Chapter 9

Measuring Program Execution Time

One common question people ask is "How fast does Program X run on Machine Y?" Such a question might be raised by a programmer trying to optimize program performance, or by a customer trying to decide which machine to buy. In our earlier discussion of performance optimization (Chapter 5), we assumed this question could be answered with perfect accuracy. We were trying to establish the cycles per element (CPE) measure for programs down to two decimal places. This requires an accuracy of 0.1% for a procedure having a CPE of 10. In this chapter, we address this problem and discover that it is surprisingly complex.

You might expect that making near-perfect timing measurements on a computer system would be straightforward. After all, for a particular combination of program and data, the machine will execute a fixed sequence of instructions. Instruction execution is controlled by a processor clock that is regulated by a precision oscillator. There are many factors, however, that can vary from one execution of a program to another. Computers do not simply execute one program at a time. They continually switch from one process to another, executing some code on behalf of one process before moving on to the next. The exact scheduling of processor resources for one program depends on such factors as the number of users sharing the system, the network traffic, and the timing of disk operations. The access patterns to the caches depend not just on the references made by the program we are trying to measure, but on those of other processes executing concurrently. Finally, the branch prediction logic tries to guess whether branches will be taken or not based on past history. This history can vary from one execution of a program to another.

In this chapter, we describe two basic mechanisms computers use to record the passage of time—one based on a low frequency timer that periodically interrupts the processor and one based on a counter that is incremented every clock cycle. Application programmers can gain access to the first timing mechanism by calling library functions. Cycle timers can be accessed by library functions on some systems, but they require writing assembly code on others. We have deferred the discussion of program timing until now, because it requires understanding aspects of both the CPU hardware and the way the operating system manages process execution.

Using the two timing mechanisms, we investigate methods to get reliable measurements of program performance. We will see that timing variations due to context switching tend to be very large and hence must be eliminated. Variations caused by other factors such as cache and branch prediction are generally managed by evaluating program operation under carefully controlled conditions. Generally, we can get accurate measurements for durations that are either very short (less than around 10 millisecond) or very long (greater than

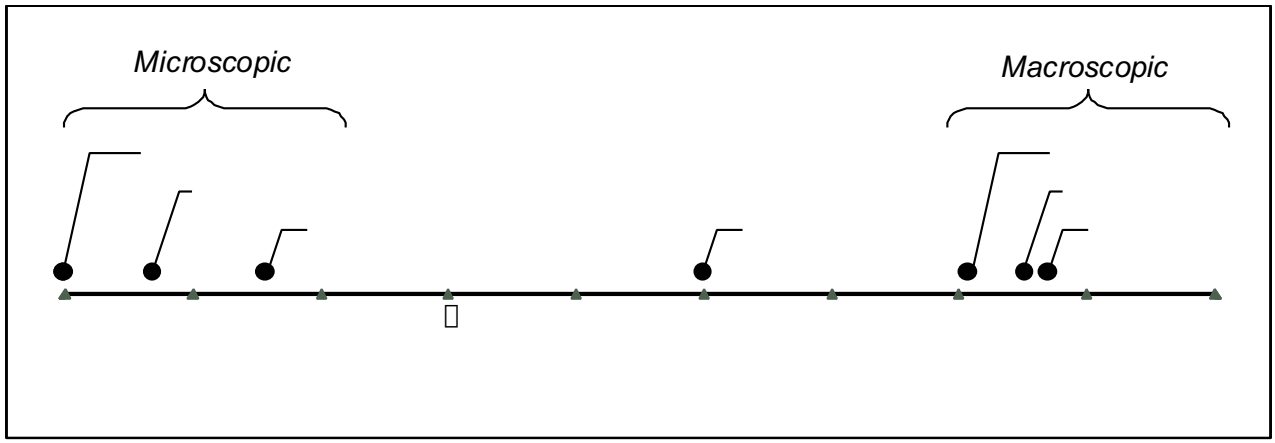

Figure 9.1: **Time Scale of Computer System Events.** The processor hardware works at a microscopic a time scale in which events having durations on the order of a few nanoseconds (ns). The OS must deal on a macroscopic time scale with events having durations on the order of a few milliseconds (ms).

around 1 second), even on heavily loaded machines. Times between around 10 milliseconds and 1 second require special care to measure accurately.

Much of the understanding of performance measurement is part of the folklore of computer systems. Different groups and individuals have developed their own techniques for measuring program performance, but there is no widely available body of literature on the subject. Companies and research groups concerned with getting highly accurate performance measurements often set up specially configured machines that minimize any sources of timing irregularity, such as by limiting access and by disabling many OS and networking services. We want methods that application programmers can use on ordinary machines, but there are no widely available tools for this. Instead, we will develop our own.

In this presentation we work through the issues systematically. We describe the design and evaluation of a number of experiments that helped us arrive at methods to achieve accurate measurements on a small set of systems. It is unusual to find a detailed experimental study in a book at this level. Generally, people expect the final answers, not a description of how those answers were determined. In this case, however, we cannot provide definitive answers on how to measure program execution time for an arbitrary program on an arbitrary system. There are too many variations of timing mechanisms, operating system behaviors, and runtime environment to have one single, simple solution. Instead, we anticipate that you will need to run your own experiments and develop your own performance measurement code. We hope that our case study will help you in this task. We summarize our findings in the form of a protocol that can guide your experiments.

9.1 The Flow of Time on a Computer System

Computers operate on two fundamentally different time scales. At a microscopic level, they execute instructions at a rate of one or more per clock cycle, where each clock cycle requires only around one nanosecond (abbreviated "ns"), or 10^{-9} seconds. On a macroscopic scale, the processor must respond to external events

that occur on time scales measured in milliseconds (abbreviated "ms"), or 10^{-3} seconds. For example, during video playback, the graphics display for most computers must be refreshed every 33 ms. A world-record typist can only type keystrokes at a rate of around one every 50 milliseconds. Disks typically require around 10 ms to initiate a disk transfer. The processor continually switches between these many tasks on a macroscopic time scale, devoting around 5 to 20 milliseconds to each task at a time. At this rate, the user perceives the tasks as being performed simultaneously, since a human cannot discern time durations shorter than around 100 ms. Within that time the processor can execute millions of instructions.

Figure 9.1 plots the durations of different event types on a logarithmic scale, with microscopic events having durations measured in nanoseconds and macroscopic events having durations measured in milliseconds. The macroscopic events are managed by OS routines that require around 5,000 to 200,000 clock cycles. These time ranges are measured in microseconds (abbreviated μs, where μ is the Greek letter "mu"). Although that may sound like a lot of computation, it is so much faster than the macroscopic events being processed that these routines place only a small load on the processor.

Practice Problem 9.1:

When a user is editing files with a real-time editor such as EMACS, every keystroke generates an interrupt signal. The operating system must then schedule the editor process to take the appropriate action for this keystroke. Suppose we had a system with a 1 GHz clock, and we had 100 users running EMACS typing at a rate of 100 words per minute. Assume an average of 6 characters per word. Assume also that the OS routine handling keystrokes requires, on average, 100,000 clock cycles per keystroke. What fraction of the processor load is consumed by all of the keystroke processing?

Note that this is a very pessimistic analysis of the load induced by keyboard usage. It's hard to imagine a real-life scenario with so many users typing this fast.

9.1.1 Process Scheduling and Timer Interrupts

External events such as keystrokes, disk operations, and network activity generate interrupt signals that make the operating system scheduler take over and possibly switch to a different process. Even in the absence of such events, we want the processor to switch from one process to another so that it will appear to the users as if the processor is executing many programs simultaneously. For this reason, computers have an external timer that periodically generates an interrupt signal to the processor. The spacing between these interrupt signals is called the *interval time*. When a timer interrupt occurs, the operating system scheduler can choose to either resume the currently executing process or to switch to a different process. This interval must be set short enough to ensure that the processor will switch between tasks often enough to provide the illusion of performing many tasks simultaneously. On the other hand, switching from one process to another requires thousands of clock cycles to save the state of the current process and to set up the state for the next, and hence setting the interval too short would cause poor performance. Typical timer intervals range between 1 and 10 milliseconds, depending on the processor and how it is configured.

Figure 9.2(a) illustrates the system's perspective of a hypothetical 150 ms of operation on a system with a 10 ms timer interval. During this period there are two active processes: A and B. The processor alternately executes part of process A, then part of B, and so on. As it executes these processes, it operates either in *user mode*, executing the instructions of the application program; or in *kernel mode*, performing operating system functions on behalf of the program, such as, handling page faults, input, or output. Recall that kernel

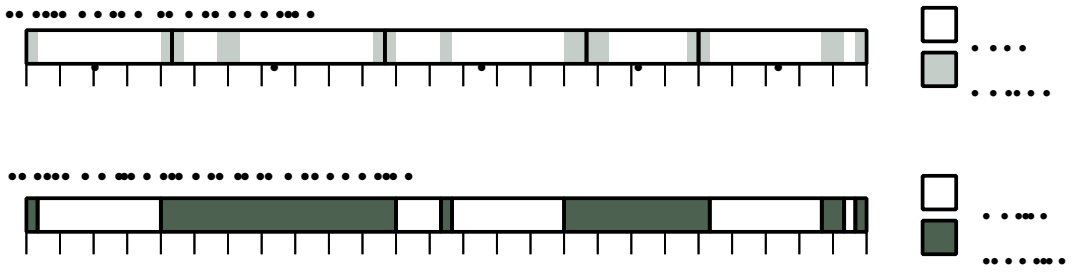

Figure 9.2: **System's vs. Applications View of Time.** The system switches from process to process, operating in either user or kernel mode. The application only gets useful computation done when its process is executing in user mode.

operation is considered part of each regular process rather than a separate process. The operating system scheduler is invoked every time there is an external event or a timer interrupt. The occurrences of timer interrupts are indicated by the tick marks in the figure. This means that there is actually some amount of kernel activity at every tick mark, but for simplicity we do not show it in the figure.

When the scheduler switches from process A to process B, it must enter kernel mode to save the state of process A (still considered part of process A) and to restore the state of process B (considered part of process B). Thus, there is kernel activity during each transition from one process to another. At other times, kernel activity can occur without switching processes, such as when a page fault can be satisfied by using a page that is already in memory.

9.1.2 Time from an Application Program's Perspective

From the perspective of an application program, the flow of time can be viewed as alternating between periods when the program is *active* (executing its instructions), and *inactive* (waiting to be scheduled by the operating system). It only performs useful computation when its process is operating in user mode. Figure 9.2(b) illustrates how program A would view the flow of time. It is active during the light-colored regions, when process A is executing in user mode; otherwise it is inactive.

