6 Measurement tools and techniques

'When the only tool you have is a hammer, every problem begins to resemble a nail.'

Abraham Maslow

The previous chapters have discussed what performance metrics may be useful for the performance analyst, how to summarize measured data, and how to understand and quantify the systematic and random errors that affect our measurements. Now that we know what to do with our measured values, this chapter presents several tools and techniques for actually measuring the values we desire.

The focus of this chapter is on fundamental measurement concepts. The goal is not to teach you how to use specific measurement tools, but, rather, to help you understand the strengths and limitations of the various measurement techniques. By the end of this chapter, you should be able to select an appropriate measurement technique to determine the value of a desired performance metric. You also should have developed some understanding of the trade-offs involved in using the various types of tools and techniques.

6.1 Events and measurement strategies

There are many different types of performance metrics that we may wish to measure. The different strategies for measuring the values of these metrics are typically based around the idea of an *event*, where an event is some predefined change in the system state. The precise definition of a specific event is up to the performance analyst and depends on the metric being measured. For instance, an event may be defined to be a memory reference, a disk access, a network communication operation, a change in a processor's internal state, or some pattern or combination of other subevents.

6.1.1 Events-type classification

The different types of metrics that a performance analyst may wish to measure can be classified into the following categories based on the type of event or events that comprise the metric.

1. **Event-count metrics.** Metrics that fall into this category are those that are simple counts of the number of times a specific event occurs. Examples of event-count metrics include the number of page faults in a system with virtual memory, and the number of disk input/output requests made by a program.
2. **Secondary-event metrics.** These types of metrics record the values of some secondary parameters whenever a given event occurs. For instance, to determine the average number of messages queued in the send buffer of a communication port, we would need to record the number of messages in the queue each time a message was added to, or removed from, the queue. Thus, the triggering event is a message-enqueue or -dequeue operation, and the metrics being recorded are the number of messages in the queue and the total number of queue operations. We may also wish to record the size (e.g. the number of bytes) of each message sent to later determine the average message size.
3. **Profiles.** A profile is an aggregate metric used to characterize the overall behavior of an application program or of an entire system. Typically, it is used to identify where the program or system is spending its execution time.

6.1.2 Measurement strategies

The above event-type classification can be useful in helping the performance analyst decide on a specific strategy for measuring the desired metric, since different types of measurement tools are appropriate for measuring different types of events. These different measurement tools can be categorized on the basis of the fundamental strategy used to determine the actual values of the metrics being measured. One important concern with any measurement strategy is how much it *perturbs* the system being measured. This aspect of performance measurement is discussed further in Section 6.6.

1. **Event-driven.** An event-driven measurement strategy records the information necessary to calculate the performance metric whenever the preselected event or events occur. The simplest type of event-driven measurement tool uses a simple counter to directly count the number of occurrences of a specific event. For example, the desired metric may be the number of page faults that occur during the execution of an application program. To find this value, the performance analyst most likely would have to modify the page-fault-handling

routine in the operating system to increment a counter whenever the routine is entered. At the termination of the program's execution, an additional mechanism must be provided to dump the contents of the counter.

One of the advantages of an event-driven strategy is that the system overhead required to record the necessary information is incurred only when the event of interest actually occurs. If the event never occurs, or occurs only infrequently, the perturbation to the system will be relatively small. This characteristic can also be a disadvantage, however, when the events being monitored occur very frequently.

When recording high-frequency events, a great deal of overhead may be introduced into a program's execution, which can significantly alter the program's execution behavior compared with its uninstrumented execution. As a result, what the measurement tool measures need not reflect the typical or average behavior of the system. Furthermore, the time between measurements depends entirely on when the measured events occur so that the inter-event time can be highly variable and completely unpredictable. This can increase the difficulty of determining how much the measurement tool actually perturbs the executing program. Event-driven measurement tools are usually considered most appropriate for low-frequency events.

