What is the hard deadline associated with this process? Let us assume for simplicity that the controller delay (i.e., the computer response time) is solely in recognizing that an impact has occurred and that the other reaction times are zero (e.g., that the thrusters can be turned on instantaneously). If this delay is too long, the controller may not be able to stop the body from moving out of the allowed state space.

What is the cost function associated with this process? Let us define the accomplishment level as the energy expended by the system in getting the body back to the ideal point as soon as possible. The longer the controller takes to respond to the change in velocity, the more work ultimately needs to be done in getting the body back to rest at the ideal point and the more energy is expended.

There is a further observation that is worth making. The hard deadline and cost functions are functions of the current state of the controlled process, that is, of the current position, velocity, and acceleration of the body. In general, the closer the body is to the boundary of the allowed state space or the faster it is moving away from the ideal point, the shorter the hard deadline.

Let us make these intuitive points more concrete by computing the hard deadlines and cost functions at several points in the allowed state space. Recall that the state of the process is defined by a three-tuple (x, v, a), where x, v, and a are its current position, velocity and acceleration, respectively.

Consider the process in state $(0, 0, 0)$—the body is already at the ideal point, has zero velocity, and is not accelerating. In such a case, the controller has nothing to accomplish; the body is already where it is supposed to be. Consequently, the hard deadline is infinity and the cost function is zero throughout. This is a formal way of saying that as long as the body is where it is supposed to be, the controller does not have to do anything.

Consider the process in state $(x, 0, 0)$—the body is stationary at point x in the allowed state space. The controller brings it back to rest at the ideal point. Since the body is not moving, no amount of controller delay will cause it to move out of the allowed state space. Consequently, the hard deadline is infinity. The energy that the controller expends in bringing back the body to rest at the ideal point is also not a function of how long the controller takes to react; the energy expended when the controller delay is ξ is exactly the same as that when it is 0. Hence, the cost function is zero in this case also.

Consider the process in state $(x, v, 0)$—the body is in position x, has velocity v, and is not accelerating. Assume for this example that both x and v are positive. The controller response time is ξ; that is, it takes ξ time for the real-time computer to realize that an impact has occurred and must be responded to. By this time, the body is in state $(x + v\xi, v, 0)$. The controller first must stop the movement of the body in the positive direction. This it can do by deploying a thrust of H in the negative direction, which will yield an acceleration of $-a = -H/m$. The basic equations of motion can be used to show that the body will come to rest at position $x_1 = x + v\xi + v^2/2a$. If $x_1 > b$, then the controlled process fails. This happens whenever

$$x + v\xi + v^2/2a > b \Rightarrow \xi > \frac{b - x - v^2/2a}{v}$$

That is, the hard deadline is

$$t_d(x, v, 0) = \frac{b - x - v^2/2a}{v}$$

What is the cost function? This can be computed by studying the distance over which the thrust has to be maintained. The body is stopped at x_1, which means that it has been decelerated over the interval $x_1 - x$. From this point, the body is accelerated towards the origin at maximal thrust until position $x_1/2$, and then it is decelerated at maximal thrust until it comes to rest at the origin. That is, the thrust of H has to be maintained over a distance of $d_\xi = x_1 - x + x_1 = 2x_1 - x$, which corresponds to an energy expenditure of $d_\xi H$. What is the energy expenditure if the response time is zero? In such a case, as the reader can easily check, the expended energy is $H(x + v^2/a)$. The cost function of this task is then given by the difference in the energies

$$C_{x,v,0}(\xi) = H(2x_1 - x) - H\left(\frac{x + v^2}{a}\right)$$
$$= v\xi H$$

The preceding example shows that the cost functions and hard deadlines depend on the state of the controlled process, but it is a simplified example. In most instances, cost functions and hard deadlines are almost impossible to calculate in closed form and numerical methods must be used instead. In Section 2.4, we provide references to the literature on this subject.

2.2.5 Discussion

The correct choice of performance measures is vital to the ability to accurately and concisely represent the performance of any system. Performance measures are not just a means to convey to the user or the buyer the relative goodness of a computer for a particular application. If properly chosen, they can also provide efficient interfaces between the worlds of the user and the control engineer, and between the worlds of the control engineer and the computer engineer. Performability and cost functions are particularly attractive in this respect.