As a way to quantify the alternations between active and inactive time periods, we wrote a program that continuously monitors itself and determines when there have been long periods of inactivity. It then generates a *trace* showing the alternations between periods of activity and inactivity. Details of this program are described later in the chapter. An example of such a trace is shown in Figure 9.3, generated while running on a Linux machine with a clock rate of around 550 MHz. Each period is labeled as either active ("A") or inactive "I"). The periods are numbered 0 to 9 for identification. For each period, the start time (relative to the beginning of the trace) and the duration are indicated. Times are expressed in both clock cycles and milliseconds. This trace shows a total of 20 time periods (10 active and 10 inactive) having a total duration of 66.9 ms. In this example, the periods of inactivity are fairly short, with the longest being 0.50 ms. Most of these periods of inactivity were caused by timer interrupts. The process was active for around 95.1% of the total time monitored. Figure 9.4 shows a graphical rendition of the trace shown in Figure 9.3. Observe the regular spacing of the boundaries between the activity periods indicated by the gray triangles. These

```
A0    Time        0    (0.00 ms),   Duration  3726508    (6.776448 ms)
I0    Time  3726508    (6.78 ms),   Duration   275025    (0.500118 ms)
A1    Time  4001533    (7.28 ms),   Duration        0    (0.000000 ms)
I1    Time  4001533    (7.28 ms),   Duration     7598    (0.013817 ms)
A2    Time  4009131    (7.29 ms),   Duration  5189247    (9.436358 ms)
I2    Time  9198378   (16.73 ms),   Duration   251609    (0.457537 ms)
A3    Time  9449987   (17.18 ms),   Duration  2250102    (4.091686 ms)
I3    Time 11700089   (21.28 ms),   Duration    14116    (0.025669 ms)
A4    Time 11714205   (21.30 ms),   Duration  2955974    (5.375275 ms)
I4    Time 14670179   (26.68 ms),   Duration   248500    (0.451883 ms)
A5    Time 14918679   (27.13 ms),   Duration  5223342    (9.498358 ms)
I5    Time 20142021   (36.63 ms),   Duration   247113    (0.449361 ms)
A6    Time 20389134   (37.08 ms),   Duration  5224777    (9.500967 ms)
I6    Time 25613911   (46.58 ms),   Duration   254340    (0.462503 ms)
A7    Time 25868251   (47.04 ms),   Duration  3678102    (6.688425 ms)
I7    Time 29546353   (53.73 ms),   Duration     8139    (0.014800 ms)
A8    Time 29554492   (53.74 ms),   Duration  1531187    (2.784379 ms)
I8    Time 31085679   (56.53 ms),   Duration   248360    (0.451629 ms)
A9    Time 31334039   (56.98 ms),   Duration  5223581    (9.498792 ms)
I9    Time 36557620   (66.48 ms),   Duration   247395    (0.449874 ms)
```

Figure 9.3: **Example Trace Showing Activity Periods.** From the perspective of an application program, processor operation alternates between periods when the program is actively executing (italicized) and when it is inactive. This trace shows a log of these periods for a program over a total duration of 66.9 ms. The program was active for 95.1% of this time.

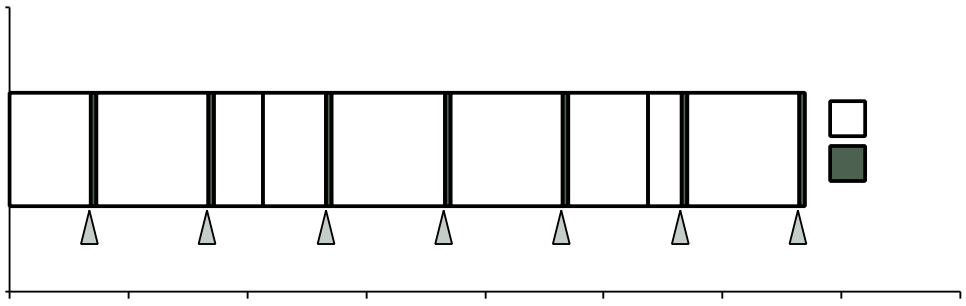

Figure 9.4: **Graphical Representation of Trace in Figure 9.3.** Timer interrupts are indicated with gray triangles.

```
A48   Time 191514104   (349.40 ms),   Duration  5224961   (9.532449 ms)
I48   Time 196739065   (358.93 ms),   Duration   247557   (0.451644 ms)
A49   Time 196986622   (359.38 ms),   Duration   858571   (1.566382 ms)
I49   Time 197845193   (360.95 ms),   Duration     8297   (0.015137 ms)
A50   Time 197853490   (360.97 ms),   Duration  4357437   (7.949733 ms)
I50   Time 202210927   (368.91 ms),   Duration  5718758  (10.433335 ms)
A51   Time 207929685   (379.35 ms),   Duration  2047118   (3.734774 ms)
I51   Time 209976803   (383.08 ms),   Duration     7153   (0.013050 ms)
A52   Time 209983956   (383.10 ms),   Duration  3170650   (5.784552 ms)
I52   Time 213154606   (388.88 ms),   Duration  5726129  (10.446783 ms)
A53   Time 218880735   (399.33 ms),   Duration  5217543   (9.518916 ms)
I53   Time 224098278   (408.85 ms),   Duration  5718135  (10.432199 ms)
A54   Time 229816413   (419.28 ms),   Duration  2359281   (4.304286 ms)
I54   Time 232175694   (423.58 ms),   Duration     7096   (0.012946 ms)
A55   Time 232182790   (423.60 ms),   Duration  2859227   (5.216390 ms)
I55   Time 235042017   (428.81 ms),   Duration  5718793  (10.433399 ms)
```

Figure 9.5: **Example Trace Showing Activity Periods on Loaded Machine.** When other active processes are present, the tracing process is inactive for longer periods of time. This trace shows a log of these periods for a program over a total duration of 89.8 ms. The process was active for 53.0% of this time.

boundaries are caused by timer interrupts.

Figure 9.5 shows a portion of a trace when there is one other active process sharing the processor. The graphical rendition of this trace is shown in Figure 9.6. Note that the time scales do not line up, since the portion of the trace we show in Figure 9.5 started at 349.4 ms into the tracing process. In this example we can see that while handling some of the timer interrupts, the OS also decides to switch context from one process to another. As a result, each process is only active around 50% of the time.

Practice Problem 9.2:

This problem concerns the interpretation of the section of the trace shown in Figure 9.5.

A. At what times during this portion of the trace did timer interrupts occur? (Some of these time points can be extracted directly from the trace, while others must be estimated by interpolation.)

B. Which of these occurred while the tracing process was active, and which while it was inactive?

C. Why are the longest periods of inactivity longer than the longest periods of activity?

D. Based on the pattern of active and inactive periods shown in this trace, what percent of the time would you expect the tracing process to be inactive when averaged over a longer time scale?

9.2 Measuring Time by Interval Counting

The operating system also uses the timer to record the cumulative time used by each process. This information provides a somewhat imprecise measure of program execution time. Figure 9.7 provides a graphic illustration of how this accounting works for the example of system operation shown in Figure 9.2. In this discussion, we refer to the period during which just one process executes as a *time segment*.

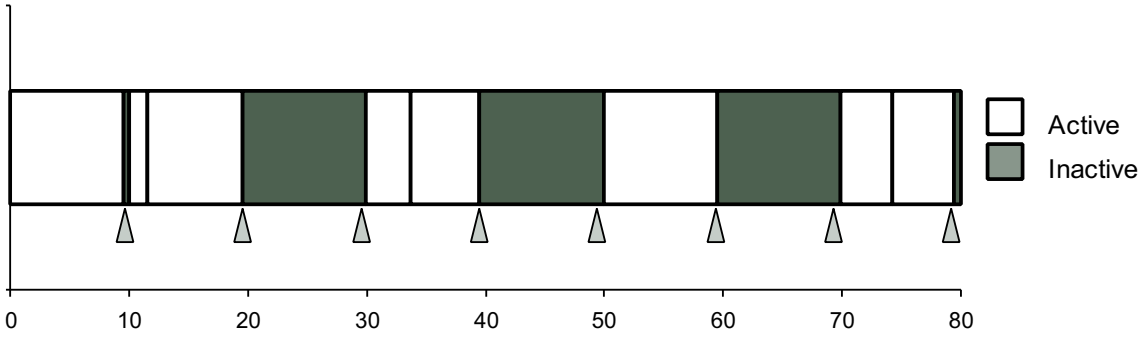

Figure 9.6: **Graphical Representation of Activity Periods for Trace in Figure 9.5.** Timer interrupts are indicated by gray triangles

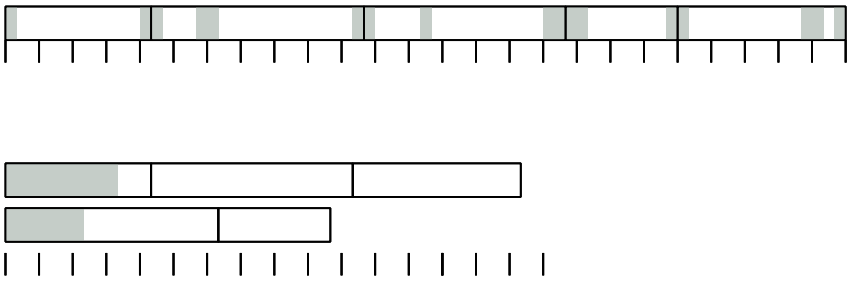

Figure 9.7: **Process Timing by Interval Counting.** With a timer interval of 10 ms, every 10 ms segment is assigned to a process as part of either its user (u) or system (s) time. This accounting provides only an approximate measure of program execution time.

9.2.1 Operation

The operating system maintains counts of the amount of user time and the amount of system time used by each process. When a timer interrupt occurs, the operating system determines which process was active and increments one of the counts for that process by the timer interval. It increments the system time if the system was executing in kernel mode, and the user time otherwise. The example shown in Figure 9.7(a) indicates this accounting for the two processes. The tick marks indicate the occurrences of timer interrupts. Each is labeled by the count that gets incremented: either `Au` or `As` for process A's user or system time, or `Bu` or `Bs` for process B's user or system time. Each tick mark is labeled according to the activity to its immediate left. The final accounting shows that process A used a total of 150 milliseconds: 110 of user time and 40 of system time. It shows that B used a total of 100 milliseconds: 70 of user time and 30 of system time.

9.2.2 Reading the Process Timers

When executing a command from the Unix shell, the user can prefix the command with the word "`time`" to measure the execution time of the command. This command uses the values computed using the accounting scheme described above. For example, to time the execution time of program `prog` with command line arguments `-n 17`, the user can simply type the command:

```
unix> time prog -n 17
```

After the program has executed, the shell will print a line summarizing the run time statistics, for example,

```
2.230u 0.260s 0:06.52 38.1% 0+0k 0+0io 80pf+0w
```

The first three numbers shown in this line are times. The first two show the seconds of user and system time. Observe how both of these show a 0 in the third decimal place. With a timer interval of 10 ms, all timings are multiples of hundredths of seconds. The third number is the total elapsed time, given in minutes and seconds. Observe that the system and user time sum to 2.49 seconds, less than half of the elapsed time of 6.52 seconds, indicating that the processor was executing other processes at the same time. The percentage indicates what fraction the combined user and system times were of the elapsed time, e.g., $(2.23 + 0.26)/6.52 = 0.381$. The remaining statistics summarize the paging and I/O behavior.