2. **Tracing.** A tracing strategy is similar to an event-driven strategy, except that, rather than simply recording that fact that the event has occurred, some portion of the system state is recorded to uniquely identify the event. For example, instead of simply counting the number of page faults, a tracing strategy may record the addresses that caused each of the page faults. This strategy obviously requires significantly more storage than would a simple count of events. Additionally, the time required to save the desired state, either by storing it within the system's memory or by writing to a disk, for instance, can significantly alter the execution of the program being measured.
3. **Sampling.** In contrast to an event-driven measurement strategy, a sampling strategy records at fixed time intervals the portion of the system state necessary to determine the metric of interest. As a result, the overhead due to this strategy is independent of the number of times a specific event occurs. It is instead a function of the sampling frequency, which is determined by the resolution necessary to capture the events of interest.

   The sampling of the state of the system occurs at fixed time intervals that are independent of the occurrence of specific events. Thus, not every occurrence of the events of interest will be recorded. Rather, a sampling strategy produces a statistical summary of the overall behavior of the system. Consequently, events that occur infrequently may be completely missed by this statistical approach. Furthermore, each run of a sampling-based experiment is likely to produce a different result since the samples occur asynchronously with respect to a program's execution. Nevertheless, while the exact behavior may differ, the statistical behavior should remain approximately the same.

4. **Indirect.** An indirect measurement strategy must be used when the metric that is to be determined is not directly accessible. In this case, you must find another metric that can be measured directly, from which you then can deduce or derive the desired performance metric. Developing an appropriate indirect measurement strategy, and minimizing its overhead, relies almost completely on the cleverness and creativity of the performance analyst.

The unique characteristics of these measurement strategies make them more or less appropriate for different situations. Program tracing can provide the most detailed information about the system being monitored. An event-driven measurement tool, on the other hand, typically provides only a higher-level summary of the system behavior, such as overall counts or average durations. The information supplied both by an event-driven measurement tool and by a tracing tool is exact, though, such as the precise number of times a certain subroutine is executed. In contrast, the information provided by a sampling strategy is statistical in nature. Thus, repeating the same experiment with an event-driven or tracing tool will produce the same results each time whereas the results produced with a sampling tool will vary slightly each time the experiment is performed.

The system resources consumed by the measurement tool itself as it collects data will strongly affect how much perturbation the tool will cause in the system. As mentioned above, the overhead of an event-driven measurement tool is directly proportional to the number of occurrences of the event being measured. Events that occur frequently may cause this type of tool to produce substantial perturbation as a byproduct of the measurement process. The overhead of a sampling-based tool, however, is independent of the number of times any specific event occurs. The perturbation caused by this type of tool is instead a function of the sampling interval, which can be controlled by the experimenter or the tool builder. A trace-based tool consumes the largest amount of system resources, requiring both processor resources (i.e. time) to record each event and potentially enormous amounts of storage resources to save each event in the trace. As a result, tracing tends to produce the largest system perturbation.

Each indirect measurement tool must be uniquely adapted to the particular aspect of the system performance it attempts to measure. Therefore, it is impossible to make any general statements about a measurement tool that makes use of an indirect strategy. The key to implementing a tool to measure a specific performance metric is to match the characteristics of the desired metric with the appropriate measurement strategy. Several of the fundamental techniques that have been used for implementing the various measurement strategies are described in the following sections.

6.2 Interval timers

One of the most fundamental measuring tools in computer-system performance analysis is the *interval timer*. An interval timer is used to measure the execution time of an entire program or any section of code within a program. It can also provide the time basis for a sampling measurement tool. Although interval timers are relatively straightforward to use, understanding how an interval timer is constructed helps the performance analyst determine the limitations inherent in this type of measurement tool.

Interval timers are based on the idea of counting the number of clock pulses that occur between two predefined events. These events are typically identified by inserting calls to a routine that reads the current timer count value into a program at the appropriate points, such as shown previously in the example in Figure 2.1. There are two common implementations of interval timers, one using a hardware counter, and the other based on a software interrupt.

Hardware timers. The hardware-based interval timer shown in Figure 6.1 simply counts the number of pulses it receives at its clock input from a free-running clock source. The counter is typically reset to 0 when the system is first powered up so that the value read from the counter is the number of clock ticks that have occurred since that time. This value is used within a program by reading the memory location that has been mapped to this counter by the manufacturer of the system.