2.3 ESTIMATING PROGRAM RUN TIMES

Since real-time systems should meet deadlines, it is important to be able to accurately estimate program run times. Estimating the execution time of any given program is a very difficult task and is the focus of current research. It depends on the following factors:

Source code: Source code that is carefully tuned and optimized takes less time to execute.

Compiler: The compiler maps the source-level code into a machine-level program. This mapping is not unique; the actual mapping will depend on the actual implementation of the particular compiler that is being used. The execution time will depend on the nature of the mapping.

Machine architecture: Many aspects of the machine architecture have an effect on the execution time that is difficult to quantify exactly. Executing a program may require much interaction between the processor(s) and the memory and I/O devices. Such an interaction can take place over an interconnection network (e.g., a bus) that can be shared by other processors. The time spent waiting to

gain access to the network affects the execution time. The number of registers per processor affects how many variables can be held in the CPU. The greater the number of registers and the cleverer the compiler is in managing these registers, the fewer the number of accesses that need to go out of the CPU. This results in reducing the memory-access time, and hence the instruction-execution time. The size and organization of the cache (if any) will also affect the memory-access time, as will the clock rate. Also, many machines use dynamic RAM for their main memory. To keep the contents of these memories, we need to periodically *refresh* them; this is done by periodically reading the contents of each memory location and writing them back. The refresh task has priority over the CPU and thus affects the memory-access time.

Operating system: The operating system determines such issues as task scheduling and memory management, both of which have a major impact on the execution time. Along with the machine architecture, it determines the interrupt-handling overhead.

These factors can interact with one another in complex ways. In order to obtain good execution-time bounds, we need to account for these interactions precisely. Such accounting is very hard to do since it requires the analysis of every aspect of the execution of the program and of any interactions of that task with others.

Ideally, we would like to have a tool that would accept as inputs the compiler, the source code, and a description of the architecture, and then produce a good estimate of the execution time of the code. Unfortunately, such a tool does not exist. In this section, we look at partial solutions to the problem of execution-time estimation.

Some readers may wonder why an experimental approach to estimating the execution time is not followed, that is, actually running the program on the target architecture for a large number of input sets and gathering statistics on the run time. This is not feasible for two reasons. First, the actual number of potential input sets is very large and it would take too long to try them all. Second, individual programs are not run in isolation; their run times are affected by the rest of the workload (including interrupts). We therefore need a more analytical approach to estimating execution time.

We begin by considering what information can be extracted from the source code. This is followed by a discussion of how to make estimates for code running on pipelined architectures using a case study of a two-stage pipeline. Finally, we discuss how instruction and data caches affect the run time estimates.

2.3.1 Analysis of Source Code

Let us begin by considering a very simple stretch of code.

```
L1: a := b * c;
L2: b := d + e;
L3: d := a - f;
```


This is straight-line code. Once control is transferred to the first of these statements, execution will continue sequentially until the last statement has been completed. *Straight-line code* is thus a stretch of code with exactly one entry point and exactly one exit point. Let us consider how to make a timing estimate for this stretch of code.

The total execution time is given by

$$\sum_{i=1}^{3} T_{\text{exec}}(Li) \tag{2.1}$$

where $T_{\text{exec}}(Li)$ is the time needed to execute Li.

The time needed to execute L1 will depend on the code that the compiler generates for it. For example, L1 could be translated into the following sequence:

```
L1.1  Get the address of c
L1.2  Load c
L1.3  Get the address of b
L1.4  Load b
L1.5  Multiply
L1.6  Store into a
```

The time needed to execute these instructions will depend on the machine architecture. If the machine is very simple, does not use pipelining, and has only one I/O port to the memory, then the execution time is given by the sum $\sum_{i=1}^{6} T_{\text{exec}}(L1.i)$. This assumes that the machine is devoted during that time to just these instructions; for example, that there are no interrupts. However, there are two factors that could make this bound rather loose. First, we are implicitly assuming that the variables `b` and `c` are not already in the CPU registers and have to be fetched from the cache or the main memory. This overlooks the possibility that some preceding code might have already loaded these variables into the registers and that they are still there. Second, the bounds on the execution times of individual instructions could be loose because they are data-dependent. For example, the multiply operation does not take a deterministic time on most machines—the time it takes to multiply two numbers depends on the numbers themselves. If we do not know in advance what these numbers are, we can only write loose bounds on the multiplication time.