Programmers can also read the process timers by calling the library function `times`, declared as follows:

```
#include <sys/times.h>

struct tms {
   clock_t tms_utime; /* user time */
   clock_t tms_stime; /* system time */
   clock_t tms_cutime; /* user time of reaped children */
   clock_t tms_cstime; /* system time of reaped children */
};

clock_t times(struct tms *buf);
```

Returns: number of clock ticks elapsed since system started

These time measurements are expressed in terms of a unit called *clock ticks*. The defined constant `CLK_TCK` specifies the number of clock ticks per second. The data type `clock_t` is typically defined to be a long integer. The fields indicating child times give the accumulated times used by children that have terminated and have been reaped. Thus, `times` cannot be used to monitor the time used by any ongoing children. As a return value, `times` returns the total number of clock ticks that have elapsed since the system was started. We can therefore compute the total time (in clock ticks) between two different points in a program execution by making two calls to `times` and computing the difference of the return values.

The ANSI C standard also defines a function `clock` that measures the total time used by the current process:

```
#include <time.h>

clock_t clock(void);
```

Returns: total time used by process

Although the return value is declared to be the same type `clock_t` used with the `times` function, the two functions do not, in general, express time in the same units. To scale the time reported by `clock` to seconds, it should be divided by the defined constant `CLOCKS_PER_SEC`. This value need not be the same as the constant `CLK_TCK`.

9.2.3 Accuracy of Process Timers

As the example illustrated in Figure 9.7 shows, this timing mechanism is only approximate. Figure 9.7(b) shows the actual times used by the two processes. Process A executed for a total of 153.3 ms, with 120.0 in user mode and 33.3 in kernel mode. Process B executed for a total of 96.7 ms, with 73.3 in user mode and 23.3 in kernel mode. The interval accounting scheme makes no attempt to resolve time more finely than the timer interval.

Practice Problem 9.3:

What would the operating system report as the user and system times for the execution sequence illustrated below. Assume a 10 ms timer interval.

Practice Problem 9.4:

On a system with a timer interval of 10 ms, some segment of process A is recorded as requiring 70 ms, combining both system and user time. What are the minimum and maximum actual times used by this segment?

Practice Problem 9.5:

What would the counters record as the system and user times for the trace shown in Figure 9.3? How does this compare to the actual time during which the process was active?

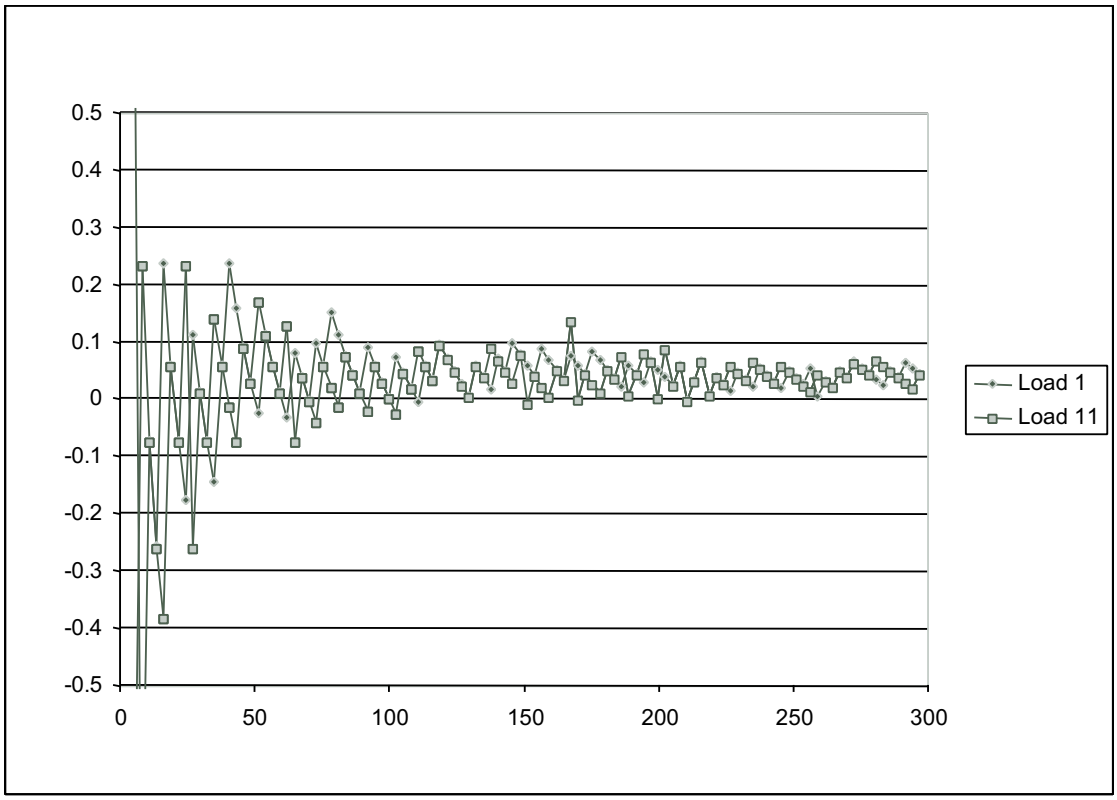

Figure 9.8: **Experimental Results for Measuring Interval Counting Accuracy.** The error is unacceptably high when measuring activities less than around 100 ms (10 timer intervals). Beyond this, the error rate is generally less than 10% regardless of whether running on lightly loaded (Load 1) or heavily loaded (Load 11) machine.

For programs that run long enough, (at least several seconds), the inaccuracies in this scheme tend to compensate for each other. The execution times of some segments are underestimated while those of others are overestimated. Averaged over a number of segments, the expected error approaches zero. From a theoretical perspective, however, there is no guaranteed bound on how far these measurements vary from the true run times.

To test the accuracy of this timing method, we ran a series of experiments that compared the time T_m measured by the operating system for a sample computation versus our estimate of what the time T_c would be if the system resources were dedicated solely to performing this computation. In general, T_c will differ from T_m for several reasons:

1. The inherent inaccuracies of the interval counting scheme can cause T_m to be either less or greater than T_c.

2. The kernel activity caused by the timer interrupt consumes 4 to 5% of the total CPU cycles, but these cycles are not accounted for properly. As can be seen in the trace illustrated in Figure 9.4, this activity finishes before the next timer interrupt and hence does not get counted explicitly. Instead, it simply

reduces the number of cycles available for the process executing during the next time interval. This will tend to increase T_m relative to T_c.

3. When the processor switches from one task to another, the cache tends to perform poorly for a transient period until the instructions and data for the new task get loaded into the cache. Thus the processor does not run as efficiently when switching between our program and other activities as it would if it executed our program continuously. This factor will tend to increase T_m relative to T_c.

We discuss how we can determine the value of T_c for our sample computation later in this chapter.

Figure 9.8 shows the results of this experiment running under two different loading conditions. The graphs show our measurements of the error rate, defined as the value of $(T_m - T_c)/T_c$ as a function of T_c. This error measure is negative when T_m underestimates T_c and is positive when T_m overestimates T_c. The two series show measurements taken under two different loading conditions. The series labeled "Load 1" shows the case where the process performing the sample computation is the only active process. The series labeled "Load 11" shows the case where 10 other processes are also attempting the same computation. The latter represents a very heavy load condition; the system is noticeably slow responding to keystrokes and other service requests. Observe the wide range of error values shown on this graph. In general, only measurements that are within $\pm 10\%$ of the true value are acceptable, and hence we want only errors ranging from around -0.1 to $+0.1$.

Below around 100 ms (10 timer intervals), the measurements are not at all accurate due to the coarseness of the timing method. Interval counting is only useful for measuring relatively long computations—100,000,000 clock cycles or more. Beyond this, we see that the error generally ranges between 0.0 and 0.1, that is, up to 10% error. There is no noticeable difference between the two different loading conditions. Notice also that the errors have a positive bias; the average error for all measurements with $T_m \geq 100$ms is 1.04, due to the fact that the timer interrupts are consuming around 4% of the CPU time.

These experiments show that the process timers are useful only for getting approximate values of program performance. They are too coarse-grained to use for any measurement having duration of less than 100 ms. On this machine they have a systematic bias, overestimating computation times by an average of around 4%. The main virtue of this timing mechanism is that its accuracy does not depend strongly on the system load.

9.3 Cycle Counters

To provide greater precision for timing measurements, many processors also contain a timer that operates at the clock cycle level. This timer is a special register that gets incremented every single clock cycle. Special machine instructions can be used to read the value of the counter. Not all processors have such counters, and those that do vary in the implementation details. As a result, there is no uniform, platform-independent interface by which programmers can make use of these counters. On the other hand, with just a small amount of assembly code, it is generally easy to create a program interface for any specific machine.

9.3.1 IA32 Cycle Counters

All of the timings we have reported so far were measured using the IA32 cycle counter. With the IA32 architecture, cycle counters were introduced in conjunction with the "P6" microarchitecture (the PentiumPro and its successors). The cycle counter is a 64-bit, unsigned number. For a processor operating with a 1 GHz clock, this counter will wrap around from $2^{64} - 1$ to 0 only once every 1.8×10^{10} seconds, or every 570 years. On the other hand, if we consider only the low order 32 bits of this counter as an unsigned integer, this value will wrap around every 4.3 seconds. One can therefore understand why the IA32 designers decided to implement a 64-bit counter.

The IA32 counter is accessed with the `rdtsc` (for "read time stamp counter") instruction. This instruction takes no arguments. It sets register `%edx` to the high-order 32 bits of the counter and register `%eax` to the low-order 32 bits. To provide a C program interface, we would like to encapsulate this instruction within a procedure:

```
void access_counter(unsigned *hi, unsigned *lo);
```

This procedure should set location `hi` to the high-order 32 bits of the counter and `lo` to the low-order 32 bits. Implementing `access_counter` is a simple exercise in using the embedded assembly feature of GCC, as described in Section 3.15. The code is shown in Figure 9.9.

Based on this routine, we can now implement a pair of functions that can be used to measure the total number of cycles that elapse between any two time points:

```
#include "clock.h"

void start_counter();
double get_counter();
```

Returns: number of cycles since last call to `start_counter`

We return the time as a `double` to avoid the possible overflow problems of using just a 32-bit integer. The code for these two routines is also shown in Figure 9.9. It builds on our understanding of unsigned arithmetic to perform the double-precision subtraction and to convert the result to a `double`.

9.4 Measuring Program Execution Time with Cycle Counters

Cycle counters provide a very precise tool for measuring the time that elapses between two different points in the execution of a program. Typically, however, we are interested in measuring the time required to execute some particular piece of code. Our cycle counter routines compute the total number of cycles between a call to `start_counter` and a call to `get_counter`. They do not keep track of which process uses those cycles or whether the processor is operating in kernel or user mode. We must be careful when using such a measuring device to determine execution time. We investigate some of these difficulties and how they can be overcome.

