also allow the data read to be sent to the CPU and used even before hit/miss is determined, particularly useful with a pipelined CPU. Hill found about a 10% difference in hit times for TTL or ECL board-level caches and 2% difference for custom CMOS caches, with an absolute change in the miss rates of less than 1% for large caches. Since a direct-mapped cache hit can be accessed faster and hit time typically sets the clock cycle time of the processor, a CPU with a direct-mapped cache can be as fast as or faster than a CPU with a two-way–set-associative cache of the same size. Przybylski, Horowitz, and Hennessy [1988] show several examples of such tradeoffs.

Fallacy: A memory system can be designed using traces from a different architecture.

Figure 8.51 (page 482) shows instruction and data cache miss rates for the same programs on two different architectures. This data is from the first portion of execution of Spice on DLX and the VAX. The shift from data accesses in the

VAX to instruction accesses on DLX seen in Figure 3.33 (page 123) of Chapter 3 is reflected here: 61% of the VAX references and 52% of the misses are to data. Note that while DLX has only three-quarters of the absolute number of data misses, its data miss **rate** is three times higher.

	VAX	DLX
Instruction references	576,169	918,537
Instruction misses	2,033	3,188
Instruction miss rate	0.4%	0.3%
Data references	923,831	264,453
Data misses	2,200	1,595
Data miss rate	0.2%	0.6%
Total references	1,500,000	1,182,990
Percentage of instructions of total references	38%	78%
Total misses	4,233	4,782
Percentage of instruction misses of total misses	48%	67%
Average miss rate	0.3%	0.4%

FIGURE 8.51 Miss rates for VAX and DLX for an initial phase of Spice. The simulation assumes separate instruction and data caches. Each cache is direct mapped, uses 16-byte blocks, and contains 64 KB. Both use write through with write allocate. (Note that unlike Chapter 2, this data was collected using the F77 compiler and was for a portion of the Spice program).

Pitfall: Basing the size of the write buffer on the speed of memory and the average mix of writes.

This seems like a reasonable approach:

$$\text{Write-buffer size} = \frac{\text{Memory references}}{\text{Clock cycle}} * \text{Write percentage} * \text{Clock cycles to write memory}$$

If there is one memory reference per clock cycle, 10% of the memory references are writes, and writing a word of memory takes 10 cycles, then a one-word buffer is added (1*10%*10=1). Calculating for the VAX-11/780 using data from the last section,

$$\frac{3.4 \text{ memory references}}{10.6 \text{ clock cycles}} * \frac{0.4 \text{ writes}}{3.4 \text{ memory references}} * \frac{6 \text{ clock cycles}}{\text{Write}} = 0.22$$

Thus, a one-word buffer seems sufficient.

The pitfall is that when writes come close together, the CPU must stall until the prior write is completed. The single-word write buffer of the VAX-11/780 is the major reason for its write stalling (about 20% of all stalls). The proper question to ask is how large a buffer is needed to keep CPU write stalls to a small amount. The impact of write-buffer size can be established by simulation or estimated with a queuing model.

Pitfall: Extending an address space by adding segments on top of a flat address space.

During the 1970s, many programs grew to the point they couldn't address all of the code and data with just a 16-bit address. Machines were then revised to offer 32-bit addresses, either through a flat 32-bit address space or by adding 16 bits of segment to the existing 16-bit address. From the point of view of marketing, adding segments solves the addressing problem. Unfortunately, there is trouble any time a programming language wants an address that is larger than one segment, such as indices for large arrays, unrestricted pointers, or reference parameters. Moreover, adding segments can turn every address into two words—one for the segment number and one for the segment offset—causing problems in the use of addresses in registers. In the 1990s, 32-bit addresses will be exhausted, and it will be interesting to see if history will repeat itself on the consequences of going to larger flat addresses versus adding segments.

Fallacy: Caches are as fast as registers.

This fallacy is important, because if caches were as fast as registers, there would be no need for registers. Without registers there would be no need for a register allocator, and so compilers could be simpler. The fallacy is difficult to prove quantitatively, yet example after example can be cited. Lampson [1982] summarized this experience:

A register bank is faster than a cache, both because it is smaller, and because the address mechanism is much simpler. Designers of high performance machines have typically found it is possible to read one register and write another in a single cycle, while two cycles [latency] are needed for a cache access. ... Also, since there are not too many registers it is feasible to duplicate or triplicate them, so that several registers can be read out simultaneously.
[p. 74]

As mentioned in Chapter 3, the short addresses of registers allow more compact instruction encoding. It seems to the authors that the deterministic access of multiported register banks will always offer lower latency or higher bandwidth, or both, when compared to the nondeterministic access of caches.

8.11 Concluding Remarks

The difficulty of building a memory system to keep pace with faster CPUs is underscored by the fact that the raw material for main memory is the same as that found in the cheapest computer. It is the principle of locality that saves us here—its soundness is demonstrated at all levels of the memory hierarchy in current computers, from disks to instruction buffers.

	Register windows	**Instruction-prefetch buffer**	**TLB**	**First-level cache**	**Second-level cache**	**Virtual memory**
Block size	64 bytes	1 byte	4 – 8 (1 PTE)	4 – 128 bytes	32 – 256 bytes	512 – 8192 bytes
Hit time	1 clock cycle	1 clock cycle	1 clock cycle	1 – 4 clock cycles	4 – 10 clock cycles	1 – 10 clock cycles
Miss penalty	32 – 64 clock cycles	2 – 6 clock cycles	10 – 30 clock cycles	8 – 32 clock cycles	30 – 80 clock cycles	100,000 – 600,000 clock cycles
Miss rate (local)	1% – 3%	10% – 25%	0.1% – 2%	1% – 20%	15% – 30%	0.00001% – 0.001%
Size	512 bytes	6 – 12 bytes	32 – 8192 (8 – 1024 PTEs)	1 KB – 256 KB	256 KB – 4 MB	4 MB – 2048 MB
Backing store	First-level cache	First-level cache	First-level cache	Second-level cache	Static-column DRAM	Disks
Q1: block placement	Circular buffer	N.A. (Queue)	Set associative	Direct mapped	Set associative	Fully associative
Q2: block identification	2 registers: high and low	Valid bits + 1 register	Tag/ block	Tag/ block	Tag/ block	Table
Q3: block replacement	First in–first out	N.A. (Queue)	Random	N.A. (Direct mapped)	Random	LRU
Q4: write strategy	Write back	Flush on write to instruction buffer (if possible)	Flush on write to page table	Write through or write back	Write through or write back	Write back

FIGURE 8.52 Summary of the memory-hierarchy examples in this chapter.

Misses in every level can be categorized by three causes—compulsory, capacity, and conflict—and different techniques work for each case. Figure 8.52 summarizes the attributes of the memory-hierarchy examples described in this chapter.

There tends to be a knee in the curve of memory-hierarchy cost/performance: Above that knee is wasted performance and below that knee is wasted hardware. Architects find that knee by simulation and quantitative analysis.

8.12 Historical Perspective and References

While the pioneers of computing knew of the need for a memory hierarchy and coined the term, the automatic management of two levels was first proposed by Kilburn, et al. [1962] and demonstrated with the Atlas computer at the University of Manchester. This was the year **before** the IBM 360 was announced. While IBM planned for its introduction with the next generation (System/370), the operating system wasn't up to the challenge in 1970. Virtual memory was announced for the 370 family in 1972, and it was for this machine that the term "translation-lookaside buffer" was coined (see Case and Padegs [1978]). The only computers today without virtual memory are a few supercomputers and personal computers.

Both the Atlas and the IBM 360 provided protection on pages, and over time machines evolved more elaborate mechanisms. The most elaborate mechanism was capabilities, which reached its highest interest in the late 1970s and early 1980s [Fabry 1974 and Wulf, Levin, and Harbison 1981]. Wilkes [1982], one of the early workers on capabilities, had this to say about capabilities:

Anyone who has been concerned with an implementation of the type just described [capability system], or has tried to explain one to others, is likely to feel that complexity has got out of hand. It is particularly disappointing that the attractive idea of capabilities being tickets that can be freely handed around has become lost ....

Compared with a conventional computer system, there will inevitably be a cost to be met in providing a system in which the domains of protection are small and frequently changed. This cost will manifest itself in terms of additional hardware, decreased runtime speed, and increased memory occupancy. It is at present an open question whether, by adoption of the capability approach, the cost can be reduced to reasonable proportions.

Today there is little interest in capabilities either from the operating systems or the computer architecture communities, although there is growing interest in protection and security.

Bell and Strecker [1976] reflected on the PDP-11 and identified a small address space as the only architectural mistake that is difficult to recover from. At the time of the creation of PDP-11, core memories were increasing at a very slow rate, and the competition from 100 other minicomputer companies meant that DEC might not have a cost-competitive product if every address had to go through the 16-bit datapath twice. Hence, the decision to add just 4 more address bits than the predecessor of the PDP-11. The architects of the IBM 360 were aware of the importance of address size and planned for the architecture to extend to 32 bits of address. Only 24 bits were used in the IBM 360, however, because the low-end 360 models would have been even slower with the larger addresses. Unfortunately, the architects didn't reveal their plans to the software people, and the expansion effort was foiled by programmers who stored extra information in the upper eight "unused" address bits.

A few years after the Atlas paper, Wilkes published the first paper describing the concept of a cache [1965]:

The use is discussed of a fast core memory of, say, 32,000 words as slave to a slower core memory of, say, one million words in such a way that in practical cases the effective access time is nearer that of the fast memory than that of the slow memory. [p. 270]

This two-page paper describes a direct-mapped cache. While this is the first publication on caches, the first implementation was probably a direct-mapped instruction cache built at the University of Cambridge. It was based on tunnel diode memory, the fastest form of memory available at the time. Wilkes states that G. Scarott suggested the idea of a cache memory.

Subsequent to that publication, IBM started a project that led to the first commercial machine with a cache, the IBM 360/85 [Liptay 1968]. Gibson [1967] describes how to measure program behavior as memory traffic as well as miss rate and shows how the miss rate varies between programs. Using a sample of 20 programs (each with 3,000,000 references!), Gibson also relied on average memory-access time to compare systems with and without caches. This was over 20 years ago, and yet many used miss rates until recently.

Conti, Gibson, and Pitkowsky [1968] describe the resulting performance of the 360/85. The 360/91 outperforms the 360/85 on only 3 of the 11 programs in the paper, even though the 360/85 has a slower clock cycle time (80 ns versus 60 ns), smaller memory interleaving (4 versus 16), and a slower main memory (1.04 µsec versus 0.75 µsec). This is the first paper to use the term "cache." Strecker [1976] published the first comparative cache-design paper examining caches for the PDP-11. Smith [1982] later published a thorough survey paper, using the terms "spatial locality" and "temporal locality"; this paper has served as a reference for many computer designers. While most studies have relied on simulations, Clark [1983] used a hardware monitor to record cache misses of the VAX-11/780 over several days. Section 8.9 reports these findings, along with the work Clark did with Emer on TLBs [1984, 1985]. A similar study was performed on the VAX 8800 [Clark et al. 1988]. Agarwal, Sites, and Horowitz [1986] changed the microcode of a VAX to make traces of system and user code. These traces are used in this book (and are available through the publisher). Hill [1987] proposed the three Cs used in Section 8.4 to explain cache misses. Caches remain an active area of research, as Smith [1986] has recorded in his extensive bibliography.

Many of the ideas in the advanced cache section have only been tried recently. The inclusion of caches on microprocessors such as the Motorola 68020 gave rise to two-level cache machines; the Sun 3/260 in 1986 was perhaps the first. In 1988, the Silicon Graphics 4D/240 had two levels of caches for data and instructions, with the second level added primarily for cache coherency to allow four-way multiprocessing. The MIPS RC 6280 is probably the first machine to go to two-level caches for the reasons given on page 465 [Roberts, Taylor, and Layman 1990]. Goodman and Chiang [1984] were the first to publish an investigation of static-column DRAM in a memory hierarchy, while Kelly [1988] refined the idea by using virtual addresses. Goodman [1987] showed that aliases can be handled at cache-miss time, and Wang, Baer, and Levy [1989] show that the extra control for this does not look too bad for two levels of cache.

In comparison to the other ideas in the advanced section, cache-coherency research is much older. Tang [1976] published the first cache-coherency protocol using directories, and this approach was implemented in the IBM 3081. Censier and Feautrier [1978] describe a technique with status tags in memory. The first machine to use snooping caches was the Synapse N+1 [Frank 1984]; the first publication on snooping caches was by Goodman [1983]. Archibald and Baer [1986] survey the wide variety of schemes for cache coherency. References on the protocols mentioned in their paper and in Figure 8.45 are Frank [1984] for Synapse; Goodman [1983] for Write Once; Katz et al. [1985] for Berkeley; McCreight [1984] for Dragon; Papamarcos and Patel [1984] for Illinois; and Thacker and Stewart [1987] for Firefly. Baer and Wang [1988] discuss multilevel inclusion. Eggers's [1989] nomenclature for categorizing snooping caches is adopted in this text. Chapter 10, Section 10.7 mentions the use of prefetching to improve cache performance, and Kroft [1981] describes the design of a cache that allows the cache to service subsequent requests while the requested data is prefetched. Przybylski [1990] and the dissertations by Agarwal [1987], Eggers [1989], and Hill [1987] investigate many aspects of the advanced cache topics in more depth.

Papers on another use of locality, register windows or stack caches, are by Patterson and Sequin [1981], Ditzel and McClellan [1982], and Lampson [1982]. Sites wrote an earlier paper [1979] suggesting one way to use the expanding resources of VLSI was to get higher performance by using a lot of registers, and these schemes are one interpretation of that recommendation.

References

AGARWAL, A. [1987]. *Analysis of Cache Performance for Operating Systems and Multiprogramming*, Ph.D. Thesis, Stanford Univ., Tech. Rep. No. CSL-TR-87-332 (May).

AGARWAL, A., R. L. SITES, AND M. HOROWITZ [1986]. "ATUM: A new technique for capturing address traces using microcode," *Proc. 13th Annual Symposium on Computer Architecture* (June 2–5), Tokyo, Japan, 119–127.

ARCHIBALD, J. AND J.-L. BAER [1986]. "Cache coherence protocols: Evaluation using a multiprocessor simulation model," *ACM Trans. on Computer Systems* 4:4 (November) 273–298.

BAER, J.-L. AND W.-H. WANG [1988]. "On the inclusion property for multi-level cache hierarchies," *Proc. 15th Annual Symposium on Computer Architecture* (May–June), Honolulu, 73–80.

BELL , C. G. AND W. D. STRECKER [1976]. "Computer structures: What have we learned from the PDP-11?," *Proc. Third Annual Symposium on Computer Architecture* (January), Pittsburgh, Penn., 1–14.

BLAKKEN, J. [1983]. "Register windows for SOAR," in *Smalltalk On A RISC: Architectural Investigations,* Proc. of CS 292R (April) 126–140, University of California.

CASE, R.P. AND A. PADEGS [1978]. "The architecture of the IBM System/370," *Communications of the ACM* 21:1, 73–96. Also appears in D. P. Siewiorek, C. G. Bell, and A. Newell, *Computer Structures: Principles and Examples* (1982), McGraw-Hill, New York, 830–855.

CENSIER, L. M. AND P. FEAUTRIER [1978]. "A new solution to the coherence problem in multicache systems," *IEEE Trans. on Computers* C-27:12 (December) 1112–1118.

CLARK, D. W. [1983]. "Cache performance of the VAX-11/780," *ACM Trans. on Computer Systems* 1:1, 24–37.

CLARK, D. W. AND J. S. EMER [1985]. "Performance of the VAX-11/780 translation buffer: Simulation and measurement," *ACM Trans. on Computer Systems* 3:1, 31–62.

CLARK, D. W, P. J. BANNON, AND J. B. KELLER [1988]. "Measuring VAX 8800 Performance with a Histogram hardware monitor," *Proc. 15th Annual Symposium on Computer Architecture* (May–June), Honolulu, Hawaii, 176–185.

CONTI, C., D. H. GIBSON, AND S. H. PITOWSKY [1968]. "Structural aspects of the System/360 Model 85, part I: General organization," *IBM Systems J.* 7:1, 2–14.

CRAWFORD, J. H AND P. P. GELSINGER [1987]. *Programming the 80386*, Sybex, Alameda, Calif.

DITZEL, D. R., AND H.R. MCCLELLAN [1982]. "Register allocation for free: The C machine stack cache" *Symposium on Architectural Support for Programming Languages and Operating Systems* (March 1–3), Palo Alto, Calif., 48–56.

EGGERS, S. [1989]. *Simulation Analysis of Data Sharing in Shared Memory Multiprocessors* , Ph. D. Thesis, Univ. of California, Berkeley, Computer Science Division Tech. Rep. UCB/CSD 89/501 (April).

EMER, J. S. AND D. W. CLARK [1984]. "A characterization of processor performance of the VAX-11/780," *Proc. 11th Annual Symposium on Computer Architecture* (June), Ann Arbor, Mich., 301–310.

FABRY, R. S. [1974]. "Capability based addressing," *Comm. ACM* 17:7 (July) 403–412.

FRANK, S. J. [1984]. "Tightly coupled multiprocessor systems speed memory access times," *Electronics* 57:1 (January) 164–169.

GIBSON, D. H. [1967]. "Considerations in block–oriented systems design," *AFIPS Conf. Proc.* 30, SJCC, 75–80.

GOODMAN, J. R. [1983]. "Using cache memory to reduce processor memory traffic," *Proc. Tenth Annual Symposium on Computer Architecture* (June 5–7), Stockholm, Sweden, 124–131.

GOODMAN, J. R. and M.-C. Chiang [1984]. "The use of static column RAM as a memory hierarchy," *Proc. 11th Annual Symposium on Computer Architecture* (June 5–7), Ann Arbor, Mich., 167–174.

GOODMAN, J. R. [1987]. "Coherency for multiprocessor virtual address caches," *Proc. Second Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, Calif., 71–81.

HALBERT, D. C. AND P. B. KESSLER [1980]. "Windows of overlapping register frames," *CS 292R Final Reports* (June) 82–100.

HILL, M. D. [1987]. *Aspects of Cache Memory and Instruction Buffer Performance*, Ph. D. Thesis, Univ. of California at Berkeley Computer Science Division, Tech. Rep. UCB/CSD 87/381 (November).

HILL, M. D. [1988]. "A case for direct mapped caches," *Computer* 21:12 (December) 25–40.