Suppose that we have the following loop.

```
L4.  while (P) do
L5.       Q1;
L6.       Q2;
L7.       Q3;
L8.  end while;
```

where `P` is a logical statement and `Q1`, `Q2`, and `Q3` are instructions. It is obvious that unless we know how many times this loop is going to be executed, we have no way of estimating the execution time. So, at the very least, we need upper and lower bounds on the number of loop iterations. These bounds may be derived

either from an analysis of `P`, `Q1`, `Q2`, and `Q3`, or from some other user-derived input. The difficulty of deriving such bounds is why `while` loops are forbidden in the Euclid real-time programming language.

Consider the following `if-then-else` construct.

```
L9. if B1 then
        S1;
    else if B2 then
        S2;
    else if B3 then
        S3;
    else
        S4;
    end if;
```

The execution time will depend on which of the conditions `B1`, `B2`, `B3` are true. In the case where `B1` is true, the execution time is

$$T(\mathtt{B1}) + T(\mathtt{S1}) + T(JMP) \qquad (2.2)$$

where $T(JMP)$ is the time it takes to jump to the end of the `if-then-else` construct.

In the case where `B1` is false but `B2` is true, the execution time is

$$T(\mathtt{B1}) + T(\mathtt{B2}) + T(\mathtt{S2}) + T(JMP) \qquad (2.3)$$

The equations for the other two cases are written similarly.

If $t_{\text{lower}}(i)$ and $t_{\text{upper}}(i)$ are the lower and upper bounds of the estimates of case i, then (since there are four cases in all) the lower and upper execution-time bounds for this construct are given by

$$\min_{i\in\{1,2,3,4\}} t_{\text{lower}}(i) \quad \text{and} \quad \max_{i\in\{1,2,3,4\}} t_{\text{upper}}(i) \qquad (2.4)$$

respectively.

This becomes considerably more complex if we allow interrupts. The execution-time bound is then a function of the rates at which the interrupts occur and on the bounds on the time taken to service each interrupt.

Figure 2.6 shows the schematic of an experimental estimation system developed at the University of Washington for programs written in C. The preprocessor produces compiled assembly-language code and marks off blocks of code to be analyzed. For example, it might mark off the code associated with a single source-level instruction or a straight-line stretch of code as a block. The *parser* analyzes the input source program. The *procedure timer* maintains a table of procedures and their execution times. The *loop bounds* module obtains bounds on the number of iterations for the various loops in the system. The *time schema* is independent of the system; it depends only on the language. It computes the execution times of each block using the execution time estimates computed by the code prediction module. The *code prediction* module does this by using the code generated by the preprocessor and using the architecture analyzer to include the influence of the architecture.

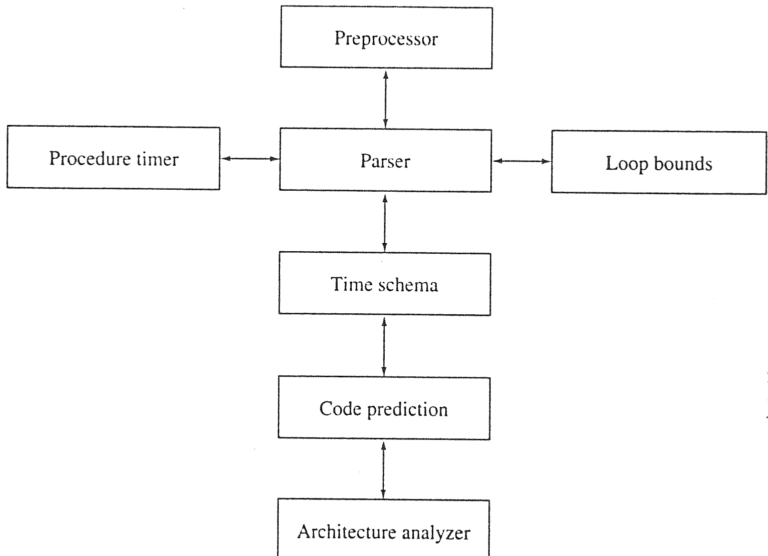

FIGURE 2.6
Schematic of a timing estimation system. (From C. Y. Park and A. Shaw, "Experiments with a Program Timing Tool Based on Source-Level Timing Schema," *IEEE Computer*, Vol. 24, No. 5, May 1991. © IEEE. Reprinted with permission.)