Assume that the value read at the start of the interval being measured is x_1 and the value read at the end of the interval is x_2. Then the total time that has elapsed between these two read operations is $T_e = (x_2 - x_1)T_c$, where T_c is the period of the clock input to the counter.

Software timers. The primary difference between a software-interrupt-based interval timer, shown in Figure 6.2, and a hardware-based timer is that the counter accessible to an application program in the software-based implementa-

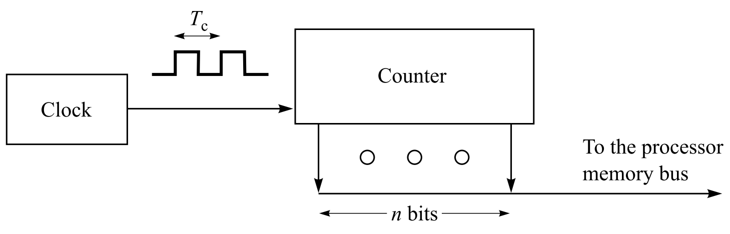


Figure 6.1 A hardware-based interval timer uses a free-running clock source to continuously increment an n-bit counter. This counter can be read directly by the operating system or by an application program. The period of the clock, T_c, determines the resolution of the timer.

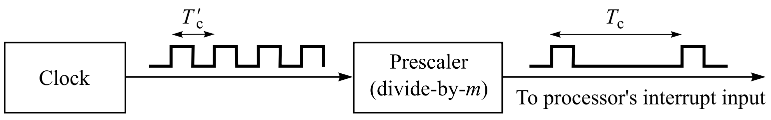


Figure 6.2 A software interrupt-based timer divides down a free-running clock to produce a processor interrupt with the period T_c. The interrupt service routine then maintains a counter variable in memory that it increments each time the interrupt occurs.

tion is not directly incremented by the free-running clock. Instead, the hardware clock is used to generate a processor interrupt at regular intervals. The interrupt-service routine then increments a counter variable it maintains, which is the value actually read by an application program. The value of this variable then is a count of the number of interrupts that have occurred since the count variable was last initialized. Some systems allow an application program to reset this counter. This feature allows the timer to always start from zero when timing the duration of an event.

The period of the interrupts in the software-based approach corresponds to the period of the timer. As before, we denote this period T_c so that the total time elapsed between two readings of the software counter value is again $T_e = (x_2 - x_1)T_c$. The processor interrupt is typically derived from a free-running clock source that is divided by m through a prescaling counter, as shown in Figure 6.2. This prescaler is necessary in order to reduce the frequency of the interrupt signal fed into the processor. Interrupts would occur much too often, and thus would generate a huge amount of processor overhead, if this prescaling were not done.

Timer rollover. One important consideration with these types of interval timers is the number of bits available for counting. This characteristic directly determines the longest interval that can be measured. (The complementary issue of the shortest interval that can be measured is discussed in Section 6.2.2.) A binary counter used in a hardware timer, or the equivalent count variable used in a software implementation, is said to 'roll over' to zero as its count undergoes a transition from its maximum value of $2^n - 1$ to the zero value, where n is the number of bits in the counter.

If the counter rolls over between the reading of the counter at the start of the interval being measured and the reading of the counter at the end, the difference of the count values, $x_2 - x_1$, will be a negative number. This negative value is obviously not a valid measurement of the time interval. Any program that uses an interval timer must take care to ensure that this type of roll over can never occur, or it must detect and, possibly, correct the error. Note that a negative value that occurs due to a single roll over of the counter can be converted to the appropriate value by adding the maximum count value, 2^n, to the negative value obtained when subtracting x_1 from x_2. Table 6.1 shows the maximum time between timer roll overs for various counter widths and input clock periods.