HUGUET, M. AND T. LANG [1985]. "A reduced register file for RISC architectures," *Computer Architecture News* 13:4 (September) 22–31.

KATZ, R., S. EGGERS, D. A. WOOD, C. PERKINS, AND R. G. SHELDON [1985]. "Implementing a cache consistency protocol," *Proc. 12th Annual Symposium on Computer Architecture*, 276–283.

KELLY, E. [1988]. "'SCRAM Cache' in Sun-4/110 beats traditional caches," *Sun Technology* 1:3 (Summer) 19–21.

KILBURN, T., D. B. G. EDWARDS, M. J. LANIGAN, F. H. SUMNER [1962]. "One-level storage system," *IRE Transactions on Electronic Computers* EC-11 (April) 223–235. Also appears in D. P. Siewiorek, C. G. Bell, and A. Newell, *Computer Structures: Principles and Examples* (1982), McGraw-Hill, New York, 135–148.

KROFT, D. [1981]. "Lockup-free instruction fetch/prefetch cache organization," *Proc. Eighth Annual Symposium on Computer Architecture* (May 12–14), Minneapolis, Minn., 81–87.

LAMPSON, B. W. [1982]. "Fast procedure calls," *Symposium on Architectural Support for Programming Languages and Operating Systems* (March 1–3), Palo Alto, Calif., 66–75.

LIPTAY, J. S. [1968]. "Structural aspects of the System/360 Model 85, part II: The cache," *IBM Systems J.* 7:1, 15–21.

MCCALL, K. [1983]. "The Smalltalk-80 benchmarks," *Smalltalk 80: Bits of History, Words of Advice,* G. Krasner, ed., Addison-Wesley, Reading, Mass., 153–174.

MCCREIGHT, E. [1984]. "The Dragon computer system: An early overview," Tech. Rep. Xerox Corp. (September).

MCFARLING, S. [1989]. "Program optimization for instruction caches," *Proc. Third International Conf. on Architectural Support for Programming Languages and Operating Systems* (April 3–6), Boston, Mass., 183–191.

PAPAMARCOS, M. AND J. PATEL [1984]. "A low coherence solution for multiprocessors with private cache memories," *Proc. of the 11th Annual Symposium on Computer Architecture* (June), Ann Arbor, Mich., 348–354.

PRZYBYLSKI, S. A. [1990]. *Cache Design: A Performance-Directed Approach,* Morgan Kaufmann Publishers, San Mateo, Calif.

PRZYBYLSKI, S. A., M. HOROWITZ, AND J. L. HENNESSY [1988]. "Performance tradeoffs in cache design," *Proc. 15th Annual Symposium on Computer Architecture* (May–June), Honolulu, Hawaii, 290–298.

ROBERTS, D., G. TAYLOR, AND T. LAYMAN [1990]. "An ECL RISC microprocessor designed for two-level cache," *IEEE Compcon* (February).

SAMPLES, A. D. AND P. N. HILFINGER [1988]. "Code reorganization for instruction caches," Tech. Rep. UCB/CSD 88/447 (October), Univ. of Calif., Berkeley.

SITES, R. L., [1979]. "How to use 1000 registers," *Caltech Conf. on VLSI* (January).

SMITH, A. J. [1982]. "Cache memories," *Computing Surveys* 14:3 (September) 473–530.

SMITH, A. J. [1986]. "Bibliography and readings on CPU cache memories and related topics," *Computer Architecture News* (January) 22–42.

SMITH, J. E. AND J. R. GOODMAN [1983]. "A study of instruction cache organizations and replacement policies," *Proc. Tenth Annual Symposium on Computer Architecture* (June 5–7), Stockholm, Sweden,, 132–137.

STRECKER, W. D. [1976]. "Cache memories for the PDP-11?," *Proc. Third Annual Symposium on Computer Architecture* (January), Pittsburgh, Penn., 155–158.

TANG, C. K. [1976]. "Cache system design in the tightly coupled multiprocessor system," *Proc. 1976 AFIPS National Computer Conf.*, 749–753.

TAYLOR, G. S., P. N. HILFINGER, J. R. LARUS, D. A. PATTERSON, AND B. G. ZORN [1986]. "Evaluation of the SPUR Lisp architecture," *Proc. 13th Annual Symposium on Computer Architecture* (June 2–5), Tokyo, Japan, 444–452.

THACKER, C. P. AND L. C. STEWART [1987]. "Firefly: a multiprocessor workstation," *Proc. Second Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, Calif., 164–172.

UNGAR, D. M. [1987]. *The Design of a High Performance Smalltalk System*, The MIT Press Distinguished Dissertation Series, Cambridge, Mass.

WANG, W.-H., J.-L. BAER, AND H. M. LEVY [1989]. "Organization and performance of a two-level virtual-real cache hierarchy," *Proc. 16th Annual Symposium on Computer Architecture* (May 28–June 1), Jerusalem, Israel , 140–148.

WILKES, M. [1965]. "Slave memories and dynamic storage allocation," *IEEE Trans. Electronic Computers* EC-14:2 (April) 270–271.

WILKES, M. V. [1982]. "Hardware support for memory protection: Capability implementations," *Proc. Symposium on Architectural Support for Programming Languages and Operating Systems* (March 1–3), Palo Alto, Calif., 107–116.

WULF, W. A., R. LEVIN AND S. P. HARBISON [1981]. *Hydra/C.mmp: An Experimental Computer System,* McGraw-Hill, New York.

EXERCISES

8.1 [15/15/12/12] <2.2,8.4> Let's try to show how you can make *unfair* benchmarks. Here are two machines with the same processor and main memory but different cache organizations. Assume the miss time is 10 times a cache-hit time for both machines. Assume writing a 32-bit word takes 5 times as long as a cache hit (for the write-through cache), and that writing a whole 16-byte block takes 10 times as long as a cache-read hit. (for the write-back cache). The caches are unified; that is, they contain both instructions and data.

Cache A: 64 sets, 2 elements per set, each block is 16 bytes, and it uses write through.

Cache B: 128 sets, 1 element per set, each block is 16 bytes, and it uses write back.

a. [15] Describe a program that makes machine A run as much faster as possible than machine B. (Be sure to state any further assumptions you need, if any.)

b. [15] Describe a program that makes machine B run as much faster as possible than machine A. (Be sure to state any further assumptions you need, if any.)

c. [12] Approximately how much faster is the program in Part a on machine A than machine B?

d. [12] Approximately how much faster is the program in Part b on machine B than machine A?

8.2 [20] <2.2,6.4,8.4> To simplify pipelined execution, some machines insert NOP instructions rather than interlock the pipeline (see pages 273–275 in Chapter 6). Ignoring cache misses, assume that the Spice code takes 2,000,000 clocks in either case (since the version without NOPS still interlocks, which takes an extra clock each time.) Figure 8.53

shows data collected for a portion of Spice execution with a 64-KB, direct-mapped, instruction-only cache with one-word blocks.

	With NOPS	Without NOPS	Ratio with/without
Total references	1,500,000	1,180,000	1.27
Cache misses	34,153	24,908	1.37
Miss rate	2.28	2.10	1.09

FIGURE 8.53 Spice miss rates with and without NOPs.

The conclusion of a study based on Figure 8.53 was that a 9% increase in the miss rate of the program with NOPS will have a small but measurable impact on performance. What is the actual impact on performance assuming a 10-clock miss penalty?

8.3 [15/15] <8.4> You purchased an Acme computer with the following features:

1. 90% of all memory accesses are found in the cache;
2. Each cache block is two words, and the whole block is read on any miss;
3. The processor sends references to its cache at the rate of 10^7 words per second;
4. 25% of the references of (3) are writes;
5. Assume that the bus can support 10^7 words per second, reads or writes;
6. The bus reads or writes a single word at a time (the bus cannot read or write two words at once);
7. Assume at any one time, 30% of the blocks in the cache have been modified;
8. The cache uses write allocate on a write miss. -write back

You are considering adding a peripheral to the bus, and you want to know how much of the bus bandwidth is already used. Calculate the percentage of bus bandwidth used on the average in the two cases below. The percentage is called the *traffic ratio* in the literature. Be sure to state your assumptions.

a. [15] The cache is write through.

b. [15] The cache is write back.

8.4 [20] <8.4> One drawback to the write-back scheme is that writes will probably take two cycles. During the first cycle, we detect whether a hit will occur, and during the second (assuming a hit) we actually write the data. Let's assume that 50% of the blocks are dirty for a write-back cache. Using statistics for loads and stores from DLX in Figure C.4 in Appendix C, estimate the performance of a write-through cache with a one-cycle write versus a write-back cache with a two-cycle write for each of the programs. For this question, assume that the write buffer for write through will never stall the CPU (no penalty). Assume a cache hit takes 1 clock cycle, the cache miss penalty is 10 clock cycles, and a block write from the cache to main memory takes 10 clock cycles. Finally, assume the instruction-cache miss rate is 2% and the data-cache miss rate is 4%.

8.5 [15/20/10] <8.4> To save development time, the Sun 3/280 and the Sun 4/280 used identical memory systems, even though the CPUs were quite different. Assume the same case exists for a new machine, one board using a VAX CPU and the other a DLX CPU. For now assume the miss-rate information in Figure 8.12 and 8.16 (pages 421 and 424) apply to both architectures. Use the average column in Figure C.4 in Appendix C as needed for DLX instruction mix, and the caption of Figure 8.16 (page 424) for VAX instruction/data mix. Assume the following:

Miss penalty is 12 clock cycles.

A perfect write buffer that never stalls the CPU.

The base CPI assuming a perfect memory system is 6.0 for the VAX and 1.5 for DLX.

A unified cache adds 1 extra clock cycle to each load and store of DLX (since there is a single memory port) but not for the VAX.

You are considering three options:

1. A 4-way–set-associative unified cache of 64 KB.

2. Two 2-way–set-associative caches of 32 KB each, one for instructions and one for data.

3. A direct-mapped unified cache of 128 KB. Assume that clock rate is 10% faster in this case since the mapping is direct and the CPU address does not need to drive two caches, nor does the data bus need to be multiplexed. This faster clock rate increases the miss penalty to 13 clock cycles.

a. [15] What is the average memory-access time in clock cycles for each organization?

b. [20] What is the CPI for each machine and cache organization?

c. [10] What cache organization gives the best average performance for the two CPUs?

8.6 [25/15] <2.3,8.4,8.8> Some microprocessors have custom single-chip caches as companions to the CPU. For example, the Motorola 88100 CPU can have up to 8 of the 88200 cache chips. These chips tend to be more expensive than off-the-shelf static RAM chips. The MIPS R3000 includes a comparator on the CPU chip so that cache tags and data can be built from off-the-shelf static RAMs.

a. [25] Using the program that analyzes cache miss rates how many 16K-by-4 cache RAMs must the R3000 use to get the same performance as two 88200 chips? Both designs use separate instruction and data caches. The MIPS design assumes a block size of 16 bytes with subblock placement for each word. The cache is write through with a 4-word write buffer. The Motorola 88200 is 4-way set associative with 16 KB per chip and a 16-byte block using LRU replacement.
b. [15] Here is the data on the price of each chip (quantity 1 as of 8/1/89):

Motorola 88100: $697

Motorola 88200: $875

MIPS R3000 (25 MHz): $300

MIPS R3010 FPU (25 MHz): $350

16K by 4 SRAM (for 25 MHz R3000): $21

Which system will be cheaper and by how much?

8.7 [15/25/15/15] <2.3,8.4> The Intel i860 has its caches on chip and its die size is 1.2 cm*1.2 cm. It has a 2-way–set-associative, 4-KB instruction cache and a 2-way–set-associative, 8-KB data cache using write through or write back. Both caches use 32-byte blocks. There are no write buffers or process identifiers to reduce cache flushing. The i860 also includes a 64-entry, 4-way–set-associative TLB to map its 4-KB pages. Address translation occurs before the caches are accessed. The Cypress 7C601 CPU chip size is 0.8 cm by 0.7 cm and has no on-board cache—a cache controller chip (7C604) and two 16K * 16 cache chips (7C157) are offered to form a 64-KB unified cache. The controller includes a TLB with 64 entries managed fully associatively with 4096 process identifiers to reduce flushing. It supports 32-byte blocks with direct-mapped placement, and either write through or write back. There is a one-block write buffer for write back and a four-word write buffer for write through. The chip sizes are 1.0 cm by 0.9 cm for the 7C604 and 0.8 cm by 0.7 cm. for the 7C157.

a. [15] Using the cost model of Chapter 2, what is the cost of the Cypress chip set versus the Intel chip? (Use Figure 2.11 on page 62 to determine chip costs by finding the closest die size in that table to the Intel and Cypress die area.)

b. [25] Use the DLX cache traces and cache simulator to determine the average memory-access time for each cache organization. Assume a miss takes 6 clocks latency plus 1 clock for each 32-bit word. Assume both systems run at the same clock rate and use write allocate.

c. [15] What is the comparative cost/performance of these chips using average memory-access time as the measure?

d. [15] What is the percent increase in cost of a color workstation that uses the more expensive chips?

8.8 [25/10/15] <8.4> The CRAY X-MP instruction buffers can be thought of as an instruction-only cache. The total size is 1 KB, broken into 4 blocks of 256 bytes per block. The cache is fully associative and uses a first-in/first-out replacement policy. The access time on a miss is 10 clock cycles, with the transfer time of 64 bytes every clock cycle. The X-MP takes 1 clock cycle on a hit. Use the cache simulator and the DLX traces to determine:

a. [25] Instruction miss rate

b. [10] Average instruction memory-access time measured in clock cycles

c. [15] What does the CPI of the CRAY X-MP have to be for the portion due to instruction cache misses to be 10% or less?

8.9 [25] <8.4> Traces from a single process give too-high estimates for caches used in a multiprocess environment. Write a program that merges the uniprocess DLX traces into a single reference stream. Use the process-switch statistics in Figure 8.25 (page 439) as the average process-switch rate with an exponential distribution about that mean. (Use number of clock cycles rather than instructions, and assume the CPI of DLX is 1.5.) Use the cache simulator on the original traces and the merged trace. What is the miss rate for each assuming a 64-KB direct-mapped cache with 16-byte blocks? (There is a process-identified tag in the cache tag so that the cache doesn't have to be flushed on each switch.)

8.10 [25] <8.4> One approach to reducing misses is to prefetch the next block. A simple but effective strategy is when block *i* is referenced to make sure block *i*+1 is in the cache, and if not, to prefetch it. Do you think prefetching is more or less effective with increasing block size? Why? Is it more or less effective with increasing cache size? Why? Use statistics from the cache simulator and the traces to support your conclusion.

8.11 [20/25] <8.4> Smith and Goodman [1983] found that for a **small-instruction–only** cache, a cache using direct mapping could consistently outperform one using fully associative with LRU replacement.

a. [20] Explain why this would be possible. (Hint: you can't explain this with the 3C model because it ignores replacement policy.)

b. [25] Use the cache simulator to see if their results hold for the traces.

8.12 [Discussion] <8.4> If you look at conflict misses for a given associativity in Figure 8.12, as capacity increases the conflict misses go up and down. For example, for 2-way–set-associative mapping the miss rate for 2-KB cache is .010, a 4-KB cache is .013, and an 8-KB cache is .008. Why in the world would this happen?

8.13 [30] <8.5> Use the cache simulator and traces to calculate the effectiveness of a 4-bank versus 8-bank interleaved memory. Assume each word transfer takes one clock on the bus and a random access is 8 clocks. Measure the bank conflicts and memory bandwidth for these cases:

a. No cache and no write buffer.

b. A 64-KB, direct-mapped, write-though cache with four-word blocks.

c. A 64-KB, direct-mapped, write-back cache with four-word blocks.

d. A 64-KB, direct-mapped, write-though cache with four-word blocks but the "interleaving" comes from a page-mode DRAM.

e. A 64-KB, direct-mapped, write-back cache with four-word blocks but the "interleaving" comes from a page mode DRAM.

8.14 [20] <8.6> If the base CPI with a perfect memory system is 1.5, what is the CPI for these cache organizations? Use Figure 8.12 (page 421):

a. Direct-mapped, 16-KB unified cache using write back.

b. Two-way–set-associative, 16-KB unified cache using write back.

c. Direct-mapped, 32-KB unified cache using write back.

Assume the memory latency is 6 clocks, the transfer rate is 4 bytes per clock cycle and that 50% of the transfers are dirty. There are 16 bytes per block and 20% of the instructions are data-transfer instructions. The caches fetch words of the block in address order and the CPUs stall until all words of the block arrive. There is no write buffer. Add to the assumptions above a TLB that takes 20 clock cycles on a TLB miss. A TLB does not slow down a cache hit. For the TLB, make the simplifying assumption that 1% of all references aren't found in TLB, either when addresses come directly from the CPU or when addresses come from cache misses. What is the impact on performance of the TLB if the cache above is physical or virtual?

8.15 [30] <3.8,8.9> The example in Section 8.9 (page 478) refines the instructions fetched into the CPU from the cache due to the instruction-prefetch buffer. How does this increase of 13% to 39% in instruction words fetched affect the difference in the instruction words fetched from DLX versus VAX? The extra instruction fetches of the VAX hurt only when they bring something into the cache that is not used before it is displaced, while DLX would seem to need a larger cache for its larger program. Write a simulator emulating the instruction-prefetch buffer to measure the increase in cache misses using the VAX address traces and see if prefetching is a significant increase in cache misses.

8.16 [25–40] <8.7> Study the impact of adding register windows to DLX. This study can range from simply estimating the register-traffic savings to modifying the DLX compiler and simulator to measure costs and benefits directly.

8.17 [10] <8.8> Data General described the design of a three-level cache for an ECL implementation of the 88000 architecture. What is the formula for average access time for a three-level cache?

8.18 [20] <8.8> What is the performance loss for a four-way multiprocessor with I/O devices? Suppose 1% of all data references to the cache cause invalidation to the other data caches and that all CPUs stall four clocks on an invalidation. Assume a 64-KB, direct-mapped cache for data and a 64-KB, direct-mapped cache for instructions with a block size of 32 bytes yields a 1% miss rate for instructions and a 2% miss rate for data, with 20% of all CPU memory references being for data. The CPI of the CPU is 1.5 with a perfect memory system and it takes 10 clocks on a cache miss whether the data is dirty or clean.