Take, for example, the `if-then-else` construct that we considered earlier in this section. The timing schema will come up with bounds such as Equation (2.4). The code prediction module will look at the assembly-level output of the preprocessor and obtain the parameters in Equation (2.4) by interacting with the architecture analyzer.

2.3.2 Accounting for Pipelining

The traditional von Neumann model of computers assumes sequential execution. An instruction is fetched, decoded, and executed. Only after that has been done does work begin on the next instruction. Under such conditions, computing the execution time of a straight-line stretch of code (in the absence of interrupts) consists, as we have seen above, of adding up the execution times of the individual instructions. The instruction execution-time is itself computed by simply adding up the times taken to fetch the instruction, decode it, fetch any operands that are required, execute the instruction, and, finally, carry out any store operations that are called for. The time taken for each of these steps can be obtained by examination of the machine architecture.

Most modern computers do not follow the von Neumann model exactly. They use *pipelining*, which involves the simultaneous handling of different parts of different instructions. To make programming easier, the machine still looks to the programmer as if it were following the von Neumann model.

Much of the complexity in dealing with pipelined machines arises from dependencies between instructions simultaneously active in the computer. For example, if instruction I_i requires the output of I_j, I_i must wait for I_j to produce that output before it can execute. Two other sources of complexity are conditional branches and interrupts.

The condition of a conditional branch has to be evaluated before the processor can tell whether the branch will be taken. Such an evaluation typically takes place in the execute stage. After the unconditional branch has been uncovered, but before the system knows whether or not it will be taken, the fetch unit has the following options.

- Stop prefetching instructions until the condition has been evaluated.
- Guess whether or not the branch will be taken and fetch instructions according to this guess.

The first alternative degrades performance, and so systems usually implement the second. In the worst case, there will be an incorrect guess. In such an event, the incorrectly prefetched instructions must be discarded and fetching must recommence from the correct point.

Interrupts are another source of complexity. Suppose an interrupt occurs and transfers control to an interrupt-handling routine. The worst-case execution time must then take into account the interrupt-handling time.

TWO-STAGE PIPELINE. In this section, we will show how to make timing estimates for a processor designed as a two-stage pipeline for some stretch, $I_1, I_2, \ldots, I_N$, of straight-line code. The first pipeline stage fetches instructions from the main memory and writes them into a prefetch buffer. The second stage handles everything else, including the operand read/write operations. Both the first and second stages will thus have occasion to access the memory, the first stage for fetching instructions and the second for loading/storing operands. They will thus have to coordinate their actions. We will assume that if the second stage needs to read one or more operands from main memory, there is a one-cycle delay in handshaking with the first stage. Similarly, if it needs to write some operands, there is a one-cycle handshaking delay. Also, the second stage will have nonpreemptive priority over the first stage for memory accesses; that is, if the second stage wishes to access the memory it will wait for any ongoing opcode fetches to finish before accessing the memory. A block diagram for the architecture is provided in Figure 2.7. In our discussion, we will assume that no cache memory is used, and that all the software and variables are memory-resident. Thus, there is no delay due to page faults. We will also ignore the effects of preemption or interrupts. Estimates of the delays caused by preemption and interrupts must be added to the estimates of the execution time. To further simplify our analysis, we will assume that if an instruction is executing that needs to access the memory (e.g., to load or store an operand), no instruction fetches will be begun during its execution. Also, the second (execute)

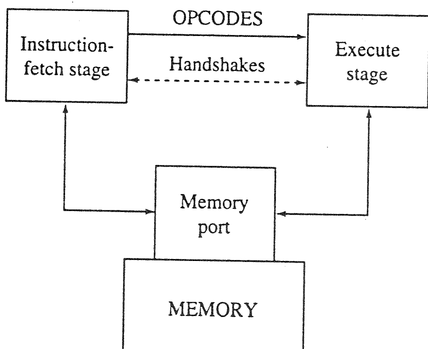

FIGURE 2.7
Two-stage pipeline.

stage is not itself pipelined; it can handle at most one instruction at any one time.