As an example of code that uses the cycle counter, the routine in Figure 9.10 provides a way to determine the clock rate of a processor. Testing this function on several systems with parameter `sleeptime` equal

code/perf/clock.c

```
1 /* Initialize the cycle counter */
2 static unsigned cyc_hi = 0;
3 static unsigned cyc_lo = 0;
4
5
6 /* Set *hi and *lo to the high and low order bits  of the cycle counter.
7    Implementation requires assembly code to use the rdtsc instruction. */
8 void access_counter(unsigned *hi, unsigned *lo)
9 {
10     asm("rdtsc; movl %%edx,%0; movl %%eax,%1"   /* Read cycle counter */
11         : "=r" (*hi), "=r" (*lo)                /* and move results to */
12         : /* No input */                        /* the two outputs */
13         : "%edx", "%eax");
14 }
15
16 /* Record the current value of the cycle counter. */
17 void start_counter()
18 {
19     access_counter(&cyc_hi, &cyc_lo);
20 }
21
22 /* Return the number of cycles since the last call to start_counter. */
23 double get_counter()
24 {
25     unsigned ncyc_hi, ncyc_lo;
26     unsigned hi, lo, borrow;
27     double result;
28
29     /* Get cycle counter */
30     access_counter(&ncyc_hi, &ncyc_lo);
31
32     /* Do double precision subtraction */
33     lo = ncyc_lo - cyc_lo;
34     borrow = lo > ncyc_lo;
35     hi = ncyc_hi - cyc_hi - borrow;
36     result = (double) hi * (1 << 30) * 4 + lo;
37     if (result < 0) {
38         fprintf(stderr, "Error: counter returns neg value: %.0f\n", result);
39     }
40     return result;
41 }
```

code/perf/clock.c

Figure 9.9: **Code Implementing Program Interface to IA32 Cycle Counter** Assembly code is required to make use of the counter reading instruction.

code/perf/clock.c

```
1 /* Estimate the clock rate by measuring the cycles that elapse */
2 /* while sleeping for sleeptime seconds */
3 double mhz(int verbose, int sleeptime)
4 {
5     double rate;
6
7     start_counter();
8     sleep(sleeptime);
9     rate = get_counter() / (1e6*sleeptime);
10    if (verbose)
11        printf("Processor clock rate ~= %.1f MHz\n", rate);
12    return rate;
13 }
```

code/perf/clock.c

Figure 9.10: `mhz`**: Determines the clock rate of a processor.**

to 1 shows that it reports a clock rate within 1.0% of the rated performance for the processor. This example clearly shows that our routines measure elapsed time rather than the time used by a particular process. When our program calls `sleep`, the operating system will not resume the process until the sleep time of one second has expired. The cycles that elapse during that time are spent executing other processes.

9.4.1 The Effects of Context Switching

A naive way to measuring the run time of some procedure `P` is to simply use the cycle counter to time one execution of `P`, as in the following code:

```
1 double time_P()
2 {
3     start_counter();
4     P();
5     return get_counter();
6 }
```

This could easily yield misleading results if some other process also executes between the two calls to the counter routines. This is especially a problem if either the machine is heavily loaded, or if the run time for `P` is especially long. This phenomenon is illustrated in Figure 9.11. This figure shows the result of repeatedly measuring a program that computes the sum of an array of 131,072 integers. The times have been converted into milliseconds. Note that the run times are all over 36 ms, greater than the timer interval. Two trials were run, each measuring 18 executions of the exact same procedure. The series labeled “Load 1” indicates the run times on a lightly loaded machine, where this is the only process actively running. All of the measurements are within 3.4% of the minimum run time. The series labeled “Load 4” indicates the run times when three other processes making heavy use of the CPU and memory system are also running.

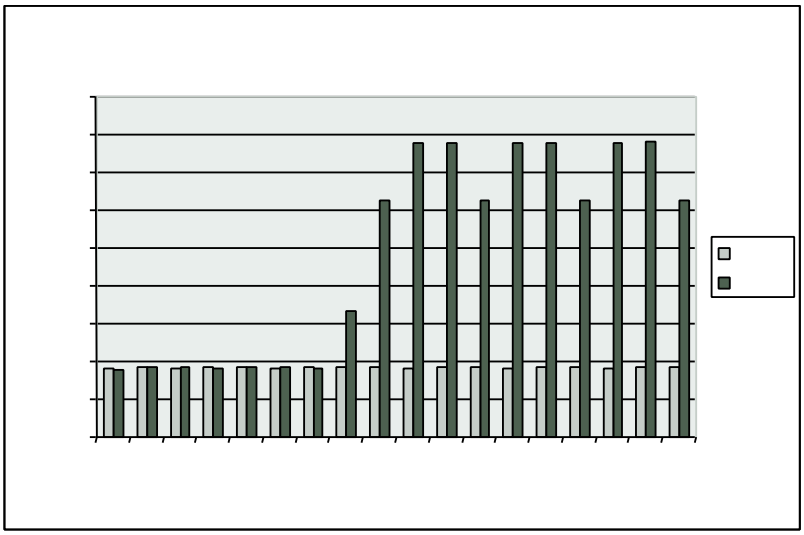

Figure 9.11: **Measurements of Long Duration Procedure under Different Loading Conditions** On a lightly loaded system, the results are consistent across samples, but on a heavily loaded system, many of the measurements overestimate the true execution time.

The first seven of these samples have times within 2% of the fastest Load 1 sample, but others range as much as 4.3 times greater.

As this example illustrates, context switching causes extreme variations in execution time. If a process is swapped out for an entire time interval it will fall behind by millions of instructions. Clearly, any scheme we devise to measure program execution times must avoid such large errors.

9.4.2 Caching and Other Effects

The effects of caching and branch prediction create smaller timing variations than does context switching. As an example, Figure 9.12 shows a series of measurements similar to those in Figure 9.11, except that the array is 4 times smaller, yielding execution times of around 8 ms. These execution times are shorter than the timer interval and therefore the executions are less likely to be affected by context switching. We see significant variations among the measurements—the slowest is 1.1 times slower the fastest, but none of these variations are as extreme as would be caused by context switching.

The variations shown in Figure 9.12 are due mainly to cache effects. The time to execute a block of code can depend greatly on whether or not the data and the instructions used by this code are present in the data and instruction caches at the beginning of execution.

As an example, we wrote two identical procedures, `procA` and `procB`, that are given a pointer of type `double *` and set the eight consecutive elements starting at this pointer to 0.0. We measured the number of clock cycles for various calls to these procedures with three different pointers: `b1`, `b2`, and `b3`. The call sequence and the resulting measurements are shown in Figure 9.13. The timings vary by almost a factor of 4, even though the calls perform identical computations. There were no conditional branches in this code, and hence we conclude that the variations must be due to cache effects.

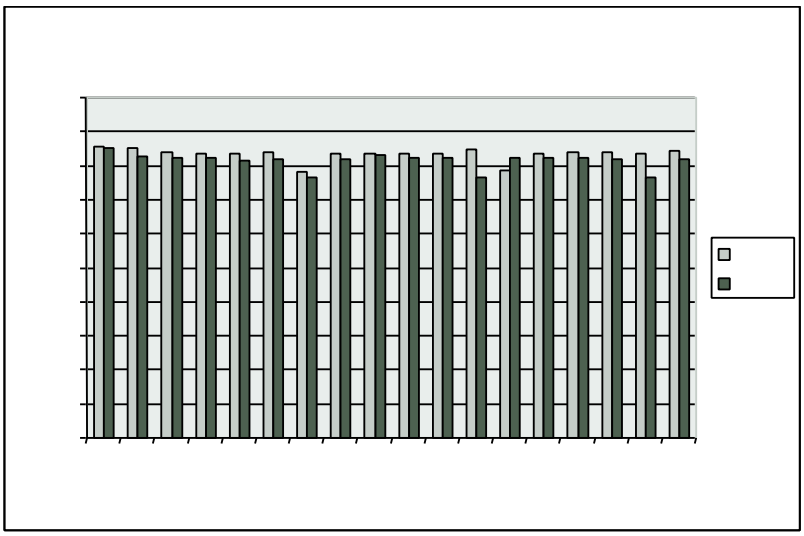

Figure 9.12: **Measurements of Short Duration Procedure under Different Loading Conditions** The variations are not as extreme as they were in Figure 9.11, but they are still unacceptably large.

Measurement	Call	Cycles
1	`procA(b1)`	399
2	`procA(b2)`	132
3	`procA(b3)`	134
4	`procA(b1)`	100
5	`procB(b1)`	317
6	`procB(b2)`	100

Figure 9.13: **Measurement Sequence with Identical Procedures Operating on Identical Data Sets.** The variations in these measurements are due to different miss conditions in the instruction and data caches.

Practice Problem 9.6:

Let c be the number of cycles that would be required by a call to `procA` or `procB` if there were no cache misses. For each computation, the cycles wasted due to cache misses can be apportioned between the different items needing to be brought into the cache:

- The instructions implementing the measurement code (e.g., `start_counter`, `get_counter`, and so on). Let the number of cycles for this be m.
- The instructions implementing the procedure being measured (`procA` or `procB`). Let the number of cycles for this be p.
- The data locations being updated (designated by `b1`, `b2`, or `b3`). Let the number of cycles for this be d.

Based on the measurements shown in Figure 9.13, give estimates of the values of c, m, p, and d.

Given the variations shown in these measurements, a natural question to ask is "Which one is right?" Unfortunately, the answer to this question is not simple. It depends on both the conditions under which our code will actually be used as well as the conditions under which we can get reliable measurements. One problem is that the measurements are not even consistent from one run to the next. The measurement table shown in Figure 9.13 show the data for just one testing run. In repeated tests, we have seen Measurement 1 range from 317 and 606, and Measurement 5 range from 301 to 326. On the other hand, the other four measurements only vary by at most a few cycles from one run to another.

Clearly Measurement 1 is an overestimate, because it includes the cost of loading the measurement code and data structures into cache. Furthermore, it is the most subject to wide variations. Measurement 5 includes the cost of loading `procB` into the cache. This is also subject to significant variations. In most real applications, the same code is executed repeatedly. As a result, the time to load the code into the instruction cache will be relatively insignificant. Our example measurements are somewhat artificial in that the effects of instruction cache misses were proportionally greater than what would occur in a real application.

To measure the time required by a procedure `P` where the effects of instruction cache misses are minimized we can execute the following code:

```
1 double time_P_warm()
2 {
3     P(); /* Warm up the cache */
4     start_counter();
5     P();
6     return get_counter();
7 }
```

Executing `P` once before starting the measurement will have the effect of bringing the code used by `P` into the instruction cache.

The code above also minimizes the effects of data cache misses, since the first execution of `P` will also have the effect of bringing the data accessed by `P` into the data cache. For procedures `procA` or `procB`, a measurement by `time_P_warm` would yield 100 cycles. This would be the right conditions to measure if we expect our code to access the same data repeatedly. For some applications, however, we would be more

likely to access new data with each execution. For example, a procedure that copies data from one region of memory to another would most likely be called under conditions where neither block is cached. Procedure `time_P_warm` would tend to underestimate the execution time for such a procedure. For `procA` or `procB`, it would yield 100 rather than the 132 to 134 measured when the procedure is applied to uncached data.