8.19 [25] <8.8> Use the traces to calculate the effectiveness of early restart and out-of-order fetch. What is the distribution of first accesses to a block as block size increases from 2 words to 64 words by factors of two for:

a. A 64-KB, instruction-only cache?

b. A 64-KB, data-only cache?

c. A 128-KB unified cache?

Assume direct-mapped placement.

8.20 [30] <8.8> Use the cache simulator and traces with a program you write yourself to compare the effectiveness schemes for fast writes:

a. 1-word buffer and the CPU stalls on a data-read cache miss with a write-through cache.

b. 4-word buffer and the CPU stalls on a data-read cache miss with a write-through cache.

c. 4-word buffer and the CPU stalls on a data-read cache miss only if there is a potential conflict in the addresses with a write-through cache.

d. A write-back cache that writes dirty data first and then loads the missed block.

e. A write-back cache with a one-block write buffer that loads the miss data first and then stalls the CPU on a clean miss if the write buffer is not empty.

f. A write-back cache with a one-block write buffer that loads the miss data first and then stalls the CPU on a clean miss only if the write buffer is not empty and there is a potential conflict in the addresses.

Assume a 64-KB, direct-mapped cache for data and a 64-KB, direct-mapped cache for instructions with a block size of 32 bytes. The CPI of the CPU is 1.5 with a perfect memory system and it takes 14 clocks on a cache miss and 7 clocks to write a single word to memory.

8.21 [30] <8.8> Use the cache simulator and traces with a program you write yourself to create a two-level cache simulator. Use this program to see at what cache size is the global miss rate of a second-level cache approximately the same as a single-level cache of the same capacity.

8.22 [Discussion] <8.6> Some people have argued that with increasing capacity of memory storage per chip, virtual memory is an idea whose time has passed, and they expect to see it dropped from future computers. Find reasons for and against this argument.

8.23 [Discussion] <8.6> So far, few computer systems take advantage of the extra security available with gates and rings found in a machine like the Intel 80286. Construct some scenario whereby the computer industry would switch over to this model of protection.

8.24 [Discussion] <8.4> Recent research has tried to use compilers to improve cache performance (see McFarling [1989] and Samples and Hilfinger [1988]):

a. Which of the 3C's are compilers trying to improve and which are they not? Why?

b. Which mapping is best for compiler improvement? Why?

8.25 [Discussion] <8.3> Many times a new technology has been invented that is expected to make a major change to the memory hierarchy. For the sake of this question, let's suppose that biological computer technology becomes a reality. Suppose biological memory technology has an unusual characteristic: It is as fast as the fastest semiconductor DRAMs, and it can be randomly accessed; but it only costs as much as magnetic-disk memory. It has the further advantage of not being any slower no matter how big it is. The only drawback is that you can only Write it Once, but you can Read it Many times. Thus it is called a "WORM" memory. Because of the way it is manufactured, the WORM- memory module can be easily replaced. See if you can come up with several new ideas to take advantage of WORMs to build better computers using "bio-technology."

I/O certainly has been lagging in the last decade.

Seymour Cray, Public Lecture (1976)

Also, I/O needs a lot of work.

David Kuck, Keynote Address,
15th Annual Symposium on Computer Architecture (1988)

9 Input/Output

9.1 Introduction

Input/output has been the orphan of computer architecture. Historically neglected by CPU enthusiasts, the prejudice against I/O is institutionalized in the most widely used performance measure, CPU time (page 35). Whether a computer has the best or the worst I/O system in the world cannot be measured by CPU time, which by definition ignores I/O. The second class citizenship of I/O is even apparent in the label "peripheral" applied to I/O devices.

This attitude is contradicted by common sense. A computer without I/O devices is like a car without wheels—you can't get very far without them. And while CPU time is interesting, response time—the time between when the user types a command and when she gets results—is surely a better measure of performance. The customer who pays for a computer cares about response time, even if the CPU designer doesn't. Finally, as rapid improvements in CPU performance compress traditional classes of computers together, it is I/O that serves to distinguish them:

- The difference between a mainframe computer and a minicomputer is that a mainframe can support many more terminals and disks.
- The difference between a minicomputer and a workstation is that a workstation has a screen, a keyboard, and a mouse.

- The difference between a file server and a workstation is that a file server has disks and tape units but no screen, keyboard, or mouse.
- The difference between a workstation and a personal computer is that workstations are always connected together on a network.

It may come to pass that computers from high-end workstations to low-end supercomputers will use the same "super-microprocessors." Differences in cost and performance would be determined only by the memory and I/O systems (and the number of processors).

I/O's revenge is at hand. Suppose we have a difference between CPU time and response time of 10%, and we speed up the CPU by a factor of 10, while neglecting I/O. Amdahl's Law tells us that we will get a speedup of only 5 times, with half the potential of the CPU wasted. Similarly, making the CPU 100 times faster without improving the I/O would obtain a speedup of only 10 times, squandering 90% of the potential. If, as predicted in Chapter 1, performance of CPUs improves at 50% to 100% per year, and I/O does not improve, every task will become I/O bound. There would be no reason to buy faster CPUs—and no jobs for CPU designers.

While this single chapter cannot fully vindicate I/O, it may at least atone for some of the sins of the past and restore some balance.

Are CPUs Ever Idle?

Some suggest that the prejudice is well founded. I/O speed doesn't matter, they argue, since there is always another process to run while one process waits for a peripheral.

There are several points to make in reply. First, this is an argument that performance is measured as throughput—more tasks per hour—rather than as response time. Plainly, if users didn't care about response time, interactive software never would have been invented, and there would be no workstations today. (The next section gives experimental evidence on the importance of response time.) It may also be expensive to rely on processes while waiting for I/O, since main memory must be larger or else the paging traffic from process switching would actually increase I/O. Furthermore, with desktop computing there is only one person per CPU, and thus fewer processes than in timesharing; many times the only waiting process is the human being! And some applications, such as transaction processing (Section 9.3), place strict limits on response time as part of the performance analysis.

But let's accept the argument at face value and explore it further. Suppose the difference between response time and CPU time today is 10%, and a CPU that is ten times faster can be achieved without changing I/O performance. A process will then spend 50% of its time waiting for I/O, and two processes will have to be perfectly aligned to avoid CPU stalls while waiting for I/O. Any further CPU improvement will only increase CPU idle time.

Thus, I/O throughput can limit system throughput, just as I/O response time limits system response time. Let's see how to predict performance for the whole system.

9.2 Predicting System Performance

System performance is limited by the slowest part of the path between CPU and I/O devices. The performance of a system can be limited by the speed of any of these pieces of the path, shown in Figure 9.1:

- The CPU
- The cache memory
- The main memory
- The memory–I/O bus
- The I/O controller or I/O channel
- The I/O device
- The speed of the I/O software
- The efficiency of the software's use of the I/O device

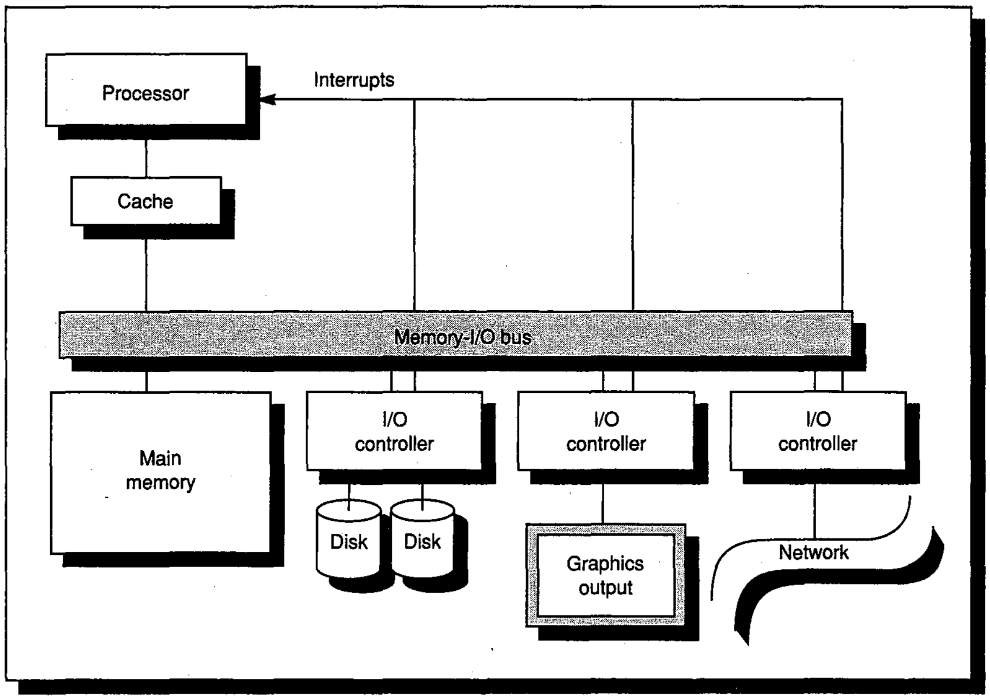

FIGURE 9.1 Typical collection of I/O devices on a computer.

If the system is not balanced, the high performance of some components may be lost due to the low performance of one link in the chain. The art of I/O design is to configure a system such that the speeds of all components are matched.

In earlier chapters we have assumed that the fastest CPU is the single object of our desire, but CPU performance is not the same as system performance. For example, suppose we have two workloads, A and B. Both workloads take 10 seconds to run. Workload A does so little I/O that it is not worth mentioning. Workload B keeps I/O devices busy four seconds, and this time is completely overlapped with CPU activities. Suppose the CPU is replaced by a newer model with five times the performance. Intuitively, we realize that workload A takes two seconds—fully five times faster—but workload B is I/O bound and cannot take less than four seconds. Figure 9.2 illustrates our intuition.

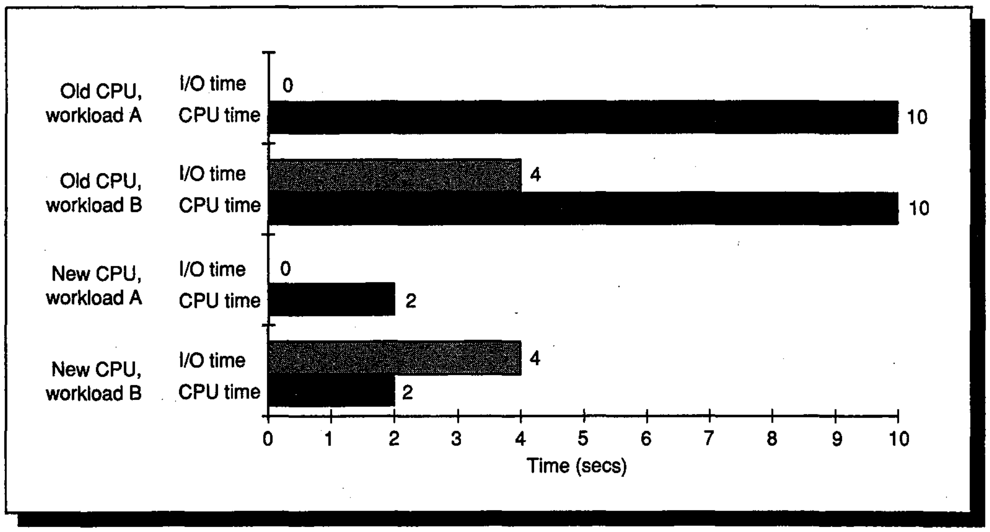

FIGURE 9.2 The overlapped execution of the two workloads with the original CPU and then a CPU with five times the performance. We can see that the elapsed time for workload A is indeed 1/5 of the time with the new CPU, but it is limited to four seconds in workload B because I/O speed is not improved.

Determining the performance of such cases requires a new formula. The elapsed execution time of a workload can be broken into three pieces

$$\text{Time}_{\text{workload}} = \text{Time}_{\text{CPU}} + \text{Time}_{\text{I/O}} - \text{Time}_{\text{overlap}}$$

where Time_{CPU} means the time the CPU is busy, $\text{Time}_{\text{I/O}}$ means the time the I/O system is busy, and $\text{Time}_{\text{overlap}}$ means the time both the CPU and the I/O system are busy. Using workload B with the old CPU in Figure 9.2 as an example, the times in seconds are:

10 for $\text{Time}_{\text{workload}}$,

10 for Time_{CPU},

4 for $\text{Time}_{\text{I/O}}$, and

4 for $\text{Time}_{\text{overlap}}$.

Assuming we speed up only the CPU, one way to calculate the time to execute the workload is:

$$\text{Time}_{\text{workload}} = \frac{\text{Time}_{\text{CPU}}}{\text{Speedup}_{\text{CPU}}} + \text{Time}_{\text{I/O}} - \frac{\text{Time}_{\text{overlap}}}{\text{Speedup}_{\text{CPU}}}$$

Since the CPU time is shrunk, it stands to reason that the overlap time is also shrunk. The system speedup when we want to improve I/O is equivalent:

$$\text{Time}_{\text{workload}} = \text{Time}_{\text{CPU}} + \frac{\text{Time}_{\text{I/O}}}{\text{Speedup}_{\text{I/O}}} - \frac{\text{Time}_{\text{overlap}}}{\text{Speedup}_{\text{I/O}}}$$

Let's try an example before explaining a limitation of these formulas.

Example One workload takes 50 seconds to run, with the CPU being busy 30 seconds and the I/O being busy 30 seconds. How much time will the workload take if we replace the CPU with one that has four times the performance?

Answer The total elapsed time is 50 seconds, yet the sum of CPU time and I/O time is 60 seconds. Thus the overlap time must be 10 seconds. Plugging into the formula:

$$\text{Time}_{\text{workload}} = \frac{\text{Time}_{\text{CPU}}}{\text{Speedup}_{\text{CPU}}} + \text{Time}_{\text{I/O}} - \frac{\text{Time}_{\text{overlap}}}{\text{Speedup}_{\text{CPU}}} = \frac{30}{4} + 30 - \frac{10}{4} = 35$$

This example uncovers a complication with this formula: How much of the time that the workload is busy on the faster CPU is overlapped with I/O? Figure 9.3 (page 504) shows three options. Depending on the resulting overlap after speedup, the time for the workload varies from 30 to 37.5 seconds.

In reality we can't know which is correct without measuring the workload on the faster CPU to see what overlap occurs. The formulas above assume option (c) in Figure 9.3; the overlap scales by the same speedup as the CPU, so we will call it $\text{Time}_{\text{scaled}}$ (rather than $\text{Time}_{\text{workload}}$). Maximum overlap assumes that as much of the overlap as possible is maintained, but that the new overlap cannot be larger than the original overlap or the CPU time after speedup. Minimum overlap assumes that as much of the overlap as possible is eliminated, but that the overlap time will not shrink by more than the time removed from the CPU or I/O time. If we introduce the abbreviations $\text{New}_{\text{CPU}} = \text{Time}_{\text{CPU}} / \text{Speedup}_{\text{CPU}}$ and $\text{New}_{\text{I/O}} = \text{Time}_{\text{I/O}} / \text{Speedup}_{\text{I/O}}$, the time of the workload for maximum overlap ($\text{Time}_{\text{best}}$) and minimum overlap ($\text{Time}_{\text{worst}}$) can be written as:

$$\text{Time}_{\text{best}} = \text{New}_{\text{CPU}} + \text{Time}_{\text{I/O}} - \text{Minimum}\ (\text{Time}_{\text{overlap}}, \text{New}_{\text{CPU}})$$

$$\text{Time}_{\text{worst}} = \text{New}_{\text{CPU}} + \text{Time}_{\text{I/O}} - \text{Maximum}\ (0, \text{Time}_{\text{overlap}} - (\text{Time}_{\text{CPU}} - \text{New}_{\text{CPU}}))$$

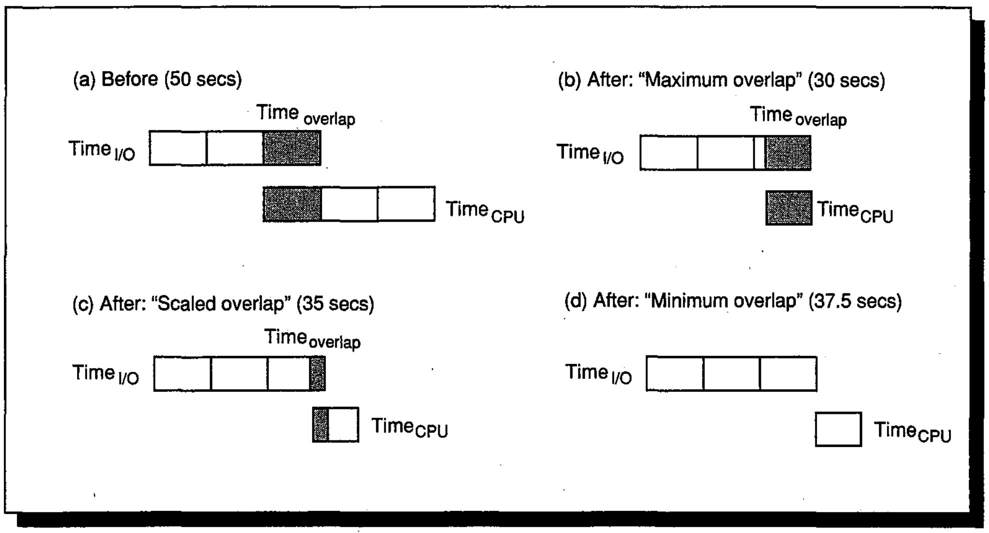

FIGURE 9.3 The original overlap in the example above (a) and three interpretations of overlap after speedup. Each block represents 10 seconds, except that the block for the new CPU time is 7.5 seconds. The overlapped portions of Time_{CPU} and $\text{Time}_{\text{I/O}}$ are shaded. (b) shows the new Time_{CPU} overlapping completely with I/O, giving a time of the workload of 30 seconds. (c) shows the overlap of the Time_{CPU} is scaled with $\text{Speedup}_{\text{CPU}}$, giving a total of 35 seconds, with 2.5 seconds of overlapped execution. (d) shows no overlap with I/O, so the total is 37.5 seconds.