We begin with some notation.

b_i	Portion of the I_i execution not overlapped with the execution of any previous instruction (excluding handshake delays); note that in the execution time of an instruction we include the fetch time.
e_i	Second-stage execution time of I_i (i.e., excluding the instruction fetch time).
η_i	Execution time of I_i excluding memory accesses.
f_i	Number of bytes in the instruction buffer at the moment I_i completes execution.
g_i	Number of bytes of opcode fetched during the time I_i is in the execute stage of the pipeline, assuming the buffer is of infinite size.
h_i	Time spent in fetching the latest byte of the instruction-fetch operation if there is an instruction fetch ongoing at time τ_i (if no instruction fetch is in progress at τ_i, $h_i = 0$).
m	Number of CPU cycles for a memory access (read or write).
N_{buff}	Size of the instruction-fetch buffer in bytes.
ν_i	Size of instruction i opcode in bytes.
r_i	Number of data memory reads required by I_i.
t_i	Execution time of I_i not overlapped with execution of any previous instruction (including handshake delays).
τ_i	Instant at which I_i completes.
w_i	Number of data memory writes required by I_i.

An expression for t_i can be written by inspection as:

$$t_i = \begin{cases} b_i & \text{if } (r_i = 0) \wedge (w_i = 0) \\ b_i + 1 & \text{if } ((r_i \neq 0) \wedge (w_i = 0)) \vee ((r_i = 0) \wedge (w_i \neq 0)) \\ b_i + 2 & \text{if } (r_i > 0) \wedge (w_i > 0) \end{cases} \tag{2.5}$$

To compute b_i, we need an expression for e_i and we need to consider the effects of the instruction I_i fetch and other memory effects. e_i is easily derived:

$$e_i = \eta_i + m(r_i + w_i) \tag{2.6}$$

To complete the b_i derivation, we must consider the following special cases:

Case b1. $\nu_i > f_{i-1}$. $(\nu_i - f_{i-1})$ bytes of the I_i opcode still need to be fetched at time τ_{i-1}, when I_{i-1} finishes executing. This will take a further $m(\nu_i - f_{i-1}) - h_{i-1}$ time.

Case b2. $\nu_i \le f_{i-1}$. The entire I_i opcode has been fetched. There are two subcases:

Case b2.1. $(r_i + w_i = 0) \vee (h_{i-1} = 0)$. No time needs to be added for memory accesses.

Case b2.2. $(r_i + w_i > 0) \wedge (h_{i-1} > 0)$. I_i needs to read/write some operands. However, since $h_{i-1} > 0$, until $m - h_{i-1}$ cycles after I_i has started executing, the instruction-fetch unit is going to be accessing the memory. It is only after that time that any operand reads or writes can be started. In the worst case, this time must be added to the execution time.

We can thus write an expression for b_i as follows:

$$b_i = \begin{cases} e_i + m(\nu_i - f_{i-1}) - h_{i-1} & \text{if Case b1 applies} \\ e_i & \text{if Case b2.1 applies} \\ e_i + m - h_{i-1} & \text{if Case b2.2 applies} \end{cases} \tag{2.7}$$

In order to complete the derivation, we need expressions for f_i, and h_i. Let us start by writing an expression for the auxiliary variable, g_i, which is the number of bytes of opcode brought into the instruction buffer by the first stage when I_i is occupying the second stage of the pipeline. In making the computation for g_i, we disregard the possibility that the buffer is full and can take no more prefetches; this will be taken care of in a subsequent step. Once again, we consider a number of cases.

Case g1. $r_i + w_i = 0$. The execution of I_i will not interfere with any opcode fetches. There are the following subcases:

Case g1.1. $(\nu_i \le f_{i-1}) \wedge (h_{i-1} > 0) \wedge (e_i < m - h_{i-1})$. All the opcode of I_i has been fetched by τ_{i-1}, but there is not enough time for the ongoing opcode fetch to finish by the time I_i finishes execution. Therefore, $g_i = 0$.