To force the timing code to measure the performance of a procedure where none of the data is initially cached, we can flush the cache of any useful data before performing the actual measurement. The following procedure does this for a system with caches of no more than 512KB:

code/perf/time_p.c

```
1  /* Number of bytes in the largest cache to be cleared */
2  #define CBYTES (1<<19)
3  #define CINTS (CBYTES/sizeof(int))
4
5  /* A large array to bring into cache */
6  static int dummy[CINTS];
7  volatile int sink;
8
9  /* Evict the existing blocks from the data caches */
10 void clear_cache()
11 {
12     int i;
13     int sum = 0;
14
15     for (i = 0; i < CINTS; i++)
16         dummy[i] = 3;
17     for (i = 0; i < CINTS; i++)
18         sum += dummy[i];
19     sink = sum;
20 }
```

code/perf/time_p.c

This procedure simply performs a computation over a very large array `dummy`, effectively evicting everything else from the cache. The code has several peculiar features to avoid common pitfalls. It both stores values into `dummy` and reads them back so that it will be cached regardless of the cache allocation policy. It performs a computation using array values and stores the result to a global integer (the declaration `volatile` indicates that any update to this variable must be performed), so that a clever optimizing compiler will not optimize away this part of the code.

With this procedure, we can get a measurement of `P` under conditions where its instructions are cached but its data is not by the following procedure:

```
1 double time_P_cold()
2 {
3     P(); /* Warm up data caches */
4     clear_cache(); /* Clear data caches */
5     start_counter();
```

```
6     P();
7     return get_counter();
8 }
```

Of course, even this method has deficiencies. On a machine with a unified L2 cache, procedure `clear_cache` will cause all instructions from `P` to be evicted. Fortunately, the instructions in the L1 instruction cache will remain. Procedure `clear_cache` also evicts much of the runtime stack from the cache, leading to an overestimate of the time required by `P` under more realistic conditions.

As this discussion shows, the effects of caching pose particular difficulties for performance measurement. Programmers have little control over what instructions and data get loaded into the caches and what gets evicted when new values must be loaded. At best, we can set up measurement conditions that somewhat match the anticipated conditions of our application by some combination of cache flushing and loading.

As mentioned earlier, the branch prediction logic also influences program performance, since the time penalty caused by branch instruction is much less when the branch direction and target are correctly predicted. This logic makes its predictions based on the past history of branch instructions that have been executed. When the system switches from one process to another, it initially makes predictions about branches in the new process based on those executed in the previous process. In practice, however, these effects create only minor performance variations from one execution of a program to another. The predictions depend most strongly on recent branches, and hence the influence by one process on another is very small.

9.4.3 The K-Best Measurement Scheme

Although our measurements using cycle timers are vulnerable to errors due to context switching, cache operation, and branch prediction, one important feature is that the errors will always cause overestimates of the true execution time. Nothing done by the processor can artificially speed up a program. We can exploit this property to get reliable measurements of execution times even when there are variances due to context switching and other effects.

Suppose we repeatedly execute a procedure and measure the number of cycles using either `time_P_warm` or `time_P_cold`. We record the K (e.g., 3) fastest times. If we find these measurements agree within some small tolerance ϵ (e.g., 0.1%), then it seems reasonable that the fastest of these represents the true execution time of the procedure. As an example, suppose for the runs shown in Figure 9.11 we set the tolerance to 1.0%. Then the fastest six measurements for Load 1 are within this tolerance, as are the fastest three for Load 4. We would therefore conclude that the run times are 35.98 ms and 35.89 ms, respectively. For the Load 4 case, we also see measurements clustered around 125.3 ms, and six around 155.8 ms, but we can safely discard these as overestimates.

We call this approach to measurement the "K-Best Scheme." It requires setting three parameters:

K: The number of measurements we require to be within some close range of the fastest.

ϵ: How close the measurements must be. That is, if the measurements in ascending order are labeled $v_1, v_2, \ldots, v_i, \ldots$, then we require $(1 + \epsilon)v_1 \geq v_K$.

M: The maximum number of measurements before we give up.

Our implementation performs a series of trials, and maintains an array of the K fastest times in sorted order. With each new measurement, it checks whether it is faster than the current one in array position K. If so, it replaces array element K and then performs a series of interchanges between adjacent array positions to move this value to the appropriate position in the array. This process continues until either the error criterion is satisfied, in which case we indicate that the measurements have "converged," or we exceed the limit M, in which case we indicate that the measurements failed to converge.

Experimental Evaluation

We conducted a series of experiments to test the accuracy of the K-best measurement scheme. Some issues we wished to determine were:

1. Does this scheme produce accurate measurements?
2. When and how quickly do the measurements converge?
3. Can the scheme determine the accuracy of its own measurements?

One challenge in designing such an experiment is to know the actual run times of the programs we are trying to measure. Only then can we determine the accuracy of our measurements. We know that our cycle timer gives accurate results as long as the computation we are measuring do not get interrupted. The likelihood of an interrupt is small for computations that are much shorter than the timer interval and when running on a lightly loaded machine. We exploit these properties to get reliable estimates of true run times.

As our object to measure, we used a procedure that repeatedly writes values to an array of 2,048 integers and then reads them back, similar to the code for `clear_cache`. By setting the number of repetitions r, we could create computations requiring a range of times. We first determined the *expected* run time of this procedure as a function of r, denoted $T(r)$, by timing it for r ranging from 1 to 10 (giving times ranging from 0.09 to 0.9 milliseconds), and performing a least squares fit to find a formula of the form $T(r) = mr + b$. By using small values of r, performing 100 measurements for each value of r, and running on a lightly loaded system we were able to get a very accurate characterization of $T(r)$. Our least squares analysis indicated that the formula $T(r) = 49273.4r + 166$ (in units of clock cycles) fits this data with a maximum error less than 0.04%. This gave us confidence in our ability to accurately predict the actual computation time for the procedure as a function of r.

We then measured performance using the K-best scheme with parameters $K = 3$, $\epsilon = 0.001$, and $M = 30$. We did this for a number of values of r to get expected run times in a range from 0.27 to 50 milliseconds. For each of the resulting measurements $M(r)$ we computed the measurement error $E_m(r)$ as $E_m(r) = (M(r) - T(r))/T(r)$. Figure 9.14 shows an experimental validation of the K-best scheme on an Intel Pentium III running Linux. In this figure we show the measurement error $E_m(r)$ as a function of $T(r)$, where we show $T(r)$ in units of milliseconds. Note that we show $E_m(r)$ on a logarithmic scale; each horizontal line represents an order of magnitude difference in measurement error. In order to be accurate within 1% we must have an error below 0.01. We do not attempt to show any errors smaller than 0.001 (i.e., 0.1%), since our testing setup does not provide high enough precision for this.

The three series indicate the errors under three different loading conditions. Observe that in all three cases the measurements for run times shorter than around 7.5 ms were very accurate. Thus, our scheme can be

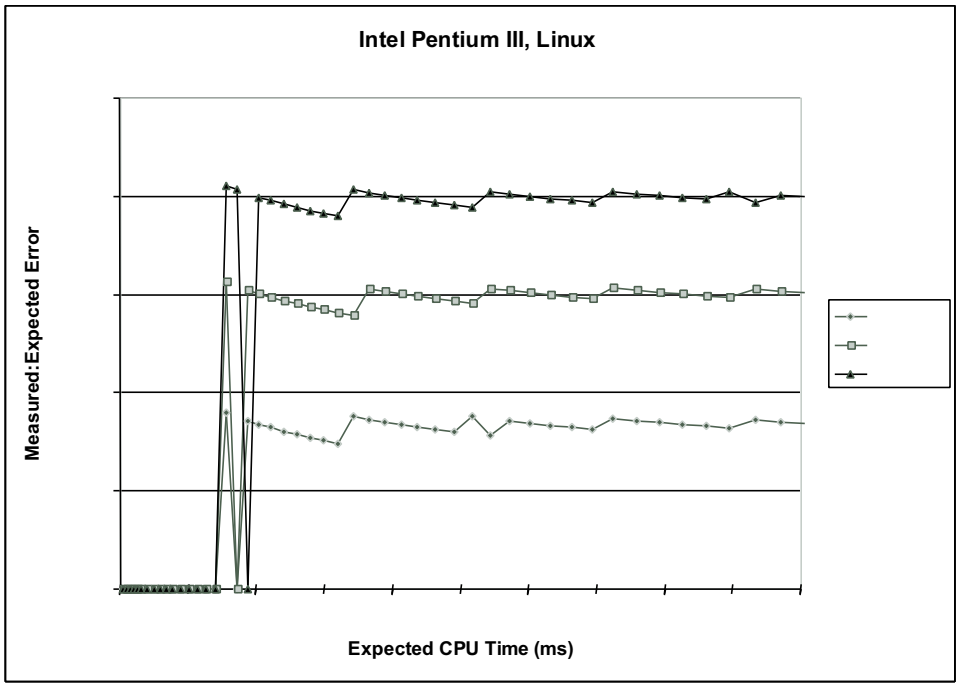

Figure 9.14: **Experimental Validation if K-Best Measurement Scheme on Linux System** We can consistently obtain very accurate measurements (around 0.1% error) for execution times up to around 8 ms. Beyond this, we encounter a systematic overestimate of around 4 to 6% on a lightly loaded machine and very poor results on a heavily loaded machine.

used to measure relatively short execution times even on a heavily loaded machine. Series "Load 1" indicates the case where there is only one active process. For execution times above 10 ms, the measurements T_m consistently overestimate the computation times T_c by around 4 to 6%. These overestimates are due to the time spent handling timer interrupts. They are consistent with the trace shown in Figure 9.3, showing that even on a lightly loaded machine, an application program can execute for only 95 to 96% of the time. Series "Load 2" and "Load 11" show the performance when other processes are actively executing. In both cases, the measurements become hopelessly inaccurate for execution times above around 7 ms. Note that an error of 1.0 means that T_m is twice T_c, while an error of 10.0 means that T_m is eleven times greater than T_c. Evidently, the operating system schedules each active process for one time interval. When n processes are active, each one only gets $1/n$th of the processor time.

From these results, we conclude that the K-best scheme provides accurate results only for very short computations. It is not really good enough for measuring execution times longer than around 7 ms, especially in the presence of other active processes.

Unfortunately, we found that our measurement program could not reliably determine whether or not it had obtained an accurate measurement. Our measurement procedure computes a prediction of its error as $E_p(r) = (v_k - v_1)/v_1$, where v_i is the i th smallest measurement. That is, it computes how well it achieves our convergence criterion. We found these estimates to be wildly optimistic. Even for the Load 11 case, where the measurements were off by a factor of 10, the program consistently estimated its error to be less than 0.001.

Setting the value of K

In our earlier experiments, we arbitrarily chose a value of 3 for the parameter K, determining the number of measurements we require to be within a small factor of the fastest in order to terminate. To more carefully evaluate the effect of this factor, we performed a series of measurements using values of K ranging from 1 to 5, as shown in Figure 9.15. We performed these measurements for execution times ranging up to 9 ms, since this is the upper limit of times for which our scheme can get useful results.