Example

Calculate the three time predictions for workload B in Figure 9.2

Answer

$$\text{Time}_{\text{best}} = \frac{10}{5} + 4 - \text{Minimum}\left(\frac{10}{5}, 4\right) = 2 + 4 - 2 = 4$$

$$\text{Time}_{\text{scaled}} = \frac{10}{5} + 4 - \frac{4}{5} = 2 + 4 - 0.8 = 5.2$$

$$\text{Time}_{\text{worst}} = \frac{10}{5} + 4 - \text{Maximum}\ (0, 4-(10-\frac{10}{5})) = 2 + 4 - 0 = 6$$

Sometimes changes will be made to both the CPU and the I/O system. The formulas become:

$$\text{Time}_{\text{scaled}} = \text{New}_{\text{CPU}} + \text{New}_{\text{I/O}} - \frac{\text{Time}_{\text{overlap}}}{\text{Maximum}(\text{Speedup}_{\text{CPU}}, \text{Speedup}_{\text{I/O}})}$$

$$\text{Time}_{\text{best}} = \text{New}_{\text{CPU}} + \text{New}_{\text{I/O}} - \text{Minimum}(\text{Time}_{\text{overlap}}, \text{New}_{\text{CPU}}, \text{New}_{\text{I/O}})$$

$$\text{Time}_{\text{worst}} = \text{New}_{\text{CPU}} + \text{New}_{\text{I/O}} - \text{Max}\ (0, \text{Time}_{\text{overlap}} - \text{Max}\ (\text{Time}_{\text{CPU}} - \text{New}_{\text{CPU}}, \text{Time}_{\text{I/O}} - \text{New}_{\text{I/O}}))$$

The formula for scaled overlap says that the overlap period is reduced by the larger of the two speedups. The formula for maximum overlap ($\text{Time}_{\text{best}}$) says that as much overlap as possible is retained, but the new overlap cannot be larger than the original overlap or the CPU or I/O time after speedup. Finally, the formula for minimum overlap ($\text{Time}_{\text{worst}}$) says that the overlap is reduced by the larger of the time removed from the CPU time and the time removed from the I/O time (but that the overlap time cannot be less than 0). Figure 9.4 shows the three examples of speedup where both the I/O and CPU are improved.

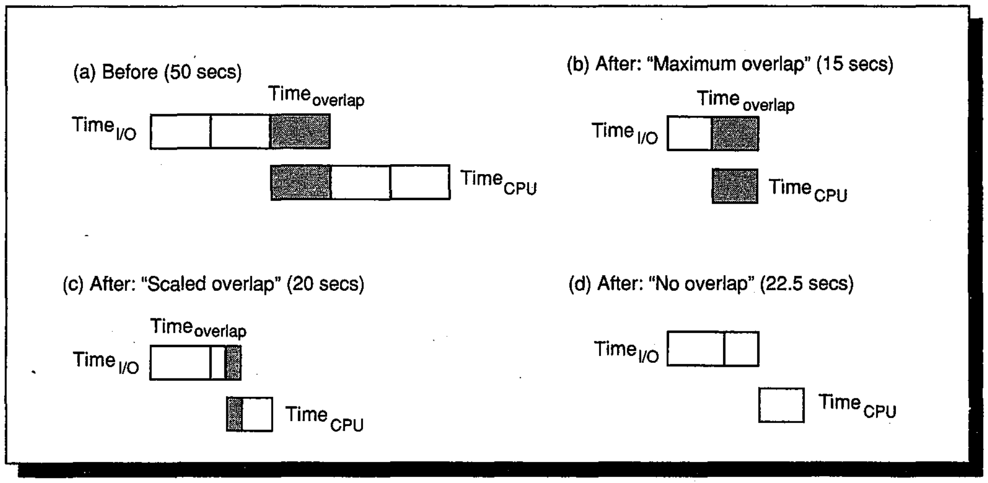

FIGURE 9.4 Time for workload in Figure 9.3(a) with $\text{Speedup}_{\text{CPU}}$ = 4 and $\text{Speedup}_{\text{I/O}}$ = 2.

Let's look at a detailed example showing speedup of both the CPU and I/O.

Example

Suppose a workload on the current systems takes 64 seconds. The CPU is busy the whole time, and the channels connecting the I/O devices to the CPU are busy 36 seconds. The computer manager is considering two upgrade options: either a single CPU that has twice the performance, or two CPUs that have twice the throughput and twice as many channels. The time of the actual I/O devices is so small it can be ignored. For the dual CPU option assume that the workload can be evenly spread between the CPUs and channels. What is the performance improvement for each option?

Answer

Since there is no change to the I/O system with the single faster CPU, time for the workload assuming scaled overlap is then simply

$$\text{Time}_{\text{scaled}} = \frac{\text{Time}_{\text{CPU}}}{\text{Speedup}_{\text{CPU}}} + \text{Time}_{\text{I/O}} - \frac{\text{Time}_{\text{overlap}}}{\text{Speedup}_{\text{CPU}}}$$

$$= \frac{64}{2} + 36 - \frac{36}{2} = 32 + 36 - 18 = 50$$

For the dual CPU with more channels,

$$\text{Time}_{\text{scaled}} =$$

$$\frac{\text{Time}_{\text{CPU}}}{\text{Speedup}_{\text{CPU}}} + \frac{\text{Time}_{\text{I/O}}}{\text{Speedup}_{\text{I/O}}} - \frac{\text{Time}_{\text{overlap}}}{\text{Maximum}(\text{Speedup}_{\text{CPU}}, \text{Speedup}_{\text{I/O}})}$$

$$= \frac{64}{2} + \frac{36}{2} - \frac{36}{\text{Maximum}(2\,,2)} = 32 + 18 - 18 = 32$$

Assuming scaled overlap, the dual CPU is more than 50% faster. Using best-case scaling, the dual CPU is 13% faster, while worst-case scaling suggests it is 39% faster.

As these examples demonstrate, we need improvement in I/O performance to match the improvement in CPU performance if we are to achieve faster computer systems. We can now examine metrics of I/O devices to understand how to improve their performance and thus the whole system.

9.3 I/O Performance Measures

I/O performance has measures that have no counterparts in CPU design. One of these is diversity: Which I/O devices can connect to the computer system? Another is capacity: How many I/O devices can connect to a computer system?

In addition to these unique measures, the traditional measures of performance, response time and throughput also apply to I/O. (I/O throughput is sometimes called "I/O bandwidth" and response time is sometimes called "latency.") The next two figures offer insight into how response time and throughput trade off against each other. Figure 9.5 shows the simple producer-server model. The producer creates tasks to be performed and places them in the queue; the server takes tasks from the queue and performs them.

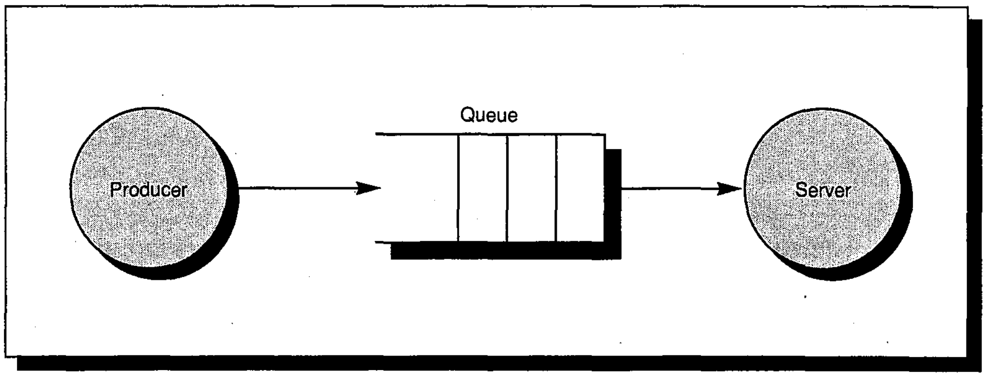

FIGURE 9.5 The traditional producer-server model of response time and throughput. Response time begins when a task is placed in the queue and ends when it is completed by the server. Throughput is the number of tasks completed by the server in unit time.

Response time is defined as the time a task takes from the moment it is placed in the queue until the server finishes the task. Throughput is simply the average number of tasks completed by the server over a time period. To get the highest possible throughput, the server should never be idle, and thus the queue should never be empty. Response time, on the other hand, counts time spent in the queue and is therefore minimized by the queue being empty.

Another measure of I/O performance is the interference of I/O with CPU execution. Transferring data may interfere with the execution of another process. There is also overhead due to handling I/O interrupts. Our concern here is how many more clock cycles a process will take because of I/O for another process.

Throughput Versus Response Time

Figure 9.6 shows throughput versus response time (or latency), for a typical I/O system. The knee of the curve is the area where a little more throughput results in much longer response time or, conversely, a little shorter response time results in much lower throughput.

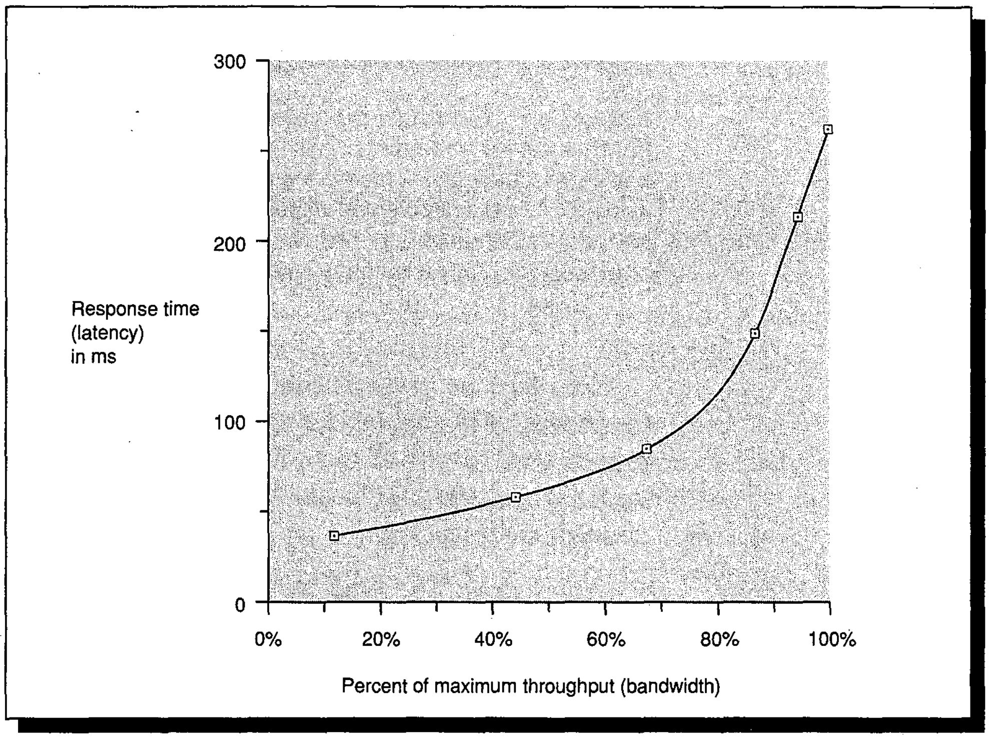

FIGURE 9.6 Throughput versus response time. Latency is normally reported as response time. Note that absolute minimum response time achieves only 11% of the throughput while the response time for 100% throughput takes seven times the minimum response time. Chen [1989] collected these data for an array of magnetic disks.

Life would be simpler if improving performance always meant improvements in both response time and throughput. Adding more servers, as in Figure 9.7, increases throughput: By spreading data across two disks instead of one, tasks may be serviced in parallel. Alas, this does not help response time, unless the workload is held constant and the time in the queues is reduced because of more resources.

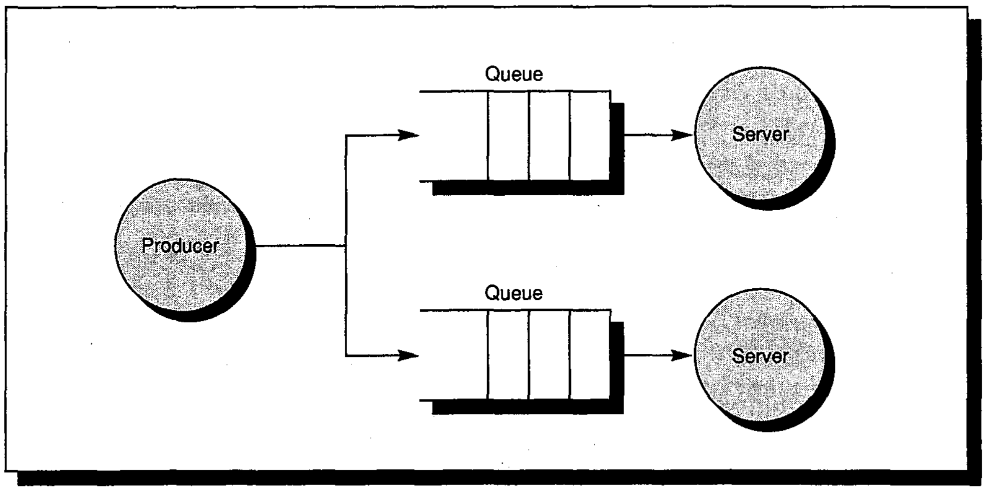

FIGURE 9.7 The single-producer, single-server model of Figure 9.5 is extended with another server and queue. This increases I/O system throughput and takes less time to service producer tasks. Increasing the number of servers is a common technique in I/O systems. There is a potential imbalance problem with two queues; unless data is placed perfectly in the queues, sometimes one server will be idle with an empty queue while the other server is busy with many tasks in its queue.

How does the architect balance these conflicting demands? If the computer is interacting with human beings, Figure 9.8 suggests an answer. This figure presents the results of two studies of interactive environments, one keyboard oriented and one graphical. An interaction or *transaction* with a computer is divided into three parts:

1. *Entry time*: The time for the user to enter the command. In the graphics system in Figure 9.8 it took 0.25 seconds on average to enter the command versus 4.0 seconds for the conventional system.
2. *System response time*: The time between when the user enters the command and the complete response is displayed.
3. *Think time*: The time from the reception of the response until the user begins to enter the next command.

The sum of these three parts is called the *transaction time*. Several studies report that user productivity is inversely proportional to transaction time; transactions per hour measures the work completed per hour by the user.

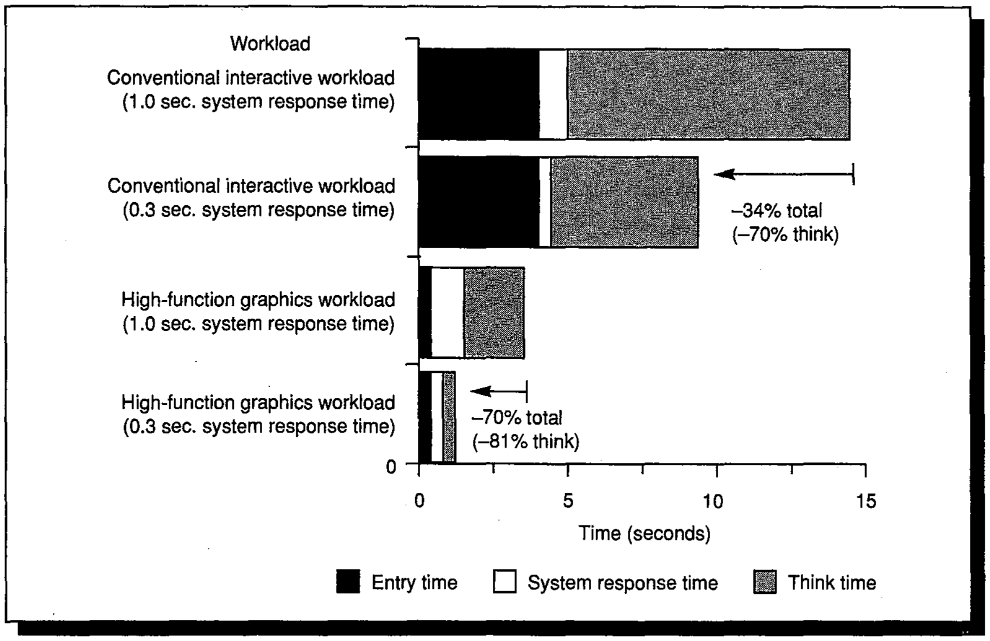

FIGURE 9.8 A user transaction with an interactive computer divided into entry time, system response time, and user think time for a conventional system and graphics system. The entry times are the same independent of system response time. The entry time was 4 seconds for the conventional system and 0.25 seconds for the graphics system. (From Brady [1986].)

The results in Figure 9.8 show that reduction in response time actually decreases transaction time by more than just the response time reduction: Cutting system response time by 0.7 seconds saves 4.9 seconds (34%) from the conventional transaction and 2.0 seconds (70%) from the graphics transaction. This implausible result is explained by human nature; people need less time to think when given a faster response.

Whether these results are explained as a better match to the human attention span or getting people "on a roll," several studies report this behavior. In fact, as computer responses drop below a second, productivity seems to make a more than linear jump. Figure 9.9 (page 510) compares transactions per hour (the inverse of transaction time) of a novice, an average engineer, and an expert performing physical design tasks at graphics displays. System response time magnified talent: a novice with subsecond response time was as productive as an experienced professional with slower response, and the experienced engineer in turn could outperform the expert with a similar advantage in response time. In all cases the number of transactions per hour jumps more than linearly with subsecond response time.

Since humans may be able to get much more work done per day with better response time, it is possible to attach an economic benefit to the customer of lowering response time into the subsecond range [IBM 1982], thereby helping the architect decide how to tip the balance between response time and throughput.