Case g1.2. $(\nu_i \le f_{i-1}) \wedge (h_{i-1} > 0) \wedge (e_i \ge m - h_{i-1})$. All the opcode of I_i has been fetched by τ_{i-1}, and the opcode fetch that was ongoing when I_i started execution will have time to finish and will be followed by subsequent fetches. The number of these subsequent fetches is

$$\left\lfloor \frac{e_i - (m - h_{i-1})}{m} \right\rfloor$$

Therefore,

$$g_i = 1 + \left\lfloor \frac{e_i - (m - h_{i-1})}{m} \right\rfloor$$

Case g1.3. $(\nu_i > f_{i-1}) \vee (h_{i-1} = 0)$. Either some of the opcode of I_i has not been fetched by time τ_{i-1} or there is no ongoing opcode fetch at time τ_{i-1}. In either event, the number of bytes of opcode fetched during I_i execution is given by $g_i = \lfloor e_i/m \rfloor$.

Case g2. $r_i + w_i > 0$. I_i needs to access memory during its execution. Recall that the second stage of the pipeline has nonpreemptive priority over the first for memory accesses. There are the following subcases:

Case g2.1. $(\nu_i > f_{i-1}) \vee (h_{i-1} = 0)$. When the execution of I_i begins, there is no ongoing instruction fetch. Since $r_i + w_i > 0$, we will prevent the instruction-fetch unit from prefetching any instructions lest that interfere with the memory operations of the second stage as it executes I_i. Hence $g_i = 0$.

Case g2.2. $(\nu_i \le f_{i-1}) \wedge (h_{i-1} > 0) \wedge (e_i \ge m - h_{i-1})$. The ongoing instruction fetch at τ_{i-1} will complete, but we will prevent any further prefetches by the first stage for the reason mentioned in Case g2.1. Hence, $g_i = 1$.

We now introduce another auxiliary variable, s_i. At τ_{i-1}, there are f_{i-1} bytes in the instruction buffer. s_i is obtained by adding f_{i-1} and the bytes brought in during the interval $[\tau_{i-1}, \tau_i]$, assuming that the buffer is of infinite size. We can therefore write (the reader is invited to prove this in the exercises)

$$s_i = \begin{cases} f_{i-1} + g_i & \text{if } \nu_i \le f_{i-1} \\ \nu_i + g_i & \text{otherwise} \end{cases} \tag{2.8}$$

The equation for f_i is then:

$$f_i = \begin{cases} 0 & \text{if } i = 0 \\ \min\{s_i, N_{\text{buff}}\} - \nu_i & \text{if } i > 0 \end{cases} \tag{2.9}$$

We turn now to h_i. We will once again work through a number of cases.

Case h.1. $(r_i + w_i > 0)$. Recall that in such a case, we do not allow any new instruction fetches to start once any ongoing fetch at τ_i is done. No new instruction fetches are begun; hence $h_i = 0$.

Case h.2. $(r_i + w_i = 0)$. There are four subcases:

Case h.2.1. $(s_i \ge N_{\text{buff}})$. Since the buffer is full, no new instruction fetches can be started; hence, $h_i = 0$.

Case h.2.2. $(s_i < N_{\text{buff}}) \wedge ((h_{i-1} = 0) \vee (\nu_i > f_{i-1}))$. If $h_{i-1} = 0$, there is no ongoing instruction fetch at τ_{i-1}. If $(\nu_i > f_{i-1})$ also, there is no ongoing instruction fetch when I_i starts execution. This is because I_i begins execution the instant the last byte of its opcode is brought into the buffer. In both cases, g_i instruction fetches are completed during the execution of I_i. Hence, $h_i = e_i - mg_i$.

Case h.2.3. $(s_i < N_{\text{buff}}) \wedge (h_{i-1} > 0) \wedge (\nu_i \leq f_{i-1}) \wedge (e_i < m - h_{i-1})$. The ongoing instruction fetch at τ_{i-1} does not have time to complete before I_i completes. Hence, $h_i = e_i + h_{i-1}$.

Case h.2.4. $(s_i < N_{\text{buff}}) \wedge (h_{i-1} > 0) \wedge (\nu_i \leq f_{i-1}) \wedge (e_i \geq m - h_{i-1})$. It takes $m - h_{i-1}$ cycles to finish the instruction fetch that is ongoing at τ_{i-1}, and a further mg_i to finish the g_i fetches that complete during the execution of I_i. The time left over is thus $h_i = e_i - (m - h_{i-1}) - mg_i$.

We now have all the ingredients required to compute t_i, and hence an estimate of the execution time.