When we have $K = 1$, the procedure returns after making a single measurement. This can yield highly erratic results, especially when the machine is heavily loaded. If a timer interrupt happens to occur, the result is extremely inaccurate. Even without such a catastrophic event, the measurements will be subject to many sources of inaccuracy. Setting K to 2 greatly improves the accuracy. For execution times less than 5 ms, we consistently get accuracy better than 0.1%. Setting K even higher gives better results, both in consistency and accuracy, up to a limit of around 8 ms. These experiments show that our initial guess of $K = 3$ is a reasonable choice.

Compensating for Timer Interrupt Handling

The timer interrupts occur in a predictable way and cause a large systematic error in our measurements for execution times over around 7 ms. It would be good to remove this bias by subtracting from the measured run time for a program an estimate of the time spent handling timer interrupts. This requires determining two factors.

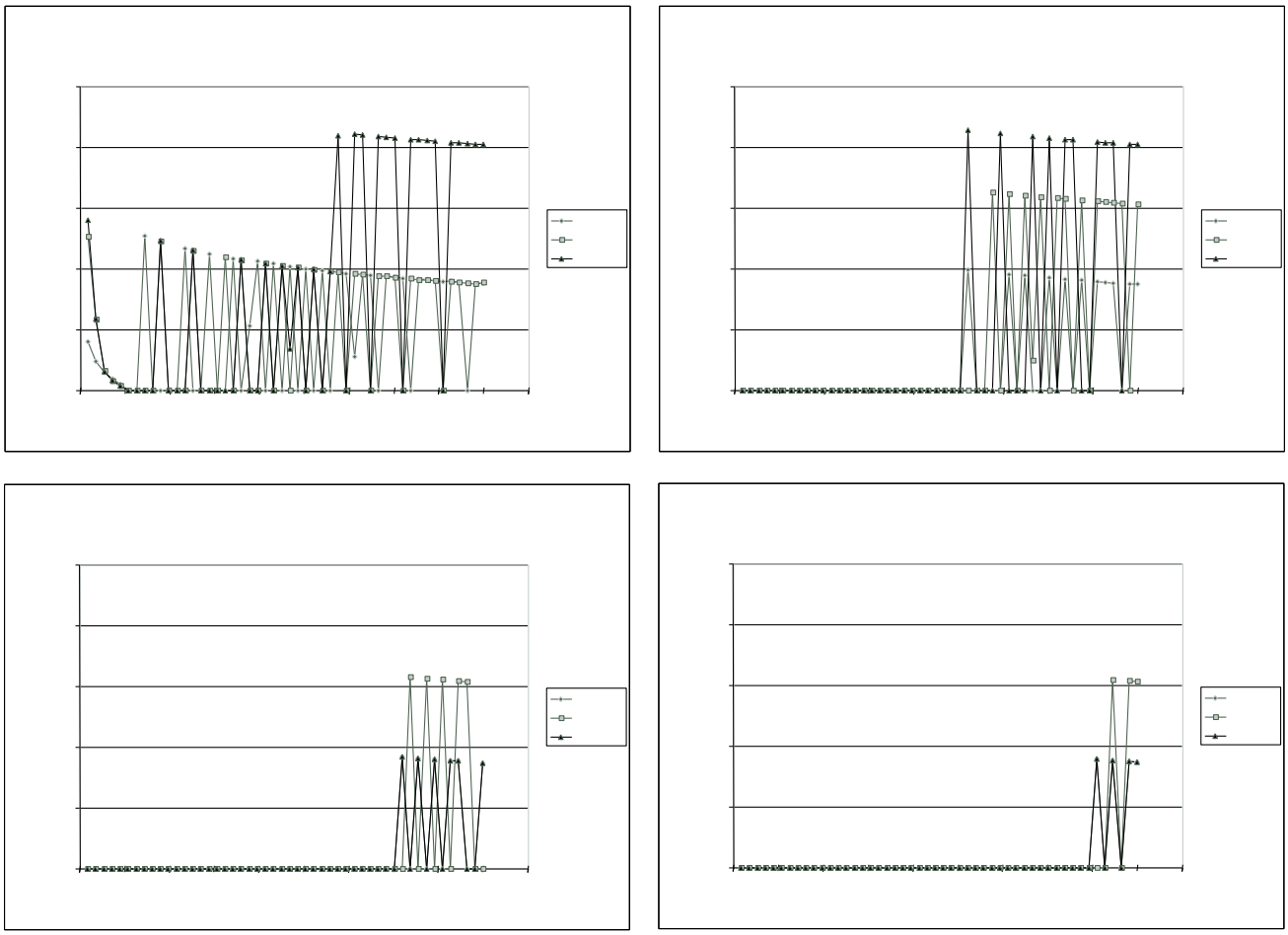

Figure 9.15: **Effectiveness of K-best scheme for different values of K.** K must be at least 2 to have reasonable accuracy. Values greater than 2 help on heavily loaded systems as the program times approach the timer interval.

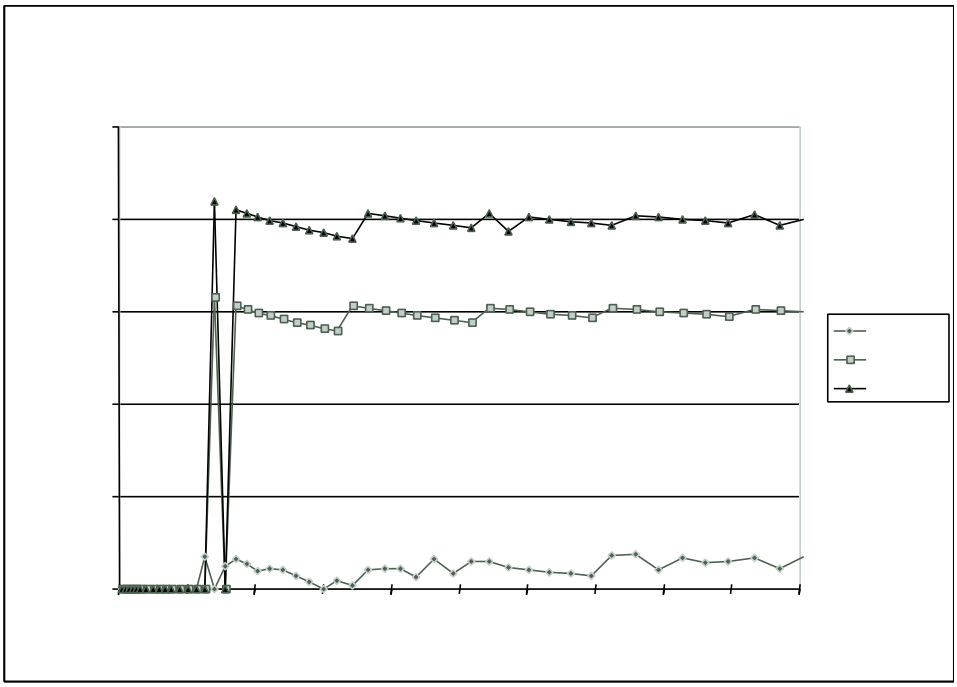

Figure 9.16: **Measurements with Compensation for Timer Interrupt Overhead** This approach greatly improves the accuracy of longer duration measurements on a lightly loaded machine.

1. We must determine how much time is required to handle a single timer interrupt. To preserve the property that we never underestimate the execution time of the procedure, we should determine the minimum number of clock cycles required to service a timer interrupt. That way we will never overcompensate.

2. We must determine how many timer interrupts occur during the period we are measuring.

Using a method similar to that used to generate the traces shown in Figures 9.3 and 9.5, we can detect periods of inactivity and determine their duration. Some of these will be due to timer interrupts, and some will be due to other system events. We can determine whether a timer interrupt has occurred by using the `times` procedure, since the value it returns will be increase one tick each time a timer interrupt occurs. We conducted such an evaluation for 100 periods of inactivity and found that the minimum timer interrupt processing period required 251,466 cycles. To determine the number of timer interrupts that occur during the program we are measuring, we simply call the `times` function twice—once before and once after the program, and then compute their difference.

Figure 9.16 shows the results obtained by this revised measurement scheme. As the figure illustrates, we can now get very accurate (within 1.0%) measurements on a lightly loaded machine, even for programs that execute for multiple time intervals. By removing the systematic error of timer interrupts, we now have a very reliable measurement scheme. On the other hand, we can see that this compensation does not help for programs running on heavily loaded machines.

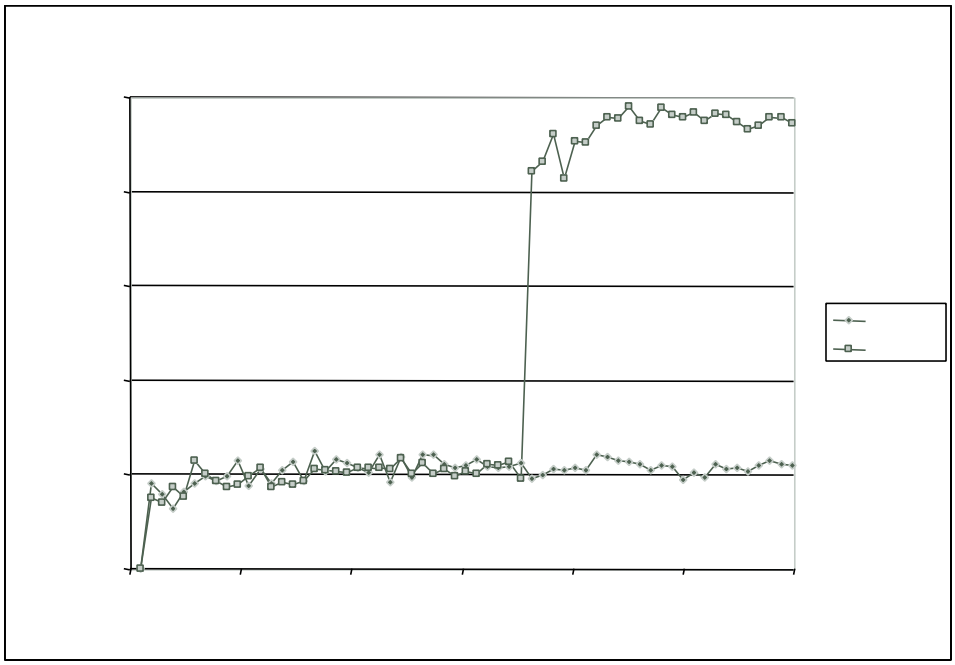

Figure 9.17: **Experimental Validation if K-Best Measurement Scheme on IA32/Linux System with Older Version of the Kernel.** On this system we could get more accurate measurements even for programs with longer execution times, especially on lightly loaded machine.