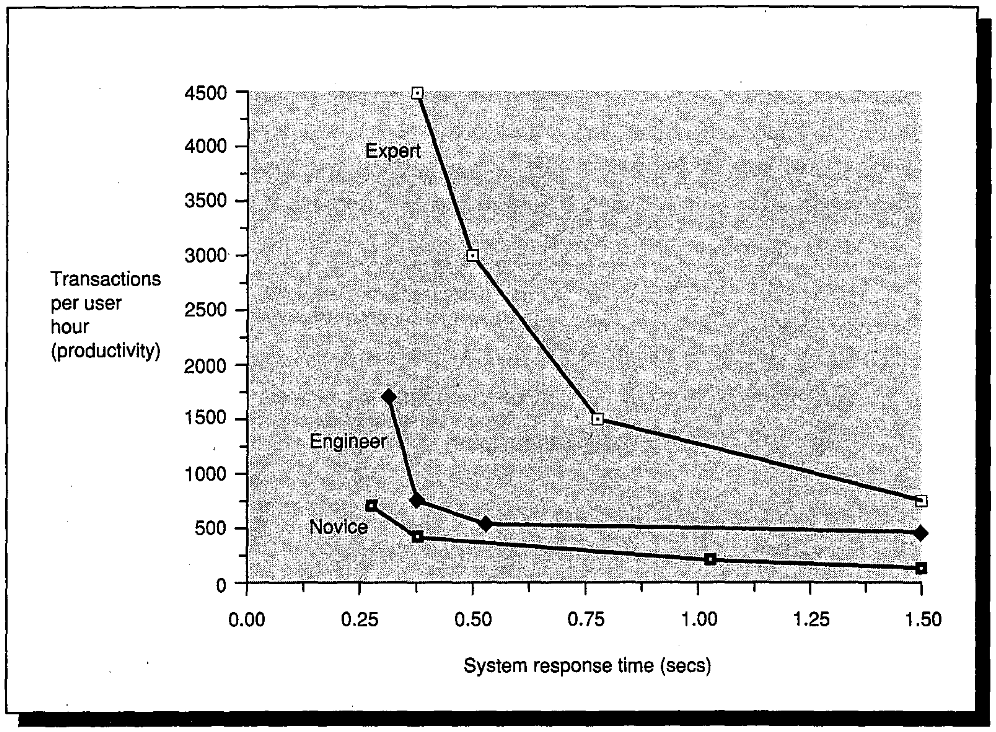

FIGURE 9.9 Transactions per hour versus computer response time for a novice, experienced engineer, and expert doing physical design on a graphics system. Transactions per hour is a measure of productivity. (From IBM [1982].)

Examples of Measurements of I/O Performance—Magnetic Disks

Benchmarks are needed to evaluate I/O performance, just as they are needed to evaluate CPU performance. We begin with benchmarks for magnetic disks. Three traditional applications of disks are with large-scale scientific problems, transaction processing, and file systems.

Supercomputer I/O Benchmarks

Supercomputer I/O is dominated by accesses to large files on magnetic disks. For example, Bucher and Hayes [1980] benchmarked supercomputer I/O using 8-MB sequential file transfers. Many supercomputer installations run batch jobs, each of which may last for hours. In these situations, I/O consists of one large read followed by writes to snapshot the state of the computation should the computer crash. As a result, supercomputer I/O in many cases consists of more output than input. Some models of Cray Research computers have such limited main memory that programmers must break their programs into overlays and swap them to disk (see Section 8.5 of Chapter 8), which also causes large sequential transfers. Thus, the overriding supercomputer I/O measure is data throughput: number of bytes per second that can be transferred between supercomputer main memory and disks during large transfers.

Transaction Processing I/O Benchmarks

In contrast, *transaction processing* (TP) is chiefly concerned with *I/O rate*: the number of disk accesses per second, as opposed to *data rate*, measured as bytes of data per second. TP generally involves changes to a large body of shared information from many terminals, with the TP system guaranteeing proper behavior on a failure. If, for example, a bank's computer fails when a customer withdraws money, the TP system would guarantee that the account is debited if the customer received the money and that the account is unchanged if the money was not received. Airline reservations systems as well as banks are traditional customers for TP.

Two dozen members of the TP community conspired to form a benchmark for the industry and, to avoid the wrath of their legal departments, published the report anonymously [1985]. This benchmark, called DebitCredit, simulates bank tellers and has as its bottom line the number of debit/credit transactions per second (TPS); in 1990, the TPS for high-end machines is about 300. The DebitCredit performs the operation of a customer depositing or withdrawing money. The performance measurement is the peak TPS, with 95% of the transactions having less than a one-second response time. The DebitCredit computes the cost per TPS, based on the five-year cost of the computer-system hardware and software. Disk I/O for DebitCredit is random reads and writes of 100-byte records along with occasional sequential writes.

Depending on how cleverly the transaction-processing system is designed, each transaction results in between two and ten disk I/Os and takes between 5,000 and 20,000 CPU instructions per disk I/O. The variation largely depends on the efficiency of the transaction processing software, although in part it depends on the extent to which disk accesses can be avoided by keeping information in main memory. The benchmark requires that for TPS to increase, the number of tellers and the size of the account file must also increase. Figure 9.10 shows this unusual relationship in which more TPS requires more users.

TPS	Number of ATMs	Account-file size
10	1,000	0.1 GB
100	10,000	1.0 GB
1,000	100,000	10.0 GB
10,000	1,000,000	100.0 GB

FIGURE 9.10 Relationship among TPS, tellers, and account-file size. The DebitCredit benchmark requires that the computer system handle more tellers and larger account files before it can claim a higher transaction-per-second milestone. The benchmark is supposed to include "terminal handling" overhead, but this metric is sometimes ignored.

This is to ensure that the benchmark really measures disk I/O; otherwise a large main memory dedicated to a database cache with a small number of accounts would unfairly yield a very high TPS. (Another perspective is the number of accounts must grow since a person is not likely to use the bank more frequently just because the bank has a faster computer!)

File System I/O Benchmarks

File systems, for which disks are mainly used in timesharing systems, have a different access pattern. Ousterhout et al. [1985] measured a UNIX file system and found that 80% of accesses to files of less than 10 KB and 90% of **all** file accesses were sequential. The distribution by type of file access was 67% reads, 27% writes, and 6% read-write accesses. In 1988, Howard et al. [1988] proposed a file-system benchmark that is becoming popular. Their paper describes five phases of the benchmark, using 70 files with a total size of 200 KB:

MakeDir—Constructs a target subtree that is identical in structure to the source subtree.

Copy—Copies every file from the source subtree to the target subtree.

ScanDir—Recursively traverses the target subtree and examines the status of every file in it. It does not actually read the contents of any file.

ReadAll—Scans every byte of every file in the target subtree once.

Make—Compiles and links all the files in the target subtree. [p. 55]

The file-system measurements of Howard et al. [1988], like those of Ousterhout et al. [1985], found the ratio of disk reads to writes to be about 2:1. This benchmark reflects that measure.

9.4 Types of I/O Devices

Now that we have covered measurements of I/O performance, let's describe the devices themselves. While the computing model has changed little since 1950, I/O devices have become rich and diverse. Three characteristics are useful in organizing this disparate conglomeration:

- *Behavior*—input (read once), output (write only, cannot be read), or storage (can be reread and usually rewritten)
- *Partner*—either a human or a machine is at the other end of the I/O device, either feeding data on input or reading data on output
- *Data rate*—the peak rate at which data can be transferred between the I/O device and the main memory or CPU

Using these characteristics, a keyboard is an input device used by a human with a peak data rate of about 10 bytes per second. Figure 9.11 shows some of the I/O devices connected to computers.

The advantage of designing I/O devices for humans is that the performance target is fixed. Figure 9.12 shows the I/O performance of people.

Device	Behavior	Partner	Data rate (KB/sec)
Keyboard	Input	Human	0.01
Mouse	Input	Human	0.02
Voice input	Input	Human	0.02
Scanner	Input	Human	200.00
Voice output	Output	Human	0.60
Line printer	Output	Human	1.00
Laser printer	Output	Human	100.00
Graphics display	Output	Human	30,000.00
(CPU to frame buffer)	Output	Human	200.00
Network-terminal	Input or output	Machine	0.05
Network-LAN	Input or output	Machine	200.00
Optical disk	Storage	Machine	500.00
Magnetic tape	Storage	Machine	2,000.00
Magnetic disk	Storage	Machine	2,000.00

FIGURE 9.11 Examples of I/O devices categorized by behavior, partner, and data rate. This is the raw data rate of the device rather than the rate an application would see. Storage devices can be further distinguished by whether they support sequential access (e.g., tapes) or random access (e.g., disks). Note that networks can act either as input or output devices but, unlike storage, cannot reread the same information.

Human organ	I/O rate (KB/sec)	I/O latency (ms)
Ear	8.000–60.000	10
Eye—reading text	0.030–0.375	10
Eye—pattern recognition	125.000	10
Hand—typing	0.010–0.020	100
Voice	0.003–0.015	100

FIGURE 9.12 Peak I/O rates for people. Input via seeing patterns is our highest I/O rate; hence the popularity of graphic output devices. Maberly [1966] says the average reading speed is 28 bytes per second and the maximum is 375 bytes per second. The telephone company sets a 170-ms limit to the time between when an operator pushes a button to accept a call until a voice path must be established. The phone company transmits voice at 8 KB per second. (None of these parameters are expected to change, unless anabolic steroids become a breakfast supplement!)

To put the data rates of each device into perspective, Figure 9.13 shows the relative peak memory bandwidth needed to support each device, assuming a computer had exactly one of each device transferring at its peak rate.

Rather than discuss the characteristics of all I/O devices, we will concentrate on the three devices with the highest data rates: magnetic disks, graphics displays, and local area networks. These are also the devices that have the highest leverage on user productivity. In this chapter we are not talking about floppy disks, but the original "hard" disks. These magnetic disks are what IBM calls *DASD*s, for *Direct-Access Storage Devices.*

Magnetic Disks

I think Silicon Valley was misnamed. If you look back at the dollars shipped in products in the last decade there has been more revenue from magnetic disks than from silicon. They ought to rename the place Iron Oxide Valley.

Al Hoagland, one of the pioneers of magnetic disks (1982)

In spite of repeated attacks by new technologies, magnetic disks have dominated secondary storage since 1965. Magnetic disks play two roles in computer systems:

- Long-term, nonvolatile storage for files, even when no programs are running
- A level of the memory hierarchy below main memory used for virtual memory during program execution (see Section 8.5 in Chapter 8)

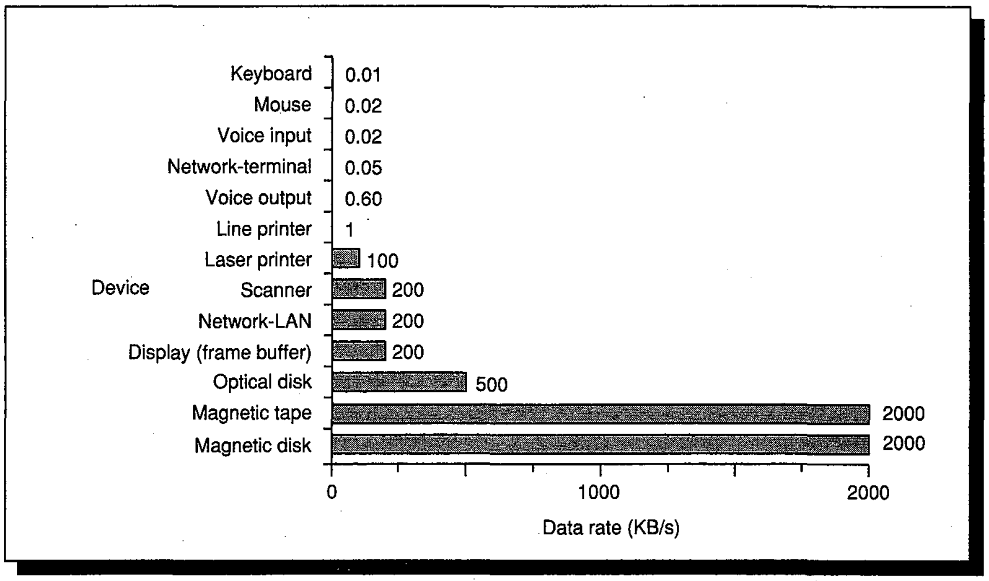

FIGURE 9.13 I/O devices sorted from lowest data rate to highest. The data rate for the graphics display is from the CPU to the frame buffer because the CPU isn't involved in the transfer from the frame buffer to the display (see Graphics Displays subsection below).

As descriptions of magnetic disks can be found in countless books, we will only list the key characteristics with the terms illustrated in Figure 9.14. A magnetic disk consists of a collection of platters (1 to 20), rotating on a spindle at about 3600 revolutions per minute (RPM). These platters are metal disks covered with magnetic recording material on both sides. Disk diameters vary by a factor of five, from 14 to 2.5 inches. Traditionally, the widest disks have the highest performance, and the smallest disks have the lowest cost per disk drive.

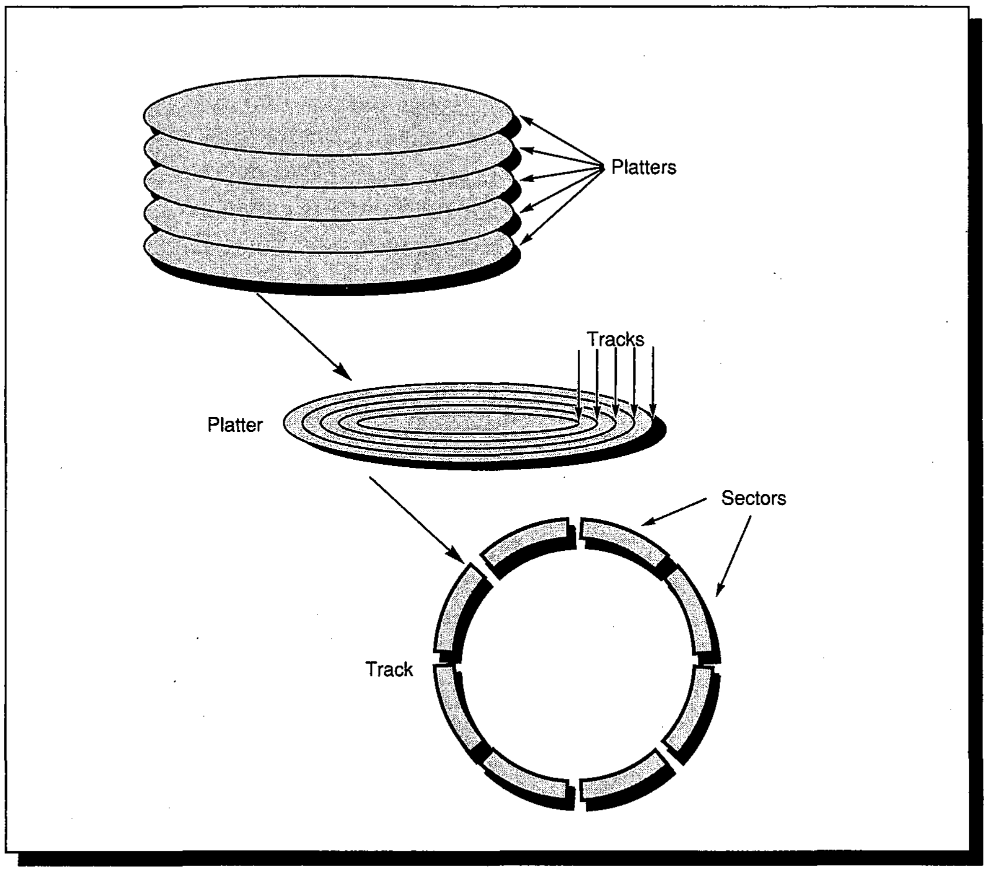

FIGURE 9.14 Disks are organized into platters, tracks, and sectors. Both sides of a platter are coated so that information can be stored on both surfaces.

Each disk surface is divided into concentric circles, designated *tracks*. There are typically 500 to 2000 tracks per surface. Each track in turn is divided into *sectors* that contain the information; each track might have 32 sectors. The sector is the smallest unit that can be read or written. The sequence recorded on the magnetic media is a sector number, a gap, the information for that sector including error correction code, a gap, the sector number of the next sector, and so on. Traditionally all tracks have the same number of sectors; the outer tracks, which are longer, record information at a lower density than the inner tracks. Recording more sectors on the outer tracks than on the inner tracks, called *constant bit density*, is becoming more widespread with the advent of intelligent interface standards such as SCSI (see Section 9.5). IBM mainframe disks allow users to select the size of the sectors, while almost all other systems fix the size of the sector.

To read and write information into a sector, a movable *arm* containing a *read/write head* is located over each surface. Bits are recorded using a run-length limited code, which improves the recording density of the magnetic media. The arms for each surface are connected together and move in conjunction, so that every arm is over the same track of every surface. The term *cylinder* is used to refer to all the tracks under the arms at a given point on all surfaces.

To read or write a sector, the disk controller sends a command to move the arm over the proper track. This operation is called a *seek*, and the time to move the arm to the desired track is called *seek time*. Average seek time is the subject of considerable misunderstanding. Disk manufacturers report minimum seek time, maximum seek time, and average seek time in the manuals. The first two are easy to measure, but average was open to wide interpretation. The industry decided to calculate average seek time as the sum of the time for all possible seeks divided by the number of possible seeks. Average seek times are advertised to be 12 ms to 20 ms, but depending on the application and operating system the actual average seek time may be only 25% to 33% of the advertised number, due to locality of disk references. Section 9.10 has a detailed example.

The time for the requested sector to rotate under the head is the *rotation latency* or *rotational delay*. Most disks rotate at 3600 RPM, and an average latency to the desired information is halfway around the disk; the average rotation time for most disks is therefore

$$\text{Average rotation time} = \frac{0.5}{3600 \text{ RPM}} = 0.0083 \text{ sec} = 8.3 \text{ ms}$$

The next component of a disk access, *transfer time*, is the time to transfer a block of bits, typically a sector, under the read-write head. This is a function of the block size, rotation speed, recording density of a track, and speed of the electronics connecting disk to computer. Transfer rates in 1990 are typically 1 to 4 MB per second.

In addition to the disk drive, there is usually also a device called a *disk controller*. Between the disk controller and main memory is a hierarchy of controllers and data paths, whose complexity varies with the cost of the computer (see Section 9.9). Since the transfer time is often a small portion of a full disk access, the controller in higher performance systems disconnects the data paths from the disks while they are seeking so that other disks can transfer their data to memory.

Thus, the final component of disk-access time is *controller time*, which is the overhead the controller imposes in performing an I/O access. When referring to performance of a disk in a computer system, the time spent waiting for a disk to become free (*queueing delay*) is added to this time.

Example

What is the average time to read or write a 512-byte sector for a typical disk today? The advertised average seek time is 20 ms, the transfer rate is 1MB/sec, and the controller overhead is 2 ms. Assume the disk is idle so that there is no queuing delay.