Example 2.9. Consider the following five-instruction stretch of straight-line code:

Instruction	η_i	ν_i	r_i	w_i
I_1	10	2	0	0
I_2	4	1	0	0
I_3	10	3	0	0
I_4	2	2	2	0
I_5	5	2	0	0

Let us assume that $m = 4$, (i.e., it takes four processor cycles to make a memory access). Since $\nu_1 = 2$, the fetch stage has to read two bytes from the main memory before I_1 can start executing. This takes a total of $2 \times 4 = 8$ cycles. Thus, I_1 starts executing at time 8, and completes at time $\tau_1 = 8+10 = 18$. Since I_1 has no operand reads or writes, there is no chance that there will be any operand accesses that are held up while the fetch unit is accessing the memory. The fetch unit starts fetching the opcodes for the following instructions at time 8. By time 12, it has brought in the opcode for I_2, and by time 24 the opcode for I_3. I_2 completes execution at 22. However, I_3 cannot start execution at this time because all of its opcode has not yet been fetched; this happens at time 24. I_3 executes over the interval [24, 34]. I_4 starts at 34. Since $r_4 > 0$, it needs to access the memory to read operands. This cannot happen until the fetch stage completes bringing in the first byte of the I_5 opcode, which is at time 36. Thus, $h_3 = 2$ and $t_4 = 2 + 2 + (2 \times 4) + 1 = 13$. I_4 thus occupies the execute stage over the interval [34, 47]. During [36, 47], the fetch unit is precluded from accessing the memory. Over [47, 51], the fetch unit brings in the second byte of the I_5 opcode and I_5 can begin at that time. Figure 2.8 summarizes this activity.

This study of a two-stage machine brings out the intricacy of any estimation of execution time. The machine under study is a particularly simple one, far simpler than many of the microprocessors currently on the market. There are only two stages to the processor pipeline and no more than one instruction can reside in the second stage at any one time. By contrast, modern processors have multiple instructions in the execute stage at the same time, and they can even complete out of program sequence. In our example, external interrupts are not accounted for, nor are problems associated with exceptions generated by the program (e.g., as a result of an arithmetic overflow).

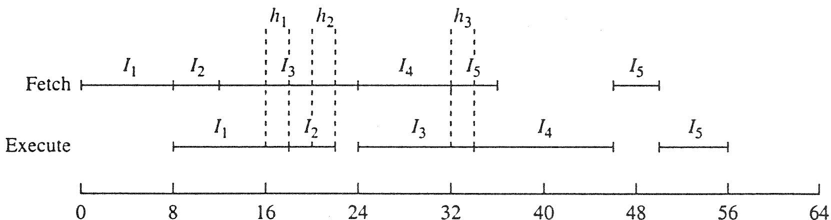

FIGURE 2.8
Two-stage example.

Only straight-line code is considered; no branches are taken into account. Despite these simplifications, the analysis is tedious and long. Essentially, one has to account for all possible cases that might affect the timing. This gets more difficult as the processor complexity increases and more cases must be considered.

2.3.3 Caches

Caches are meant to ameliorate the effects on execution of the wide disparity that exists between processor and main memory cycle times. However, they can also impair our ability to obtain good execution-time bounds. The time taken by an access depends on whether or not the word being accessed is in the cache. If it is, the access time is simply the cache access time; if it is not, it is the time to access the main memory, which is much larger.[5]

It is fairly difficult to predict whether a given access will result in a cache miss, since the cache contents are not easy to predict—they are a function of the size, organization, and replacement strategy of the cache, in addition to the sequence of accesses by the processor. Once a block is brought into the cache, it will remain there until it has to be removed to make room for another block. To determine accurately the presence or absence of a data block thus requires that we know the sequence of accesses. It is impossible in most cases to predict exactly what this sequence is going to be. Two of the main reasons are:

Conditional branches: These determine the actual execution path of the program. We don't know in advance which branches are going to be taken and we cannot explicitly follow through on every possible execution path, since these increase exponentially in number with the number of conditional branches.

Preemptions: When task A is preempted by task B, the blocks that were brought into the cache by task A may have to be removed to make room for B's accesses. As a result, when A resumes execution it will encounter a flurry

[5] We are assuming that everything is in main memory and that page faults do not occur. If they do, this will cause even longer delays.

```
A(i) = 0, i=1, ..., n. /*A(i) is the number of partitions
                          assigned to T_i so far.*/
while(number_of_partitions_left ≠ 0) do
    i_max = {i |δ(i, A(i)) is maximized}.
    A(i_max) = A(i_max) + 1
    number_of_partitions_left = number_of_partitions_left - 1
end while
end
```

FIGURE 2.9
Allocation of partitions to tasks.

of cache misses. This can be avoided by giving each task its own portion of the cache so that during its lifetime, each task "owns" its portion and no other task is allowed access to it.