Evaluation on Other Machines

Since our scheme depends heavily on the scheduling policy of the operating system, we also ran experiments on three other system configurations:

1. Intel Pentium III running older version (2.0.36 vs. 2.2.16) of the Linux kernel.
2. Intel Pentium II running Windows-NT. Although this system uses an IA32 processor, the operating system is fundamentally different from Linux.
3. Compaq Alpha running Tru64 Unix. This uses a very different processor, but the operating system is similar to Linux.

As Figure 9.17 indicates, the performance characteristics under an older version of Linux are very different. On a lightly loaded machine, the measurements are within 0.2% accuracy for programs of almost arbitrary duration. We found that the processor spends only around 3500 cycles processing a timer interrupt with this version of Linux. Even on a heavily loaded machine, it will allow processes to run up to around 180 ms at a time. This experiment shows that the internal details of the operating system can greatly affect system performance and our ability to obtain accurate measurements.

Figure 9.18 shows the results on the Windows-NT system. Overall, the results are similar to those for the older Linux system. For short computations, or on a lightly loaded machine, we could get accurate

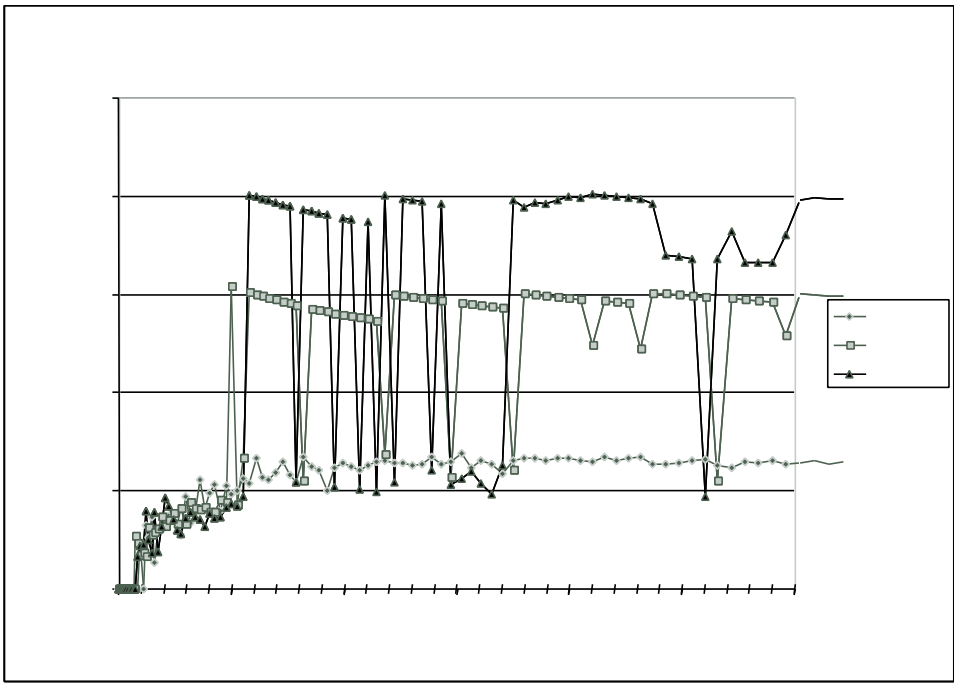

Figure 9.18: **Experimental Validation of K-Best Measurement Scheme on Windows-NT System.** On a lightly loaded system, we can consistently obtain accurate measurements (around 1.0% error). On a heavily loaded system, the accuracy becomes very poor for measurements longer than around 48 ms.

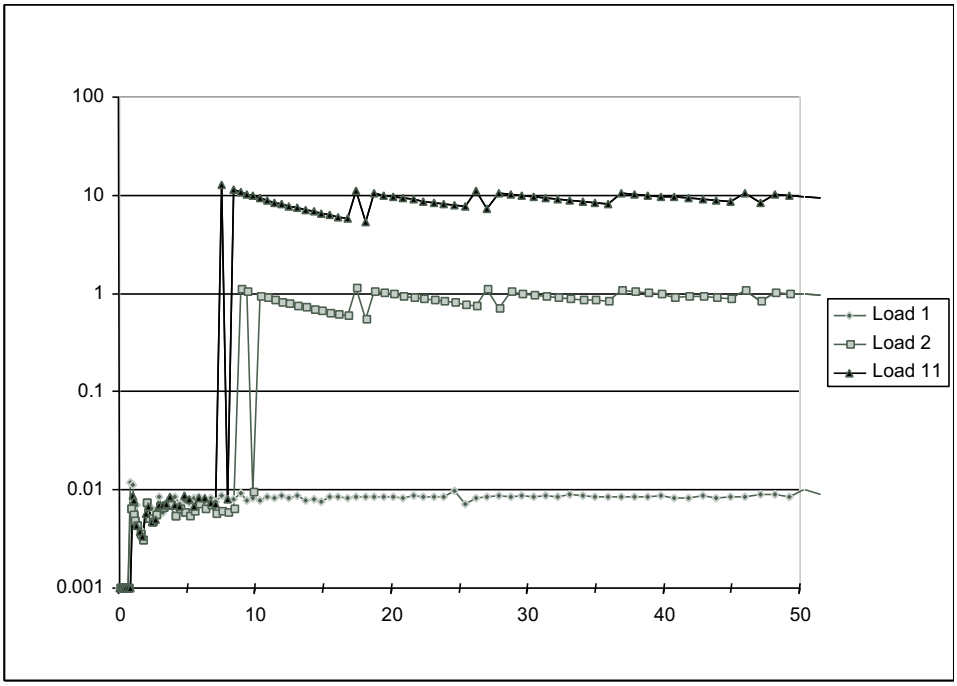

Figure 9.19: **Experimental Validation if K-Best Measurement Scheme on Compaq Alpha System.** For a lightly loaded system, we can consistently obtain accurate ($< 1.0\%$ error) measurements. For a heavily loaded system, durations beyond around 10 ms cannot be measured accurately.

measurements. In this case, our accuracies were around 0.01 (i.e., 1.0%), rather than 0.001. Still, this is good enough for most applications. In addition, our threshold between reliable and unreliable measurements on a heavily loaded machine was around 48 ms. One interesting feature is that we were sometimes able to get accurate measurements on a heavily loaded machine even for computations ranging up to 245 ms. Evidently, the NT scheduler will sometimes allow processes to remain active for longer durations, but we cannot rely on this property.

The Compaq Alpha results are shown in Figure 9.19. Again, we find that on a lightly loaded machine, programs of almost arbitrary duration can be measured with an error of less than 1%. On a heavily loaded machine, only programs with durations less than around 10 ms can be measured accurately.

Practice Problem 9.7:

Suppose we wish to measure a procedure that requires t milliseconds. The machine is heavily loaded and hence will not allow our measurement process to run more than 50 ms at a time.

A. Each trial involves measuring one execution of the procedure. What is the probability this trial will be allowed to run to completion without being swapped out, assuming it starts at some arbitrary point within the 50 ms time segment? Express your answer as a function of t, considering all possible values of t.

B. What is the expected number of trials required so that three of them are reliable measurements of the procedure, i.e., each runs within a single time segment? Express your answer as a function of t. What values do you predict for $t = 20$ and $t = 40$?

Observations

These experiments demonstrate that the K-best measurement scheme works fairly well on a variety of machines. On lightly loaded processors, it consistently gets accurate results on most machines, even for computations with long durations. Only the newer version of Linux incurs a sufficiently high timer interrupt overhead to seriously affect the measurement accuracy. For this system, compensating for this overhead greatly improves the measurement accuracy.

On heavily loaded machines, getting accurate measurements becomes difficult as execution times become longer. Most systems have some maximum execution time beyond which the measurement accuracy becomes very poor. The exact value of this threshold is highly system dependent, but typically ranges between 10 and 200 milliseconds.

9.5 Time-of-Day Measurements

Our use of the IA32 cycle counter provides high-precision timing measurements, but it has the drawback that it only works on IA32 systems. It would be good to have a more portable solution. We have seen that the library functions `times` and `clock` are implemented using interval counters and hence are not very accurate.

Another possibility is to use the library function `gettimeofday`. This function queries the *system clock* to determine the current date and time.

```
#include "time.h"

struct timeval {
   long tv_sec; /* Seconds */
   long tv_usec; /* Microseconds */
}

int gettimeofday(struct timeval *tv, NULL);
```

Returns: 0 for success, -1 for failure

The function writes the time into a structure passed by the caller that includes one field in units of seconds, and another field in units of microseconds. The first field encodes the total number of seconds that have elapsed since January 1, 1970. (This is the standard reference point for all Unix systems.) Note that the second argument to `gettimeofday` should simply be `NULL` on Linux systems, since it refers to an unimplemented feature for performing time zone correction.

Practice Problem 9.8:

On what date will the `tv_sec` field written by `gettimeofday` become negative on a 32-bit machine?

As shown in Figure 9.20, we can can use `gettimeofday` to create a pair of timer functions `start_timer` and `get_timer` that are similar to our cycle-timing functions, except that they measure time in seconds rather than clock cycles.

code/perf/tod.c

```
#include <sys/time.h>
#include <unistd.h>

static struct timeval tstart;

/* Record current time */
void start_timer()
{
    gettimeofday(&tstart, NULL);
}

/* Get number of seconds since last call to start_timer */
double get_timer()
{
    struct timeval tfinish;
    long sec, usec;

    gettimeofday(&tfinish, NULL);
    sec = tfinish.tv_sec - tstart.tv_sec;
    usec = tfinish.tv_usec - tstart.tv_usec;
    return sec + 1e-6*usec;
}
```

code/perf/tod.c

Figure 9.20: **Timing Procedures Using Unix Time of Day Clock.** This code is very portable, but its accuracy depends on how the clock is implemented.

System	Resolution (μs)	Latency (μs)
Pentium II, Windows-NT	10,000	5.4
Compaq Alpha	977	0.9
Pentium III Linux	1	0.9
Sun UltraSparc	2	1.1

Figure 9.21: **Characteristics of `gettimeofday` Implementations.** Some implementations use interval counting, while others use cycle timers. This greatly affects the measurement precision.

The utility of this timing mechanism depends on how `gettimeofday` is implemented, and this varies from one system to another. Although the fact that the function generates a measurement in units of microseconds looks very promising, it turns out that the measurements are not always that precise. Figure 9.21 shows the result of testing the function on several different systems. We define the *resolution* of the function to be the minimum time value the timer can resolve. We computed this by repeatedly calling `gettimeofday` until the value written to the first argument changed. The resolution is then the number of microseconds by which it changed. As indicated in the table, some implementations can actually resolve times at a microsecond level, while others are much less precise. These variations occur, because some systems use cycle counters to implement the function, while others use interval counting. In the former case, the resolution can be very high—potentially higher than the 1 microsecond resolution provided by the data representation. In the latter case, the resolution will be poor—no better than what is provided by functions `times` and `clock`.