Answer

Average disk access is equal to average seek time + average rotational delay + transfer time + controller overhead. Using the calculated, average seek time, the answer is

$$20 \text{ ms} + 8.3 \text{ ms} + \frac{0.5 \text{ KB}}{1.0 \text{ MB/sec}} + 2 \text{ ms} = 20 + 8.3 + 0.5 + 2 = 30.8 \text{ ms}$$

Assuming the measured, average seek time is 25% of the calculated number, the answer is

$$5 \text{ ms} + 8.3 \text{ ms} + 0.5 \text{ ms} + 2 \text{ ms} = 15.8 \text{ ms}$$

Figure 9.15 shows characteristics of magnetic disks for four manufacturers. Large-diameter drives have many more megabytes to amortize the cost of electronics, so the traditional wisdom was that they had the lowest cost per megabyte. But this advantage is offset for the small drives by the much higher sales volume, which lowers manufacturing costs: 1990 OEM prices are $2 to $3

Characteristics	IBM 3380	Fujitsu M2361A	Imprimis Wren IV	Conner CP3100
Disk diameter (inches)	14	10.5	5.25	3.5
Formatted data capacity (MB)	7500	600	344	100
MTTF (hours)	52,000	20,000	40,000	30,000
Number of arms/box	4	1	1	1
Maximum I/Os/second/arm	50	40	35	30
Typical I/Os/second/arm	30	24	28	20
Maximum I/Os/second/box	200	40	35	30
Typical I/Os/second/box	120	24	28	20
Transfer rate (MB/sec)	3	2.5	1.5	1
Power/box (W)	1,650	640	35	10
MB/W	1.1	0.9	9.8	10.0
Volume (cu. ft.)	24	3.4	0.1	.03
MB/cu. ft.	310	180	3440	3330

FIGURE 9.15 Characteristics of magnetic disks from four manufacturers. Comparison of IBM 3380 disk model AK4 for mainframe computers, Fujitsu M2361A "Super Eagle" disk for minicomputers, Imprimis Wren IV disk for workstations, and Conner Peripherals CP3100 disk for personal computers. Maximum I/Os/second signifies maximum number of average seeks and average rotates for a single sector access. (Table from Katz, Patterson, and Gibson [1990].)

per megabyte, almost independent of width. The small drives also have advantages in power and volume. The price of a megabyte of disk storage in 1990 is 10 to 30 times cheaper than the price of a megabyte of DRAM in a system.

The Future of Magnetic Disks

The disk industry has concentrated on improving the capacity of disks. Improvement in capacity is customarily expressed as *areal density*, measured in bits per square inch:

$$\text{Areal density} = \frac{\text{Tracks}}{\text{Inch}} \text{ on a disk surface} * \frac{\text{Bits}}{\text{Inch}} \text{ on a track}$$

Areal density can be predicted according to the *maximum areal density* (MAD) formula:

$$\text{MAD} = 10^{(\text{year}-1971)/10} \text{ million bits per square inch}$$

Thus, storage density improves by a factor of 10 every decade, doubling density every three years.

Cost per megabyte has dropped consistently at 20% to 25% per year, with smaller drives playing the larger role in this improvement. Because it is easier to

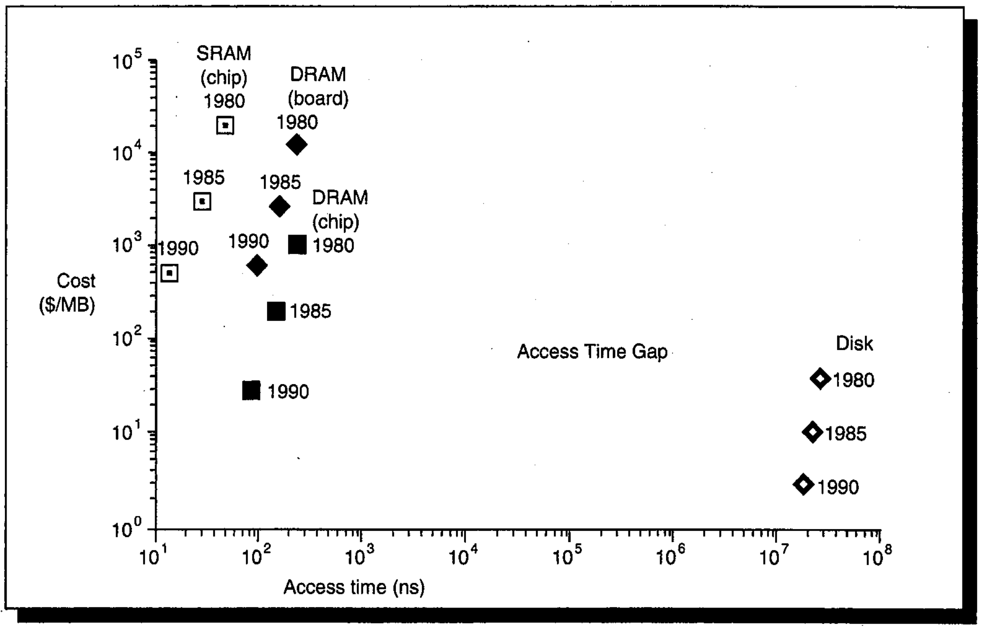

FIGURE 9.16 Cost versus access time for SRAM, DRAM, and magnetic disk in 1980, 1985, and 1990. (Note the difference in cost between a DRAM chip and DRAM chips packaged on a board and ready to plug into a computer.) The two-order-of-magnitude gap in cost and five-order-of-magnitude gap in access times between semiconductor memory and rotating magnetic disk has inspired a host of competing technologies to try to fill it. So far, such attempts have been made obsolete before production by improvements in magnetic disks, DRAMs, or both.

spin the smaller mass, smaller diameter disks save power as well as volume. Smaller drives also have fewer cylinders so the seek distances are shorter. In 1990, 5.25-inch or 3.5-inch drives are probably the leading technology, while the future may see even smaller drives. We can expect significant savings in volume and power, but little in speed. Increasing density (bits per inch on a track) has improved transfer times, and there has been some small improvement in seek speed. Rotation speeds have been steady at 3600 RPM for a decade, but some manufacturers plan to go to 5400 RPM in the early 1990s.

As mentioned earlier, magnetic disks have been challenged many times for supremacy of secondary storage. One reason has been the fabled *Access Time Gap* as shown in Figure 9.16. Many a scientist has tried to invent a technology to fill that gap. Let's look at some of the recent attempts.

Using DRAMs as Disks

A current challenger to disks for dominance of secondary storage is *solid state disks* (SSDs), built from DRAMs with a battery to make the system nonvolatile; and *expanded storage* (ES), a large memory that allows only block transfers to or from main memory. ES acts like a software-controlled cache (the CPU stalls during the block transfer) while SSD involves the operating system just like a transfer from magnetic disks. The advantages of SSD and ES are trivial seek times, higher potential transfer rate, and possibly higher reliability. Unlike just a larger main memory, SSDs and ESs are autonomous: They require special commands to access their storage, and thus are "safe" from some software errors that write over main memory. The block-access nature of SSD and ES allows error correction to be spread over more words, which means lower cost or greater error recovery. For example, IBM's ES uses the greater error recovery to allow it to be constructed from less reliable (and less expensive) DRAMs without sacrificing product availability. SSDs, unlike main memory and ES, may be shared by multiple CPUs because they function as separate units. Placing DRAMs in an I/O device rather than memory is also one way to get around the address-space limits of the current 32-bit computers. The disadvantage of SSD and ES is cost, which is at least ten times per megabyte the cost of magnetic disks.

Optical Disks

Another challenger to magnetic disks is *optical compact disks* or CDs. The *CD/ROM* is removable and inexpensive to manufacture, but it is a read-only media. The newer CD/writable is also removable, but has a high cost per megabyte and low performance. A common misperception about *write-once optical disks* is that once they are written, the information cannot be destroyed; in fact, write once means one reliable write and then a "fuzzy" bitwise ORing of the previous and new data.

So far, magnetic disk challengers have never had a product to market at the right time. By the time a new product ships, disks have made advances as predicted by MAD formula, and costs have dropped accordingly. Optical disks, however, may have the potential to compete with new tape technologies for archival storage.

Disk Arrays

One other future candidate for optimizing storage is not a new technology, but a new organization of disk storage—arrays of small and inexpensive disks. The argument for arrays is that since price per megabyte is independent of disk size, potential throughput can be increased by having many disk drives and, hence, many disk arms. Simply spreading data over multiple disks automatically forces accesses to several disks. (While arrays improve throughput, latency is not necessarily improved.) The drawback to arrays is that with more devices, reliability drops: N devices generally have $1/N$ the reliability of a single device.

Reliability and Availability

This brings us to two terms that are often confused—reliability and availability. The term reliability is commonly used incorrectly to mean availability; if something breaks, but the user can still use the system, it seems as if the system still "works," and hence it seems more reliable. Here is the proper distinction:

Reliability—is anything broken?

Availability—is the system still available to the user?

Adding hardware can therefore improve availability (for example, ECC on memory), but it cannot improve reliability (the DRAM is still broken). Reliability can only be improved by bettering environmental conditions, by building from more reliable components, or by building with fewer components. Another term, *data integrity*, refers to always reporting when information is lost when a failure occurs; this is very important to some applications.

So, while a disk array can never be more reliable than a smaller number of larger disks when each disk has the same failure rate, availability can be improved by adding redundant disks. That is, if a single disk fails, the lost information can be reconstructed from redundant information. The only danger is in getting another disk failure between the time a disk fails and the time it is replaced (termed *mean time to repair* or MTTR). Since the *mean time to failure* (MTTF) of disks is three to five years, and the MTTR is measured in hours, redundancy can make the availability of 100 disks much higher than that of a single disk.

Since disk failures are self-identifying, information can be reconstructed from just parity: The good disks plus the parity disk can be used to calculate the information that is on the failed disk. Hence, the cost of higher availability is

1/*N*, where *N* is the number of disks protected by parity. Just as direct-mapped associative placement in caches can be considered a special case of set-associative placement (see Section 8.4), the *mirroring* or *shadowing* of disks can be considered the special case of one data disk and one parity disk (*N*=1). Parity can be accomplished by duplicating the data, so mirrored disks have the advantage of simplifying parity calculation. Duplicating data also means that the controller can improve read performance by reading from the disk of the pair that has the shortest seek distance, although this optimization is at the cost of write performance because the arms of the pair of disks are no longer always over the same track. Of course, the redundancy of $N = 1$ has the highest overhead for increasing disk availability.

The higher throughput, measured either as megabytes per second or as I/Os per second, and the ability to recover from failures make disk arrays attractive. When combined with the advantages of smaller volume and lower power of small-diameter drives, redundant arrays of small or inexpensive drives may play a larger role in future disk systems. The current drawback is the added complexity of a controller for disk arrays.

Graphics Displays

Through computer displays I have landed an airplane on the deck of a moving carrier, observed a nuclear particle hit a potential well, flown in a rocket at nearly the speed of light and watched a computer reveal its innermost workings.

Ivan Sutherland (the "father" of computer graphics), quoted in "Computer Software for Graphics," *Scientific American* (1984)

While magnetic disks may dominate throughput and cost of I/O devices, the most fascinating I/O device is the graphics display. Based on television technology, a *raster cathode ray tube* (CRT) *display* scans an image out one line at a time, 30 to 60 times per second. At this *refresh rate* the human eye doesn't notice a "flicker" on the screen. The image is composed of a matrix of picture elements, or *pixels*, which can be represented as a matrix of bits, called a *bit map*. Depending on size of screen and resolution, the display matrix consists of 340*512 to 1560*1280 pixels. For black and white displays, often 0 is black and 1 is white. For displays that support over 100 different shades of black and white, sometimes called *gray-scale* displays, 8 bits per pixel are required. A color display might use 8 bits for each of the three primary colors (red, blue, and green), for 24 bits per pixel.

The hardware support for graphics consists mainly of a *raster refresh buffer*, or *frame buffer*, to store the bit map. The image to be represented on screen is stored into the frame buffer, and the bit pattern per pixel is read out to the graphics display at the refresh rate. Figure 9.17 (page 522) shows a frame buffer with four bits per pixel and Figure 9.18 (page 522) shows how the buffer is connected to the bus.

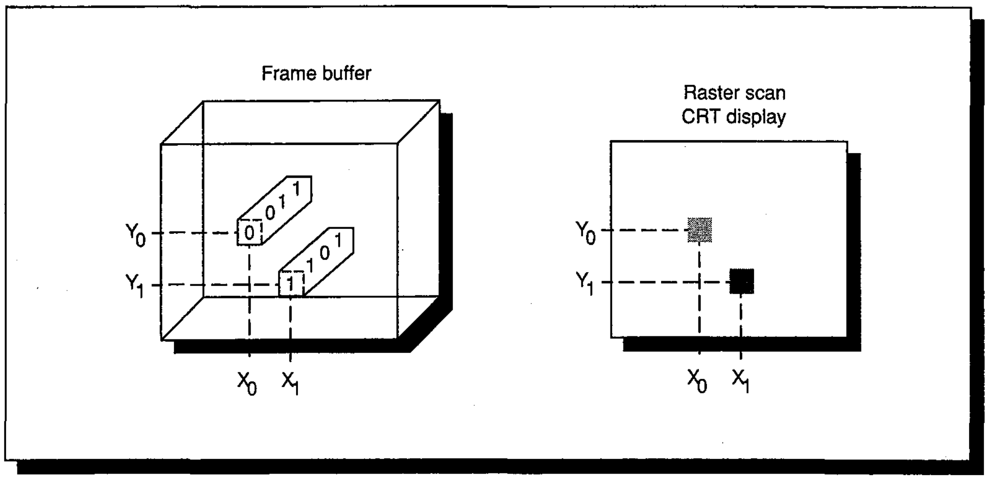

FIGURE 9.17 Each coordinate in the frame buffer on the left determines the shade of the corresponding coordinate for the raster scan CRT display on the right. Pixel (x_0,y_0) contains the bit pattern 0011, which is a lighter shade of gray on the screen than the bit pattern 1101 in pixel (x_1,y_1).

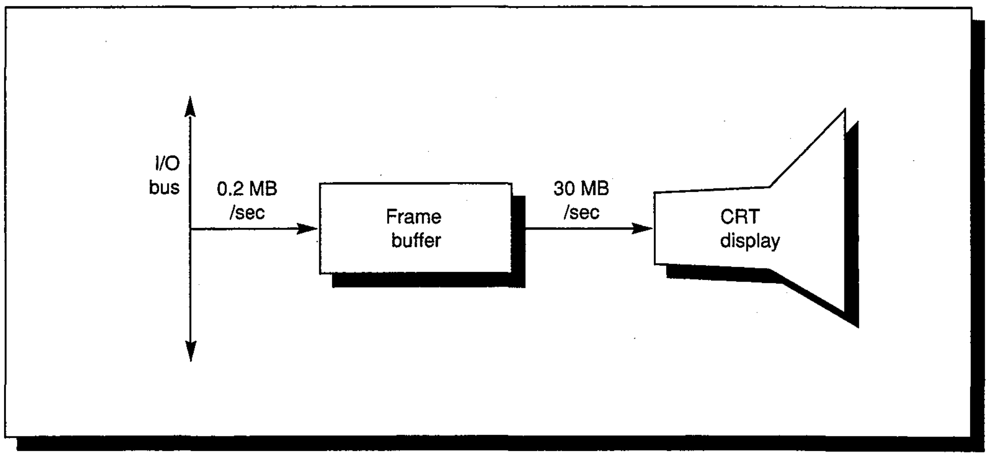

FIGURE 9.18 The frame buffer is connected to both the I/O bus and the display. Because of the high data rate from the buffer to the display, the frame buffer is frequently dual ported.

The goal of the bit map is to faithfully represent what is on the screen. As the computer switches from one image to another, the screen may look "splotchy" during the change. Here are two ways of dealing with this:

- Change the frame buffer only during the "vertical blanking interval." This is the time the gun in the raster CRT display takes to go back to the upper-left-hand corner before starting to paint the pixels of the next image. This takes 1 to 2 ms of every 16 ms at the 60-Hz refresh rate each time the screen is painted.

- If the vertical blanking interval is not long enough, the frame buffer can be double buffered, so that one is read while the other is being written. This way, images in sequence (as in animation) are drawn in alternate frame buffers. Double buffering, of course, doubles the cost of the memory in the frame buffer.

From the point of view of the CPU, graphics is logically output only. But the frame buffer is capable of being read as well as written, permitting operations to be performed directly on the screen images. These operations are called *bit blts*, for bit block transfer. Bit blts are commonly used for operations such as moving a window or changing the shape of the cursor. A current debate in graphics architecture is whether reading the frame buffer is limited to the operating system or should user programs be able to read it as well.

Cost of Computer Graphics

The CRT monitor itself is based on television technology and is sensitive to consumer demand. Today prices vary from $100 for a black-and-white monitor to $15,000 for a large studio color monitor, not including memory. The amount of memory in a frame buffer depends directly on the size of the screen and the bits per pixel:

$$340*512*1 \text{ bits} = 21.5 \text{ KB}$$

$$1280*1024*24 \text{ bits} = 3840 \text{ KB}$$

(By the way, this bottom dimension is the proposed size for high-definition television.) Note that the memory cost is doubled if double buffering is used.

To reduce costs of a color frame buffer, many systems use a two-level representation that takes advantage of the fact that few pictures need the full pallet of possible colors (see Figure 9.19 on page 524).

The intermediate level contains the full color width of, say, 24 bits and a large collection of the possible colors that can appear on the screen—256 different colors, for example. While this collection is large, it is still much smaller than 2^{24}. This intermediary table has been variously named a *color map*, *color table*, or *video look-up table*. Each pixel need have only enough bits to indicate a color in the color map. As a simple example, Figure 9.19 uses a 4-word color map, which means the frame buffer needs only 2 bits per pixel. The savings for a full-sized color display with a 256-color map is

$$1280*1024*24 - (1280*1024*8 + 256*24)$$

$$= 3{,}840 \text{ KB} - (1280 \text{ KB} + .75 \text{ KB}) \approx 2560 \text{ KB}$$

This amounts to a threefold reduction in memory size. In 1990 a 256- by 24-bit color map and an analog interface to a color CRT fit in a single chip.