6.2.1 Timer overhead

The implementation of an interval timer on a specific system determines how the timer must be used. In general, though, we can think of using an interval timer to measure any portion of a program, much as we would use a stopwatch to time a runner on a track, for instance. In particular, we typically would use an interval time within a program as follows:

```
x_start = read_timer();
<event being timed>
x_end = read_timer();
elapsed_time = (x_end - x_start) * t_cycle;
```

When it is used in this way, we can see that the time we actually measure includes more than the time required by the event itself. Specifically, accessing the timer requires a minimum of one memory-read operation. In some implementations, reading the timer may require as much as a call to the operating-system kernel, which can be very time-consuming. Additionally, the value read from the timer must be stored somewhere before the event being timed begins. This requires at least one store operation, and, in some systems, it could require substantially more. These operations must be performed twice, once at the start of the event, and once again at the end. Taken altogether, these operations can add up to a significant amount of time relative to the duration of the event itself.

To obtain a better understanding of this timer overhead, consider the time line shown in Figure 6.3. Here, T_1 is the time required to read the value of the interval timer's counter. It may be as short as a single memory read, or as long as a call into the operating-system kernel. Next, T_2 is the time required to store the current time. This time includes any time in the kernel after the counter has been read, which would include, at a minimum, the execution of the return instruction. Time T_3 is the actual duration of the event we are trying to measure. Finally, the time from when the event ends until the program actually reads the counter value again is T_4. Note that reading the counter this second time involves the same set of operations as the first read of the counter so that $T_4 = T_1$.

Assigning these times to each of the components in the timing operation now allows us to compare the timer overhead with the time of the event itself, which is what we actually want to know. This event time, T_e is time T_3 in our time line, so that $T_e = T_3$. What we measure, however, is $T_m = T_2 + T_3 + T_4$. Thus, our

Table 6.1 The maximum time available before a binary interval timer with *n* bits and an input clock with a period of T_c rolls over is $T_c 2^n$

	Counter width, *n*				
T_c	16	24	32	48	64
10 ns	655 μs	168 ms	42.9 s	32.6 days	58.5 centuries
100 ns	6.55 ms	1.68 s	7.16 min	326 days	585 centuries
1 μs	65.5 ms	16.8 s	1.19 h	9.15 years	5,850 centuries
10 μs	655 ms	2.8 min	11.9 h	89.3 years	58,500 centuries
100 μs	6.55 s	28.0 min	4.97 days	893 years	585,000 centuries
1 ms	1.09 min	4.66 h	49.7 days	89.3 centuries	5,850,000 centuries

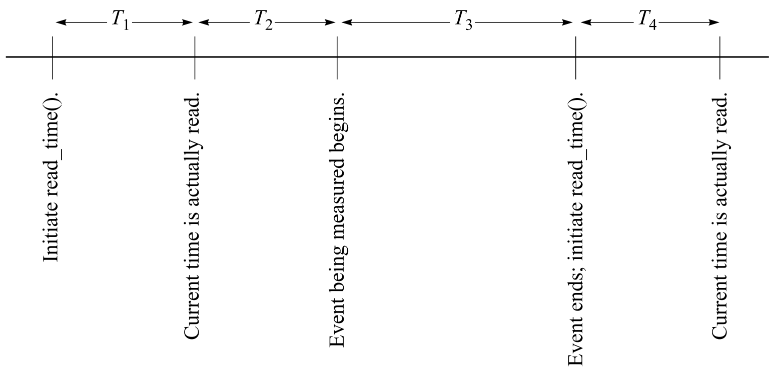

T_1
T_2
T_3
T_4
Initiate read_time().
Current time is actually read.
Event being measured begins.
Event ends; initiate read_time().
Current time is actually read.

Figure 6.3 The overhead incurred when using an interval timer to measure the execution time of any portion of a program can be understood by breaking down the operations necessary to use the timer into the components shown here.

desired measurement is $T_e = T_m - (T_2 + T_4) = T_m - (T_1 + T_2)$, since $T_4 = T_1$. We call $T_1 + T_2$ the *timer overhead* and denote it T_{ovhd}.

If the interval being measured is substantially larger than the timer overhead, then the timer overhead can simply be ignored. If this condition is not satisfied, though, then the timer overhead should be carefully measured and subtracted from the measurement of the event under consideration. It is important to recognize, however, that variations in measurements of the timer overhead itself can often be quite large relative to variations in the times measured for the event. As a result, measurements of intervals whose duration is of the same order of magnitude as the timer overhead should be treated with great suspicion. A good rule of thumb is that the event duration, T_e, should be 100–1,000 times larger than the timer overhead, T_{ovhd}.