Of course, one can always obtain a worst-case execution time by assuming that every access will result in a cache miss, but such an estimate is likely to be very inaccurate. How to make reasonably accurate estimates where caches are concerned is a matter for research.

The Strategic Memory Allocation for Real-Time (SMART) cache was developed to get around this difficulty. A SMART cache is broken down into exclusive partitions and a shared area. When a critical task starts executing, it is assigned one or more exclusive partitions. It restricts its cache accesses to its partitions and to the shared area. Until that task has completed or aborted, it has exclusive rights over its assigned partitions. Even if it is preempted, no other task can overwrite the contents of its partitions.[6] Such a policy prevents a flurry of misses upon task resumption and permits more accurate performance estimates.

How many partitions should be assigned to each task? This is a discrete optimization problem; like most such problems it is NP-hard and requires suboptimal heuristics. One heuristic is as follows. Let $\epsilon_i(k)$ be the run time of task T_i if k partitions have been assigned to it. We will assume that we have some means of estimating $\epsilon_i(k)$. If f_i is the average frequency with which T_i is executed (f_i is the inverse of the period for periodic tasks and is the inverse of the average interinvocation interval for aperiodic tasks), define the weight $w_i(k) = f_i\epsilon(k)$. Define $\delta(i,k) = w_i(k) - w_i(k+1)$. Given a set of tasks $T_1, \ldots, T_n$ assign the partitions one by one using the gradient descent algorithm in Figure 2.9. This is a greedy algorithm that allocates the partitions one by one.

[6] The shared area contents could be overwritten unless they are locked in the cache.

2.3.4 Virtual Memory

Virtual memory is a major source of execution-time uncertainty. That is why it is wise to avoid using virtual memory whenever possible. The time taken to handle page faults, for example, can very widely and obtaining a good bound on the page-fault rate is almost impossible.

2.4 SUGGESTIONS FOR FURTHER READING

A brief survey of performance measures for computer controllers, with many references to the technical literature, is provided in [10]. Capacity reliability is described in [6], and computation reliability and computation threshold in [2]. Performability is covered in [17], and a detailed example is worked out in [18]. Example 2.7 is based on [17]. Furchtgott [5] is a particularly detailed exposition of performability. Cost functions are covered in [8, 9], which contain some further examples of their calculation. See [20] for a derivation of hard deadlines and cost functions relating to an aircraft in the process of landing. A procedure for the computation of hard deadlines for linear time-invariant control systems is described in [18]. There is a vast literature on life-cycle costing; see, for example, [1].

Not much has been published on estimating task run times. Our discussion in Section 2.3.1 is based largely on [15, 17], and that in Section 2.3.2 on [21]. Some other interesting work related to Section 2.3.1 can be found in [14, 16]. Work on studying the impact of pipelined architectures on program run times is described in [3, 7, 11]. The effect of caches is also considered in [11].

EXERCISES

2.1. Select accomplishment levels to describe the performability of a controller for
  (a) a motor car
  (b) traffic lights

2.2. Define accomplishment levels for a nongracefully degrading system (e.g., a lightbulb) so that performability reduces to traditional reliability.

2.3. Suppose you are designing a collision-avoidance system for a car. The system monitors the distance between the car and the one in front of it, and slows the car down whenever it is too close. Write accomplishment levels for such a system.

2.4. There are two ways of implementing a time limit specified for executing a loop. The first is for the compiler to set a maximum number of iterations that may be carried out. The second is to maintain during execution a timer that determines if allowing another iteration would cause the limit to be exceeded. Discuss the advantages and disadvantages of each approach.

2.5. Pick some two-stage pipelined microprocessor for which you have access to the hardware manual. Write a twenty-instruction stretch of straight-line code and compute an estimate of how long it will take to execute.

2.6. Prove Equation (2.8).

2.7. Suppose you have a two-stage pipelined system with two memory modules. Instructions are stored in one memory module, and operands in the other. As a result,