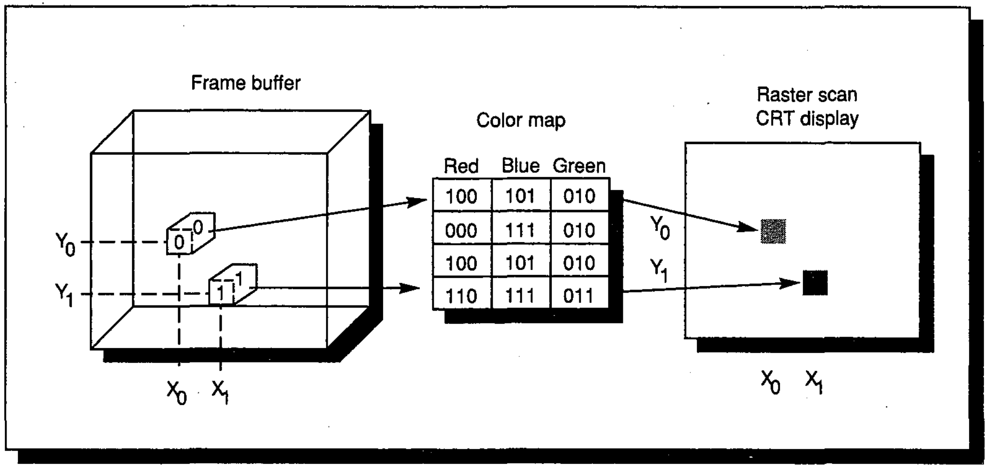

FIGURE 9.19 An example of a color map to reduce the cost of the frame buffer. Suppose only nine bits per color are needed. Rather than store the full nine bits per pixel in the frame buffer, just enough bits per pixel are stored to index the table containing the unique colors in a picture. Only the color map has the nine bits for the colors in the display. Near photographic color pictures can be produced with about 125 colors using the right shades of the color spectrum; but at least 24 bits are needed to get the right shades! The color map is loaded by the application program, offering each picture its own palette of colors to chose from.

Performance Demands of Graphics Displays

The performance of graphics is determined by the frequency an application needs new images and by the quality of those images. The amount of information transferred from memory to the frame buffer depends on complexity of image, with a full color display requiring almost four megabytes. The transfer rate depends on the speed with which the image should be changed as well as the amount of information. Animation requires at least 15 changes per second for movement to appear smooth on a screen. For interactive graphics, the time to update the frame buffer measures the effectiveness of the application; for people to feel comfortable the total reaction time must be less than a second (see Figure 9.9, page 510). With a drawing system, the portion of the screen one is working on must change almost immediately, as human visual perception is on the order of 0.02 seconds. Figure 9.20 shows some sample graphics tasks and their performance requirements. Note that the frame buffer must have enough bandwidth to refresh the display **and** to allow the CPU to change the image being refreshed.

The high data rate—and the large market of graphics displays—has made a dual-ported DRAM chip popular. This chip has a serial I/O port and internal shift register that is connected to the display in a graphics application in addition to the traditional randomly addressed data port. This chip is so widely used in frame buffers that it is called a *video DRAM.*

Graphics tasks	Bandwidth requirements
Text editor—Scrolling text in window means moving all bits in half the frame buffer about 10 times per second.	0.8 MB/sec
VLSI design—Moving a portion of the design means moving all bits in half of a color frame buffer in less than 0.1 second.	6.3 MB/sec
Television commercial—Showing movie-quality images means changing 24 times per second.	90.0 MB/sec
Visualization of scientific data—About the same as a television commercial.	90.0 MB/sec

FIGURE 9.20 Graphics tasks and their performance requirements. VLSI design uses 8 bits of color while the television commercial and visualization use 24 bits. Bandwidth is measured at the frame buffer.

Future Directions in Graphics Displays

It is safe to predict that people will want better pictures in the future. They will want, for example, more lines on a screen and more bits per inch on a line to make sharper images, more bits per color to make more colorful images, and more bandwidth to allow animation.

To simplify the display of three-dimensional images, a z dimension per pixel can be added to the x and y coordinates. It says where the pixel is located from the viewer along a z axis (e.g., into the CRT). A 3D image starts with z set to the furthest possible location from the viewer and the color set to the background color. To get a proper 3D perspective, the z coordinate stored with the pixel in the frame buffer is checked before placing a color in a pixel. If the new color is closer, the old color is replaced and the z coordinate is updated; if it is further away, the new color is discarded. This scheme is called a *z buffer* approach to *hidden surface elimination.* It adds at least 8 bits per pixel, plus the performance cost of reading and comparing before writing a pixel. The Silicon Graphics 4D series of graphics workstations uses 16 bits for the z dimension in its pixels, meaning objects are assigned a 16-bit number to show how close they are to the viewer.

The increasing number of bits per DRAM chip reduces the number of chips needed in the frame buffer, as well as the number of chips that can simultaneously transfer bits to the screen. This is why video DRAMS are so popular. As capacity increases, the serial ports of video DRAMs will have to become faster and wider to match the demands of future graphics systems.

Networks

There is an old network saying: Bandwidth problems can be cured with money. Latency problems are harder because the speed of light is fixed—you can't bribe God.

David Clark, M.I.T.

Networks are the backbone of current computer systems; a new machine without an optional network interface would be ridiculed. By connecting computers electronically, networked computers have these advantages:

- *Communication*—Information is exchanged between computers at high speeds.
- *Resource sharing*—Rather than each machine having its own I/O devices, devices can be shared by computers on the network.
- *Nonlocal access*—By connecting I/O devices over long distances, users need not be near the computer they are using.

Figure 9.21 shows the characteristics of networks. These characteristics are illustrated below with three examples.

Distance	0.01 to 10,000 kilometers
Speed	0.001 MB/sec to 100 MB/sec
Topology	Bus, ring, star, tree
Shared lines	None (point-to-point) or shared (multidrop)

FIGURE 9.21 Range of network characteristics.

The RS232 standard provides a 0.3- to 19.2-Kbits-per-second *terminal network*. A central computer connects to many terminals over slow but cheap dedicated wires. These point-to-point connections form a star from the central computer, with each terminal ranging from 10 to 100 meters in distance from the computer.

The *local area network*, or LAN, is what is commonly meant today when people mention a network, and *Ethernet* is what most people mean when they mention a LAN. (Ethernet has in fact become such a common term that it is often used as a generic term for LAN.) The Ethernet is essentially a 10,000 Kbits-per-second bus that has no central control. Messages or *packets* are sent over the Ethernet in blocks that vary from 128 bytes to 1530 bytes and take 0.1 ms and 1.5 ms to send, respectively. Since there is no central control, all nodes "listen" to see if there is a message for that node. Without a central arbiter to decide who gets the bus, a computer first listens to make sure it doesn't send a message while another message is on the network. If the network is idle the node tries to send. Of course, some other node may decide to send at the same instant. Luckily, the computer can detect any resulting collisions by listening to what is sent. (Mixed messages will sound like garbage.) To avoid repeated head-on collisions, each node whose packet was trashed backs off a random time before resending. If Ethernets do not have high utilization, this simple approach to arbitration works well. Many LANs become overloaded through poor capacity planning, and response time and throughput can degrade rapidly at higher utilization.

The success of LANs has led to multiples of them at a single site. Connecting computers to separate Ethernets becomes necessary at a certain point because there is a limit to the number of nodes that can be active on a bus if effective communication speeds are to be achieved; one limit is 1024 nodes per Ethernet. There is also a physical limit to the distance of an Ethernet, usually about 1 kilometer. To allow Ethernets to work together, two kinds of devices have been created:

- A *bridge* connects two Ethernets. There are still two independent buses that can simultaneously send messages, but the bridge acts as a filter, allowing only those messages from nodes on one bus to nodes on the other bus to cross over the bridge.
- A *gateway* typically connects several Ethernets. It receives a message, looks up the destination address in a table, and then routes the message over the appropriate network to the proper node. This *routing table* can be changed during execution to reflect the state of the networks. Some use the term *router* instead of gateway since it is closer to the function performed.

When Ethernets are connected together with gateways they form an *Internet.*

Long-haul networks cover distances of 10 to 10,000 kilometers. The first and most famous long-haul network was the ARPANET (named after its funding agency, the Advanced Research Projects Agency of the U.S. government). It transferred at 50 Kbits per second and used point-to-point dedicated lines leased from telephone companies. The host computer talked to an *interface message processor* (IMP), which communicated over the telephone lines. The IMP took information and broke it into 1-Kbit packets. At each hop the packet was stored and then forwarded to the proper IMP according to the address in the packet. The destination IMP reassembled the packets into a message and then gave it to the host. *Fragmentation and reassembly*, as it was called, was done to reduce the latency due to the *store and forward delay*. Most networks today use this *packet switched* approach, where packets are individually routed from source to destination. Figure 9.22 (page 528) summarizes the performance, distance, and costs of these various networks.

While these networks have been presented here as alternatives, a computer system is really a hierarchy of networks, as Figure 9.23 (page 528) shows. To deal with this hierarchy of networks connecting machines that communicate differently, there must be a standard software interface to handle messages. These are called *protocols*, and are typically layered to interface with different levels of software in computer systems. The overhead of these protocols can eat up a significant portion of the network bandwidth.

Just as with disks in Figure 9.6 (page 507), there is a tradeoff of latency and throughput in networks. Small messages give the lowest latency in most networks, but they also result in lower network bandwidth; similarly, a network can achieve higher bandwidth at the cost of longer latency.

Network	Performance (Kbits / sec)	Distance (km)	Cable cost	Connect to network cost	Connector to computer cost
RS232	19	0.1	$0.25 /foot	$1–$5 /connector	$5 /serial port chip
Ethernet	10,000	1	$1–$5 /foot	$100 /transceiver	$50 /Ethernet interface chip
ARPANET	50	10,000	$10,000 /month	$50,000–$100,000/ IMP	$5,000–$10,000 /IMP connection

FIGURE 9.22 The performance, maximum distance, and costs of three example networks. An Internet is simply multiple Ethernets and a bridge, which costs about $2,000 to $5,000, or a gateway, which costs about $20,000 to $50,000.

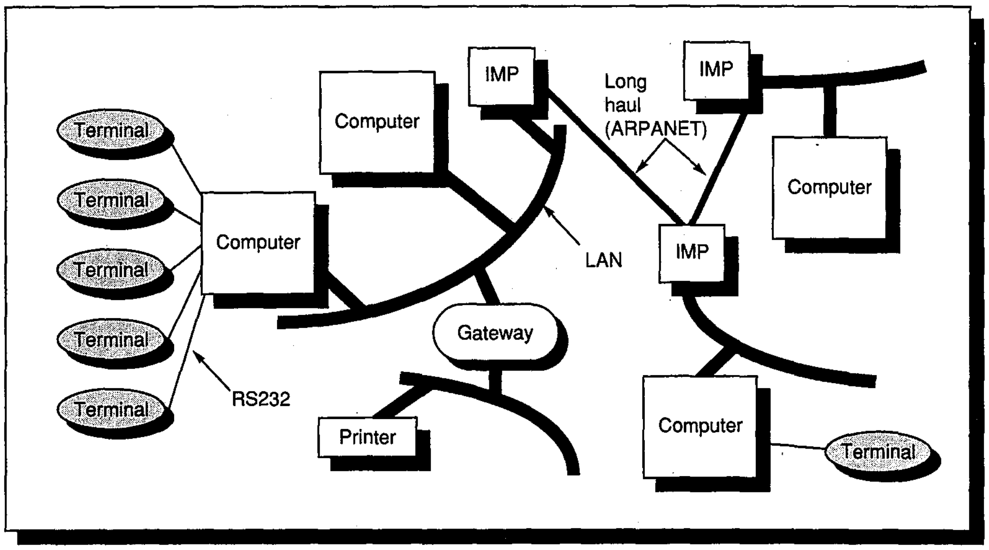

FIGURE 9.23 A computer system today participates in a hierarchy of networks. Ideally, the user is not aware of what network is being used in performing tasks. The gateway routes packets to a particular network, a network routes packets to a particular host computer, and the host computer routes packets to a particular process.

9.5 Buses—Connecting I/O Devices to CPU/Memory

In a computer system, the various subsystems must have interfaces to one another; for instance, the memory and CPU need to communicate, as well as the CPU and I/O devices. This is commonly done with a *bus*. The bus serves as a shared communication link between the subsystems. The two major advantages of the bus organization are low cost and versatility. By defining a single interconnection scheme, new devices can easily be added, and peripherals may even be ported between computer systems that use a common bus. The cost is low, since a single set of wires is shared multiple ways.

The major disadvantage of a bus is that it creates a communication bottleneck, possibly limiting the maximum I/O throughput. When I/O must pass through a central bus this bandwidth limitation is as real as—and sometimes more severe than—memory bandwidth. In commercial systems, where I/O is very frequent, and in supercomputers, where the necessary I/O rates are very high because the CPU performance is high, designing a bus system capable of meeting the demands of the processor is a major challenge.

One reason bus design is so difficult is that the maximum bus speed is largely limited by physical factors: the length of the bus and the number of devices (and, hence, bus loading). These physical limits prevent arbitrary bus speedup. The desire for high I/O rates (low latency) and high I/O throughput can also lead to conflicting design requirements.

Buses are traditionally classified as *CPU–memory buses* or *I/O buses*. I/O buses may be lengthy, may have many types of devices connected to them, have a wide range in the data bandwidth of the devices connected to them (see Figure 9.1 on page 501), and normally follow a bus standard. CPU–memory buses, on the other hand, are short, generally high speed, and matched to the memory system to maximize memory–CPU bandwidth. During the design phase, the designer of a CPU–memory bus knows all the types of devices that must connect together, while the I/O bus designer must accept devices varying in latency and bandwidth capabilities. To lower costs, some computers have a single bus for both memory and I/O devices.

Let's consider a typical bus transaction. A *bus transaction* includes two parts: sending the address and receiving or sending the data. Bus transactions are usually defined by what they do to memory: A *read* transaction transfers data *from* memory (to either the CPU or an I/O device), and a *write* transaction writes data to the memory. In a read transaction, the address is first sent down the bus to the memory, together with the appropriate control signals indicating a read. The memory responds by returning the data on the bus with the appropriate control signals. A write transaction requires that the CPU or I/O device send both address and data and requires no return of data. Usually the CPU must wait between sending the address and receiving the data on a read, but the CPU often does not wait on writes.

The design of a bus presents several options, as Figure 9.24 (page 530) shows. Like the rest of the computer system, decisions will depend on cost and performance goals. The first three options in the figure are clear choices—separate address and data lines, wider data lines, and multiple-word transfers all give higher performance at more cost.

The next item in the table concerns the number of *bus masters*. These are devices that can initiate a read or write transaction; the CPU, for instance, is always a bus master. A bus has multiple masters when there are multiple CPUs or when I/O devices can initiate a bus transaction. If there are multiple masters, an arbitration scheme is required among the masters to decide who gets the bus next. Arbitration is often a fixed priority, as is the case with daisy-chained devices or an approximately fair scheme that randomly chooses which master gets the bus.

With multiple masters a bus can offer higher bandwidth by going to packets, as opposed to holding the bus for the full transaction. This technique is designated *split transactions*. (Some systems call this ability *connect/disconnect* or a *pipelined bus*.) The read transaction is broken into a read-request transaction that contains the address, and a memory-reply transaction that contains the data. Each transaction must now be tagged so that the CPU and memory can tell what is what. Split transactions make the bus available for other masters while the memory reads the words from the requested address. It also normally means that the CPU must arbitrate for the bus to send the data and the memory must arbitrate for the bus to return the data. Thus, a split-transaction bus has higher bandwidth, but it usually has higher latency than a bus that is held during the complete transaction.

The final item, clocking, concerns whether a bus is synchronous or asynchronous. If a bus is *synchronous* it includes a clock in the control lines and a fixed protocol for address and data relative to the clock. Since little or no logic is needed to decide what to do next, these buses can be both fast and inexpensive. However, they have two major disadvantages. Everything on the bus must run at the same clock rate, and because of clock-skew problems, synchronous buses cannot be long. CPU–memory buses are typically synchronous.

An *asynchronous* bus, on the other hand, is not clocked. Instead, self-timed, handshaking protocols are used between bus sender and receiver. This scheme makes it much easier to accommodate a wide variety of devices and to lengthen the bus without worrying about clock skew or synchronization problems. If a synchronous bus can be used, it is usually faster than an asynchronous bus because of the overhead of synchronizing the bus for each transaction. The choice of synchronous versus asynchronous bus has implications not only for data bandwidth but also for an I/O system's capacity in terms of physical

Option	**High performance**	**Low cost**
Bus width	Separate address and data lines	Multiplex address and data lines
Data width	Wider is faster (e.g., 32 bits)	Narrower is cheaper (e.g., 8 bits)
Transfer size	Multiple words has less bus overhead	Single-word transfer is simpler
Bus masters	Multiple (requires arbitration)	Single master (no arbitration)
Split transaction?	Yes—separate Request and Reply packets gets higher bandwidth (needs multiple masters)	No—continuous connection is cheaper and has lower latency
Clocking	Synchronous	Asynchronous

FIGURE 9.24 The main options for a bus. The advantage of separate address and data buses is primarily on writes.

distance and number of devices that can be connected to the bus; asynchronous buses scale better with technological changes. I/O buses are typically asynchronous. Figure 9.25 suggests the relationship of when to use one over the other.

Bus Standards

The number and variety of I/O devices are not fixed on most computer systems, permitting customers to tailor computers to their needs. As the interface to which devices are connected, the I/O bus can also be considered an expansion bus for adding I/O devices over time. Standards that let the computer designer and I/O-device designer work independently, therefore, play a large role in determining the choice of buses. As long as both the computer-system designer and the I/O-device designer meet the requirements, any I/O device can connect to any computer. In fact, an I/O bus standard is the document that defines how to connect them.