6.2.2 Quantization errors

The smallest change that can be be detected and displayed by an interval timer is its *resolution*. This resolution is a single clock tick, which, in terms of time, is the period of the timer's clock input, T_c. This finite resolution introduces a random *quantization error* into all measurements made using the timer.

For instance, consider an event whose duration is n ticks of the clock input, plus a little bit more. That is, $T_e = nT_c + \Delta$, where n is a positive integer and $0 < \Delta < T_c$. If, when one is measuring this event, the timer value is read shortly after the event has actually begun, as shown in Figure 6.4(a), the timer will count n clock ticks before the end of the event. The total execution time reported then will be nT_c. If, on the other hand, there is slightly less time between the actual start of the event and the point at which the timer value is read, as shown in Figure 6.4(b), the timer will count $n + 1$ clock ticks before the end of the event is detected. The total time reported in this case will then be $(n + 1)T_c$.

In general, the actual event time is within the range $nT_c < T_e < (n + 1)T_c$. Thus, the fact that events are typically not exactly whole number factors of the timer's clock period causes the time value reported to be rounded either up or down by one clock period. This rounding is completely unpredictable and is one readily identifiable (albeit possibly small) source of random errors in our measurements (see Section 4.2). Looking at this quantization effect another way, if we made ten measurements of the same event, we would expect that approximately five of them would be reported as nT_c with the remainder reported as $(n + 1)T_c$. If T_c is large relative to the event being measured, this quantization effect can make it impossible to directly measure the duration of the event. Consequently, we typically would like T_c to be as small as possible, within the constraints imposed by the number of bits available in the timer (see Table 6.1).

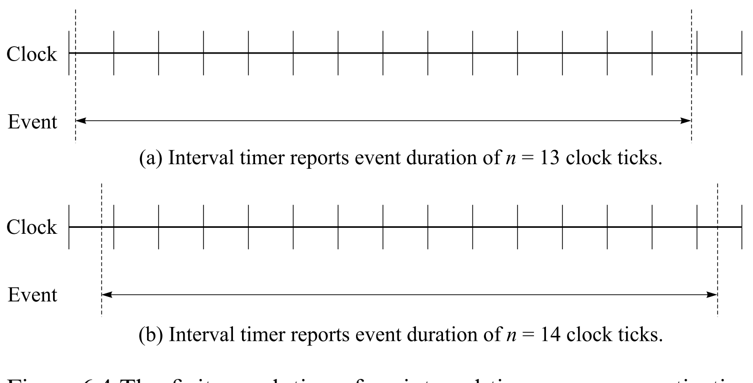

Clock
Event
(a) Interval timer reports event duration of n = 13 clock ticks.
Clock
Event
(b) Interval timer reports event duration of n = 14 clock ticks.

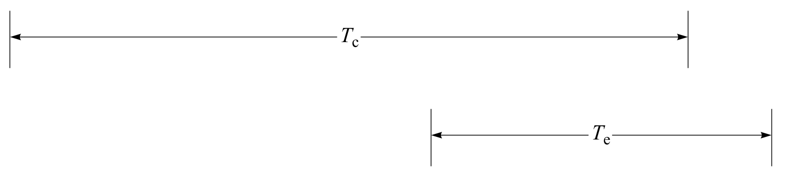
Figure 6.4 The finite resolution of an interval timer causes quantization of the reported duration of the events measured.

6.2.3 Statistical measures of short intervals

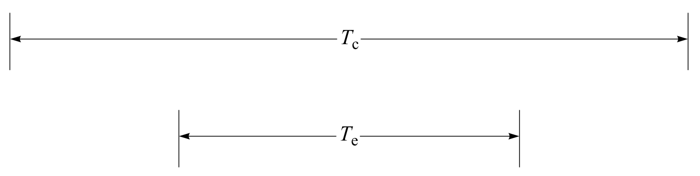
Owing to the above quantization effect, we cannot directly measure events whose durations are less than the resolution of the timer. Similarly, quantization makes it difficult to accurately measure events with durations that are only a few times larger than the timer's resolution. We can, however, make many measurements of a short duration event to obtain a statistical estimate of the event's duration.