Figure 9.21 also shows the *latency* required by a call to `get_timer` on various systems. This property indicates the minimum time required for a call to the function. We computed this by repeatedly calling the function until one second had elapsed and dividing 1 by the number of calls. As can be seen, this function requires around 1-microsecond on most systems, and several microseconds on others. By comparison, our procedure `get_counter` requires only around 0.2 microseconds per call. In general, system calls involve more overhead than ordinary function calls. This latency also limits the precision of our measurements. Even if the data structure allowed expressing time in units with higher resolution, it is unclear how much more precisely we could measure time when each measurement incurs such a long delay.

Figure 9.22 shows the performance we get from an implementation of the K-best measurement scheme using `gettimeofday` rather than our own functions to access the cycle counter. We show the results on two different machines to illustrate the effect of the time resolution on accuracy. The measurements on a Windows-NT system show characteristics similar to those we found for Linux using `times` (Figure 9.8). Since `gettimeofday` is implemented using the process timers, the error can be negative or positive, and it is especially erratic for short duration measurements. The accuracy improves for longer durations, to the point where the error is less than 2.0% for durations greater than 200 ms. The measurements on a Linux system give results similar to those seen when making direct use of cycle counters. This can be seen by comparing the measurements to the Load 1 results in Figure 9.14 (without compensation) and in Figure 9.16 (with compensation). Using compensation, we can achieve better than 0.04% accuracy, even for measurements as long as 300 ms. Thus, `gettimeofday` performs just as well as directly accessing the cycle counter on this machine.

9.6 Putting it Together: An Experimental Protocol

We can summarize our experimental findings in the form of a protocol to determine how to answer the question "How fast does Program X run on Machine Y?"

- If the anticipated run times of X are long (e.g., greater than 1.0 second), then interval counting should work well enough and be less sensitive to processor load.
- If the anticipated run times of X are in a range of around 0.01 to 1.0 seconds, then it is essential to perform measurements on a lightly loaded system, and to use accurate, cycle-based timing. We should

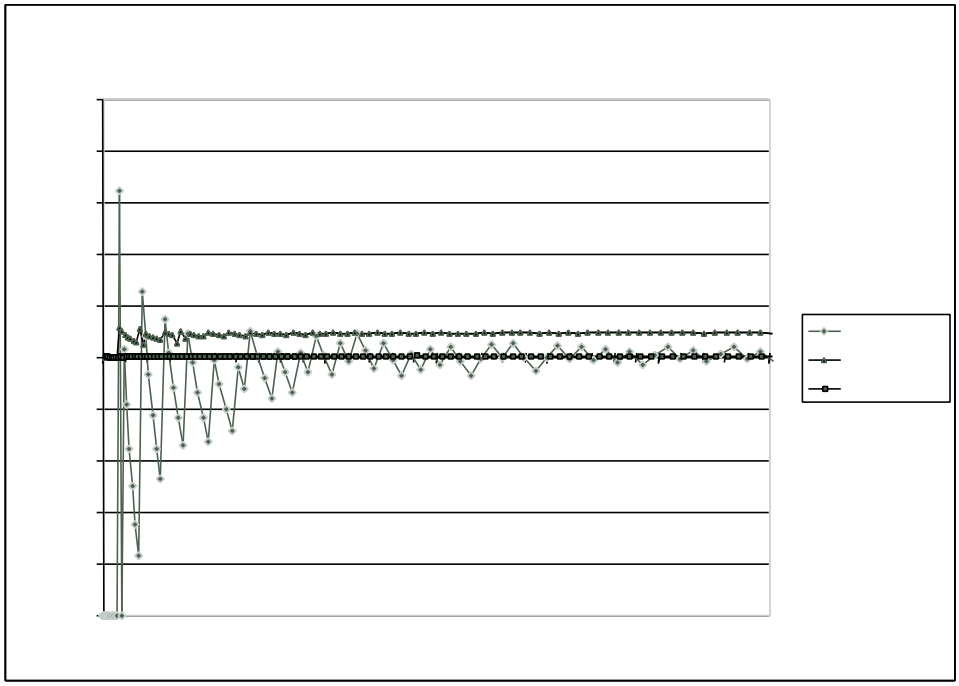

Figure 9.22: **Experimental Validation if K-Best Measurement Scheme Using `gettimeofday` Function.** Linux implements this function using cycle counters and hence achieve the same accuracy as do our own timing routines. Windows-NT implements this function using interval counting, and hence the accuracy is low, especially for small duration measurements.

perform tests of the `gettimeofday` library function to determine whether its implementation on machine Y is cycle based or interval based.

  - If the function is cycle based, then use it as the basis for the K-best timing function.
  - If the function is interval based, then we must find some method of using the machine's cycle counters. This may require assembly language coding.

- If the anticipated run times of X are less than around 0.01 second, then accurate measurements can be performed even on a heavily loaded system, as long as it is uses cycle-based timing. We then proceed in implementing a K-best timing function using either `gettimeofday` or by direct access to the machine's cycle counter.

9.7 Looking into the Future

There are several features that are being incorporated into systems that will have significant impact on performance measurements.

- *Process-specific cycle timing*. It is relatively easy for the operating system to manage the cycle counter so that it indicates the elapsed number of cycles for a specific process. All that is required is to store the count as part of the process' state. Then when the process is reactivated, the cycle counter is set to the value it had when the process was last deactivated, effectively freezing the counter while the process is inactive. Of course, the counter will still be affected by the overhead of kernel operation and by cache effects, but at least the effects of other processes will not be as severe. Already some systems support this feature. In terms of our protocol, this will allow us to use cycle-based timing to get accurate measurements of durations greater than around 0.01 second, even on heavily loaded systems.

- *Variable Rate Clocks*. In an effort to reduce power consumption, future systems will vary the clock rate, since power consumption is directly proportional to the clock rate. In that case, we will not have a simple conversion between clock cycles and nanoseconds. It even becomes difficult to know which unit should be used to express program performance. For a code optimizer, we gain more insight by counting cycles, but for someone implementing an application with real-time performance constraints, actual run times are more important.

9.8 Life in the Real World: An Implementation of the K-Best Measurement Scheme

We have created a library function `fcyc` that uses the K-best scheme to measure the number of clock cycles required by a function `f`.

```
#include "clock.h"
#include "fcyc.h"

typedef void (*test_funct)(int *);

double fcyc(test_funct f, int *params);
```

Returns: number of cycles used by `f` running `params`

The parameter `params` is a pointer to an integer. In general, it can point to an array of integers that form the parameters of the function being measured. For example, when measuring the lower-case conversion functions `lower1` and `lower2`, we pass as a parameter a pointer to a single `int`, which is the length of the string to be converted. When generating the memory mountain (Chapter 6, we pass a pointer to an array of size two containing the size and the stride.

There are a number of parameters that control the measurement, such as the values of K, ϵ, and M, and whether or not to clear the cache before each measurement. These parameters can be set by functions that are also in the library. See the file `fcyc.h` for details.

9.9 Summary

We have seen that computer systems have two fundamentally different methods of recording the passage of time. Timer interrupts occur at a rate that seems very fast when viewed on a macroscopic scale but very slow when viewed on a microscopic scale. By counting intervals, the system can get a very rough measure of program execution time. This method is only useful for long-duration measurements. Cycle counters are very fast, giving good measurements on a microscopic scale. For cycle counters that measure absolute time, the effects of context switching can induce error ranging from small (on a lightly loaded system) to very large (on a heavily loaded system). Thus, no scheme is ideal. It is important to understand the accuracy achievable on a particular system.

Through this effort to devise an accurate timing scheme and to evaluate its performance on a number of different systems, we have learned some important lessons:

- *Every system is different*. Details about the hardware, operating system, and library function implementations can have a significant effect on what kinds of programs can be measured and with what accuracy.

- *Experiments can be quite revealing*. We gained a great deal of insight into the operating system scheduler running simple experiments to generate activity traces. This led to the compensation scheme that greatly improves accuracy on a lightly loaded Linux system. Given the variations from one system to the next, and even from one release of the OS kernel to the next, it is important to be able to analyze and understand the many aspects of a system that affect its performance.

- *Getting accurate timings on heavily loaded systems is especially difficult*. Most systems researchers do all of their measurements on dedicated benchmark systems. They often run the system with many OS and networking features disabled to reduce sources of unpredictable activity. Unfortunately, ordinary programmers to not have this luxury. They must share the system with other users. Even on

heavily loaded systems, our K-best scheme is reasonably robust for measuring durations shorter than the timer interval.

- *The experimental setup must control some sources of performance variations*. Cache effects can greatly affect the execution time for a program. The conventional technique is to make sure that the cache is flushed of any useful data before the timing begins, or else that it is loaded with any data that would typically be in the cache initially.

Through a series of experiments, we were able to design and validate the K-best timing scheme, where we make repeated measurements until the fastest K are within some close range to each other. On some systems, we can make measurements using the library functions for finding the time of day. On other systems, we must access the cycle counters via assembly code.

Bibliographic Notes

There is surprisingly little literature on program timing. Stevens' Unix programming book [77] documents all of the different library functions for program timing. Wadleigh and Crawford's book on software optimization [81] describe code profiling and standard timing functions.

Homework Problems

Homework Problem 9.9 [Category 2]:

Determine the following based on the trace shown in Figure 9.3. Our program estimated the clock rate as 549.9 MHz. It then computed the millisecond timings in the trace by scaling the cycle counts. That is, for a time expressed in cycles as c, the program computed the millisecond timing as $c/549900$. Unfortunately, the program's method of estimating the clock rate is imperfect, and hence some of the millisecond timings are slightly inaccurate.

A. The timer interval for this machine is 10 ms. Which of the time periods above were initiated by a timer interrupt?

B. Based on this trace, what is the minimum number of clock cycles required by the operating system to service a timer interrupt?

C. From the trace data, and assuming the timer interval is exactly 10.0 ms, what can you infer as the value of the true clock rate?

Homework Problem 9.10 [Category 2]:

Write a program that uses library functions `sleep` and `times` to determine the approximate number of clock ticks per second. Try compiling the program and running it on multiple systems. Try to find two different systems that produce results that differ by at least a factor of two.

Homework Problem 9.11 [Category 1]:

We can use the cycle counter to generate activity traces such as was shown in Figures 9.3 and 9.5. Use the functions `start_counter` and `get_counter` to write a function:

```
#include "clock.h"

int inactiveduration(int thresh);
```

Returns: Number of inactive cycles

This function continually checks the cycle counter and detects when two successive readings differ by more than `thresh` cycles, an indication that the process has been inactive. Return the duration (in clock cycles) of that inactive period.

Homework Problem 9.12 [Category 1]:

Suppose we call function `mhz` (Figure 9.10) with parameter `sleeptime` equal to 2. The system has a 10 ms timer interval. Assume that `sleep` is implemented as follows. The processor maintains a counter that is incremented by one every time a timer interrupt occurs. When the system executes `sleep(`x`)`, the system schedules the process to be restarted when the counter reaches $t+100x$, where t is the current value of the counter.

A. Let w denote the time that our process is inactive due to the call to `sleep`. Ignoring the various overheads of function calls, timer interrupts, etc., what range of values can w have?

B. Suppose a call to `mhz` yields 1000.0. Again ignoring the various overheads, what is the possible range of the true clock rate?