Machines sometimes grow to be so popular that their I/O buses become de facto standards; examples are the PDP-11 Unibus and the IBM PC-AT Bus. Once many I/O devices have been built for the popular machine, other computer designers will build their I/O interface so that those devices can plug into their machines as well. Sometimes standards also come from an explicit standards effort on the part of I/O device makers. The *intelligent peripheral interface* (IPI)

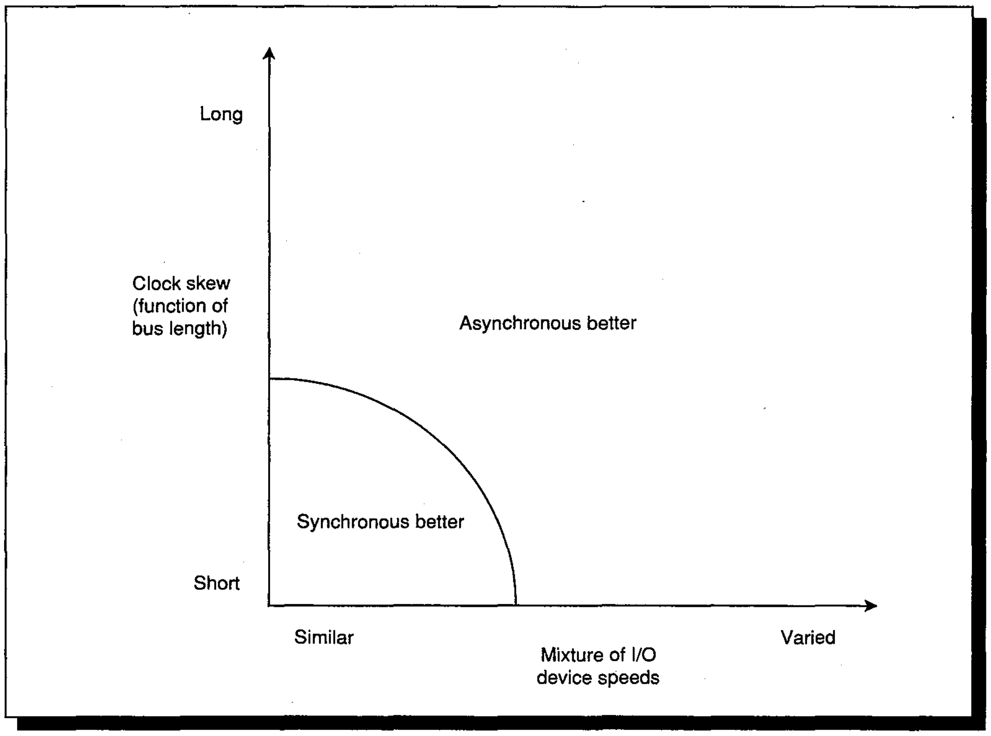
Long
Clock skew (function of bus length)
Short
Asynchronous better
Synchronous better
Similar
Mixture of I/O device speeds
Varied

FIGURE 9.25 Preferred bus type as a function of length/clock skew and variation in I/O device speed. Synchronous is best when the distance is short and the I/O devices on the bus all transfer at similar speeds.

and Ethernet are examples of standards from cooperation of manufacturers. If standards are successful, they are eventually blessed by a sanctioning body like ANSI or IEEE. Occasionally, a bus standard comes top-down directly from a standards committee—the FutureBus is one example.

Figure 9.26 summarizes characteristics of several bus standards. Note that the bandwidth entries in the figure are not listed as single numbers for the CPU–memory buses (VME, FutureBus, and Multibus II). Because of the bus overhead, the size of the transfer affects bandwidth significantly. Since the bus usually transfers to or from memory, the speed of the memory also affects the bandwidth. For example, with infinite transfer size and infinitely fast (0 ns) memory, FutureBus is 240% faster than VME, but FutureBus is only about 20% faster than VME for single-word transfers from a 150-ns memory.

	VME bus	FutureBus	Multibus II	IPI	SCSI
Bus width (signals)	128	96	96	16	8
Address/data multiplexed?	Not multiplexed	Multiplexed	Multiplexed	N/A	N/A
Data width (primary)	16 to 32 bits	32 bits	32 bits	16 bits	8 bits
Transfer size	Single or multiple	Single or multiple	Single or multiple	Single or multiple	Single or multiple
Number of bus masters	Multiple	Multiple	Multiple	Single	Multiple
Split transaction?	No	Optional	Optional	Optional	Optional
Clocking	Asynchronous	Asynchronous	Synchronous	Asynchronous	Either
Bandwidth, 0-ns access memory, single word	25.0 MB/sec	37.0 MB/sec	20.0 MB/sec	25.0 MB/sec	5.0 MB/sec or 1.5 MB/sec
Bandwidth, 150-ns access memory, single word	12.9 MB/sec	15.5 MB/sec	10.0 MB/sec	25.0 MB/sec	5.0 MB/sec or 1.5 MB/sec
Bandwidth, 0-ns access memory, multiple words (infinite block length)	27.9 MB/sec	95.2 MB/sec	40.0 MB/sec	25.0 MB/sec	5.0 MB/sec or 1.5 MB/sec
Bandwidth, 150-ns access memory, multiple words (infinite block length)	13.6 MB/sec	20.8 MB/sec	13.3 MB/sec	25.0 MB/sec	5.0 MB/sec or 1.5 MB/sec
Maximum number of devices	21	20	21	8	7
Maximum bus length	0.5 meter	0.5 meter	0.5 meter	50 meters	25 meters
Standard	IEEE 1014	IEEE 896.1	ANSI/IEEE 1296	ANSI X3.129	ANSI X3.131

FIGURE 9.26 Information on five bus standards. The first three were defined originally as CPU–memory buses and the last two as I/O buses. For the CPU–memory buses the bandwidth calculations assume a fully loaded bus and are given for both single-word transfers and block transfers of unlimited length; measurements are shown both ignoring memory latency and assuming 150-ns access time. Bandwidth assumes the average distance of a transfer is one-third of the backplane length. (Data in the first three columns is from Borrill [1986].) The bandwidth for the I/O buses is given as their maximum data-transfer rate. The SCSI standard offers either asynchronous or synchronous I/O; the asynchronous version transfers at 1.5 MB/sec and the synchronous at 5 MB/sec.

9.6 Interfacing to the CPU

Having described I/O devices and looked at some of the issues of the connecting bus, we are ready to discuss the CPU end of the interface. The first question is how the physical connection of the I/O bus should be made. The two choices are connecting it to memory or to the cache. In the following section we will discuss the pros and cons of connecting an I/O bus directly to the cache; in this section we examine the more usual case in which the I/O bus is connected to the main memory bus. Figure 9.27 shows a typical organization. In low-cost systems, the I/O bus **is** the memory bus; this means an I/O command on the bus could interfere with a CPU instruction fetch, for example.

Once the physical interface is chosen, the question becomes how does the CPU address an I/O device that it needs to send or receive data. The most common practice is called *memory-mapped* I/O. In this scheme, portions of the address space are assigned to I/O devices. Reads and writes to those addresses may cause data to be transferred; some portion of the I/O space may also be set aside for device control, so commands to the device are just accesses to those memory-mapped addresses. The alternative practice is to use dedicated I/O opcodes in the CPU. In this case, the CPU sends a signal that this address is for I/O devices. Examples of computers with I/O instructions are the Intel 80x86 and the IBM 370 computers. No matter which addressing scheme is selected, each I/O device has registers to provide status and control information. Either

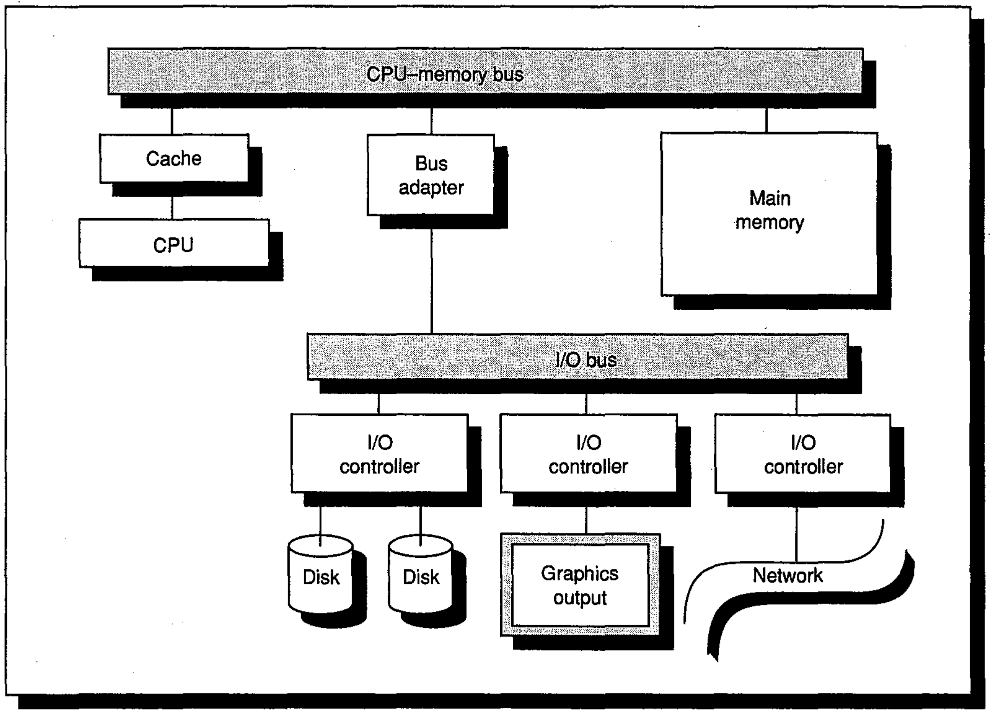

FIGURE 9.27 A typical interface of I/O devices and an I/O bus to the CPU–memory bus.

through loads and stores in memory-mapped I/O or through special instructions, the CPU sets flags to determine the operation the I/O device will perform.

I/O is rarely a single operation. For example, the DEC LP11 line printer has two I/O device registers: one for status information and one for data to be printed. The status register contains a *done bit*, set by the printer when it has printed a character, and an *error bit*, indicating that the printer is jammed or out of paper. Each byte of data to be printed is put into the data register; the CPU must then wait until the printer sets the done bit before it can place another character in the buffer.

This simple interface, in which the CPU periodically checks status bits to see if it is time for the next I/O operation, is called *polling*. As one might expect, the fact that CPUs are so much faster than I/O devices means polling may waste a lot of CPU time. This was recognized long ago, leading to the invention of interrupts to notify the CPU when it is time to do something for the I/O device. *Interrupt-driven* I/O, used by most systems for at least some devices, allows the CPU to work on some other process while waiting on the I/O device. For example, the LP11 has a mode that allows it to interrupt the CPU whenever the done bit or error bit is set. In general-purpose applications, interrupt driven I/O is the key to multitasking operating systems and good response times.

The drawback to interrupts is the operating system overhead on each event. In real-time applications with hundreds of I/O events per second, this overhead can be intolerable. One hybrid solution for real-time systems is to use a clock to periodically interrupt the CPU, at which time the CPU polls all I/O devices.

Delegating I/O Responsibility from the CPU

Interrupt-driven I/O relieves the CPU from waiting for every I/O event, but there are still many CPU cycles spent in transferring data. Transferring a disk block of 2048 words, for instance, would require at least 2048 loads and 2048 stores, as well as the overhead for the interrupt. Since I/O events so often involve block transfers, *direct memory access* (DMA) hardware is added to many computer systems to allow transfers of numbers of words without intervention by the CPU.

DMA is a specialized processor that transfers data between memory and an I/O device, while the CPU goes on with other tasks. Thus, it is external to the CPU and must act as a master on the bus. The CPU first sets up the DMA registers, which contain a memory address and number of bytes to be transferred. Once the DMA transfer is complete, the controller interrupts the CPU. There may be multiple DMA devices in a computer system; for example, DMA is frequently part of the controller for an I/O device.

Increasing the intelligence of the DMA device can further unburden the CPU. Devices called *I/O processors*, (or *I/O controllers*, or *channel controllers*) operate from either fixed programs or from programs downloaded by the operating system. The operating system typically sets up a queue of *I/O control* *blocks* that contain information such as data location (source and destination) and data size. The I/O processor then takes items from the queue, doing everything requested and sending a single interrupt when the task specified in the I/O control blocks is complete. Whereas the LP11 line printer would cause 4800 interrupts to print a 60-line by 80-character page, an I/O processor could save 4799 of those interrupts.

I/O processors can be compared to multiprocessors in that they facilitate several processes executing simultaneously in the computer system. I/O processors are less general than CPUs, however, since they have dedicated tasks, and thus parallelism is also much more limited. Also, an I/O processor doesn't normally change information, as a CPU does, but just moves information from one place to another.

9.7 Interfacing to an Operating System

In a manner analogous to the way compilers use an instruction set (see Section 3.7 of Chapter 3), operating systems control what I/O techniques implemented by the hardware will actually be used. For example, many I/O controllers used in early UNIX systems were 16-bit microprocessors. To avoid problems with 16-bit addresses in controllers, UNIX was changed to limit the maximum I/O transfer to 63 KB or less; at the time of this book's publication, that limit is still in effect. Thus, a new I/O controller designed to efficiently transfer 1-MB files would never see more than 63 KB at a time under UNIX, no matter how large the files.

Caches Cause Problems for Operating Systems—Stale Data

The prevalence of caches in computer systems has added to the responsibilities of the operating system. Caches imply the possibility of two copies of the data—one each for cache and main memory—while virtual memory can result in three copies—for cache, memory and disk. This brings up the possibility of *stale data*: the CPU or I/O system could modify one copy without updating the other copies (see Section 8.8 in Chapter 8). Either the operating system or the hardware must make sure that the CPU reads the most recently input data and that I/O outputs the correct data, in the presence of caches and virtual memory. Whether the stale-data problem arises depends in part on where the I/O is connected to the computer. If it is connected to the CPU cache, as shown in Figure 9.28 (page 536), there is no stale-data problem; all I/O devices and the CPU see the most accurate version in the cache, and existing mechanisms in the memory hierarchy ensure that other copies of the data will be updated. The side effect is lost CPU performance, since I/O will replace blocks in the cache with data that are unlikely to be needed by the process running in the CPU at the time of the transfer. In other words, all I/O data goes through the cache but little of it is referenced. This arrangement also requires arbitration between CPU and I/O to decide who accesses the cache. If I/O is connected to memory, as in Figure 9.27 (page 533), then it doesn't interfere with CPU, provided the CPU has a cache. In this situation, however, the stale-data problem occurs. Alternatively, I/O can just invalidate data—either all data that might match (no tag check) or only data that matches.

There are two parts to the stale-data problem:

1. The I/O system sees stale data on output because memory is not up to date.
2. The CPU sees stale data in the cache on input after the I/O system has updated memory.

The first dilemma is how to output correct data if there is a cache and I/O is connected to memory. A write-through cache solves this by ensuring that memory will have the same data as the cache. A write-back cache requires the operating system to flush output addresses to make sure they are not in the cache. This takes time, even if the data is not in the cache, since address checks are sequential. Alternatively, the hardware can check cache tags during output to see if they are in a write-back cache, and only interact with the cache if the output tries to read data that is in the cache.

The second problem is ensuring that the cache won't have stale data after input. The operating system can guarantee that the input data area can't possibly

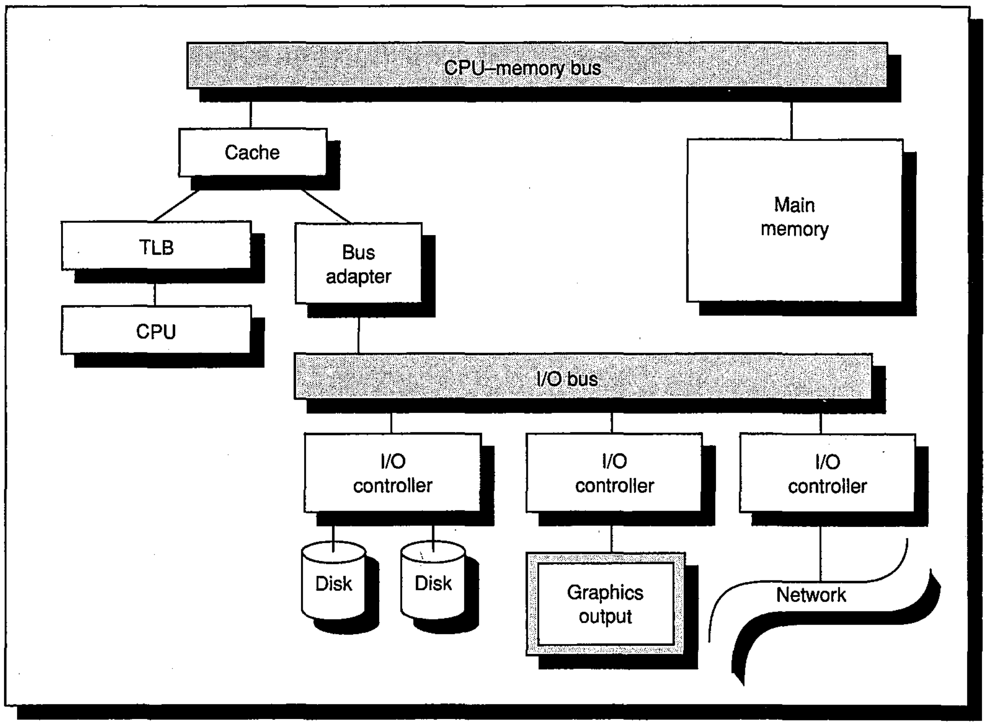

FIGURE 9.28 Example of I/O connected directly to the cache.

be in the cache. If it can't guarantee this, the operating system flushes input addresses to make sure they are not in the cache. Again, this takes time, whether or not the input addresses are in the cache. As before, extra hardware can be added to check tags during an input and invalidate the data if there is a conflict. These problems are basically the same as cache coherency in a multiprocessor, discussed in Section 8.8 of Chapter 8; I/O can be thought of as a second dedicated processor in a multiprocessor.

DMA and Virtual Memory

Given the use of virtual memory, there is the matter of whether DMA should transfer using virtual addresses or physical addresses. Here are some problems with DMA using physically mapped I/O:

- Transferring a buffer that is larger than one page will cause problems, since the pages in the buffer will not usually be mapped to sequential pages in physical memory.
- Suppose DMA is ongoing between memory and a frame buffer, and the operating system removes some of the pages from memory (or relocates them). The DMA would then be transferring data to or from the wrong page of memory.

One answer to these questions is *virtual DMA*. It allows the DMA to use virtual addresses that are mapped to physical addresses during the DMA. Thus, a buffer must be sequential in virtual memory but the pages can be scattered in physical memory. The operating system could update the address tables of a DMA if a process is moved using virtual DMA, or the operating system could "lock" the pages in memory until the DMA is complete. Figure 9.29 (page 538) shows address-translation registers added to the DMA device.

Caches Helping Operating Systems— File or Disk Caches

While the invention of caches made the life of the operating systems designer more difficult, operating systems designers' concern for performance led them to cache-like optimizations, using main memory as a "cache" for disk traffic to improve I/O performance. The impact of using main memory as a buffer or cache for file or disk accesses is demonstrated in Figure 9.30 (page 538). It shows the change in disk I/Os for a cacheless system measured as miss rate (see Section 8.2 in Chapter 8). File caches or disk caches change the number of disk I/Os and the mix of reads and writes; depending on cache size and write policy, between 50% to 70% of all disk accesses could become writes with such caches. Without file or disk caches, between 15% and 33% of all accesses are writes, depending on the environment.