Consider an event whose duration is smaller than the timer's resolution, that is, $T_e < T_c$. If we measure this interval once, there are two possible outcomes. If we happen to start our measurement such that the event straddles the active edge of the clock that drives the timer's internal counter, as shown in Figure 6.5(a), we will see the clock advance by one tick. On the other hand, since $T_e < T_c$, it is entirely possible that the event will begin and end within one clock period, as shown in Figure 6.5(b). In this case, the timer will not advance during this measurement. Thus, we have a Bernoulli experiment whose outcome is 1 with probability p, which corresponds to the timer advancing by one tick while are measuring the event. If the clock does not advance, though, the outcome is 0 with probability $1 - p$.

Repeating this measurement n times produces a distribution that approximates a binomial distribution. (It is only approximate since, for a true binomial dis-



(a) Event T_e straddles the active edge of the interval timer.



(b) Event T_e begins and ends within the resolution of the interval timer.

Figure 6.5 When one is measuring an event whose duration is less than the resolution of the interval timer, that is, $T_e < T_c$, there are two possible outcomes for each measurement. Either the event happens to straddle the active edge of the timer's clock, in which case the counter advances by one tick, or the event begins and completes between two clock edges. In the latter case, the interval timer will show the same count value both before and after the event. Measuring this event multiple times approximates a binomial distribution.

tribution, each of the n measurements must be independent. However, in a real system it is possible that obtaining an outcome of 0 in one measurement makes it more likely that one will obtain a 0 in the next measurement, for instance. Nevertheless, this approximation appears to work well in practice.) If the number of outcomes that produce 1 is m, then the ratio m/n should approximate the ratio of the duration of the event being measured to the clock period, T_e/T_c. Thus, we can estimate the average duration of this event to be

$$T_e = \frac{m}{n} T_c. \tag{6.1}$$

We can then use the technique for calculating a confidence interval for a proportion (see Section 4.4.3) to obtain a confidence interval for this average event time.[1]

Example. We wish to measure an event whose duration we suspect is less than the 40 μs resolution of our interval timer. Out of $n = 10{,}482$ measurements of this event, we find that the clock actually advances by one tick during $m = 852$ of them. For a 95% confidence level, we construct the interval for the ratio $m/n = 852/10{,}482$ as follows:

$$(c_1, c_2) = \frac{852}{10{,}482} \mp (1.96)\sqrt{\frac{\frac{852}{10{,}482}\left(1 - \frac{852}{10{,}482}\right)}{10{,}482}} = (0.0786, 0.0840). \tag{6.2}$$

Scaling this interval by the timer's clock period gives us the 95% confidence interval (3.14, 3.36)μs for the duration of this event. ◇

6.3 Program profiling

A *profile* provides an overall view of the execution behavior of an application program. More specifically, it is a measurement of how much time, or the fraction of the total time, the system spends in certain states. A profile of a program can be useful for showing how much time the program spends executing each of its various subroutines, for instance. This type of information is often used by a programmer to identify those portions of the program that consume the largest fraction of the total execution time. Once the largest time consumers have been identified, they can, one assumes, be enhanced to thereby improve performance.

Similarly, when a profile of an entire system multitasking among several different applications is taken, it can be used by a system administrator to find system-level performance bottlenecks. This information can be used in turn to

[1] The basic idea behind this technique was first suggested by Peter H. Danzig and Steve Melvin in an unpublished technical report from the University of Southern California.

tune the performance of the overall system by adjusting such parameters as buffer sizes, time-sharing quanta, disk-access policies, and so forth.

There are two distinct techniques for creating a program profile – program-counter (PC) sampling and basic-block counting. Sampling can also be used to generate a profile of a complete system.

6.3.1 PC sampling

Sampling is a general statistical measurement technique in which a subset (i.e. a sample) of the members of a population being examined is selected at random. The information of interest is then gathered from this subset of the total population. It is assumed that, since the samples were chosen completely at random, the characteristics of the overall population will approximately follow the same proportions as do the characteristics of the subset actually measured. This assumption allows conclusions about the overall population to be drawn on the basis of the complete information obtained from a small subset of this population.

While this traditional population sampling selects all of the samples to be tested at (essentially) the same time, a slightly different approach is required when using sampling to generate a profile of an executing program. Instead of selecting all of the samples to be measured at once, samples of the executing program are taken at fixed points in time. Specifically, an external periodic signal is generated by the system that interrupts the program at fixed intervals. Whenever one of these interrupts is detected, appropriate state information is recorded by the interrupt-service routine.

For instance, when one is generating a profile for a single executing program, the interrupt-service routine examines the return-address stack to find the address of the instruction that was executing when the interrupt occurred. Using symbol-table information previously obtained from the compiler or assembler, this program-instruction address is mapped onto a specific subroutine identifier, i. The value i is used to index into a single-dimensional array, H, to then increment the element H_i by one. In this way, the interrupt-service routine generates a histogram of the number of times each subroutine in the program was being executed when the interrupt occurred.

The ratio H_i/n is the fraction of the program's total execution time that it spent executing in subroutine i, where n is the total number of interrupts that occurred during the program's execution. Multiplying the period of the interrupt by these ratios provides an estimate of the total time spent executing in each subroutine.

It is important to remember that sampling is a statistical process in which the characteristics of an entire population (in our present situation, the execution behavior of an entire program or system) are inferred from a randomly selected subset of the overall population. The calculated values of these inferences are, therefore, subject to random errors. Not surprisingly, we can calculate a confidence interval for these proportions to obtain a feel for the precision of our sampling experiment.

Example. Suppose that we use a sampling tool that interrupts an executing program every $T_c = 10$ ms. Including the time required to execute the interrupt-service routine, the program executes for a total of 8 s. If $H_X = 12$ of the $n = 800$ samples find the program counter somewhere in subroutine X when the interrupt occurred, what is the fraction of the total time the program spends executing this subroutine?

Since there are 800 samples in total, we conclude that the program spends 1.5% ($12/800 = 0.015$) of its time in subroutine X. Using the procedure from Section 4.4.3, we calculate a 99% confidence interval for this proportion to be

$$(c_1, c_2) = 0.015 \mp 2.576\sqrt{\frac{0.015(1-0.015)}{800}} = (0.0039, 0.0261). \tag{6.3}$$

So, with 99% confidence, we estimate that the program spends between 0.39% and 2.6% of its time executing subroutine X. Multiplying by the period of the interrupt, we estimate that, out of the 8 s the program was executing, there is a 99% chance that it spent between 31 (0.0039×8) and 210 (0.0261×8) ms executing subroutine X. ◇

The confidence interval calculated in the above example produces a rather large range of times that the program could be spending in subroutine X. Put in other terms, if we were to repeat this experiment several times, we would expect that, in 99% of the experiments, from three to 21 of the 800 samples would come from subroutine X. While this 7 : 1 range of possible execution times appears large, we estimate that subroutine X still accounts for less than 3% of the total execution time. Thus, we most likely would start our program-tuning efforts on a routine that consumes a much larger fraction of the total execution time.

This example does demonstrate the importance of having a sufficient number of samples in each state to produce reliable information, however. To reduce the size of the confidence interval in this example we need more samples of each event. Obtaining more samples per event requires either sampling for a longer period of time, or increasing the sampling rate. In some situations, we can simply let the program execute for a longer period of time. This will increase the total number of samples and, hence, the number of samples obtained for each subroutine.

Some programs have a fixed duration, however, and cannot be forced to execute for a longer period. In this situation, we can run the program multiple times and simply add the samples from each run. The alternative of increasing the sampling frequency will not always be possible, since the interrupt period is often fixed by the system or the profiling tool itself. Furthermore, increasing the sampling frequency increases the number of times the interrupt-service routine is executed, which increases the perturbation to the program. Of course, each run of the program must be performed under identical conditions. Otherwise, if the test conditions are not identical, we are testing two essentially different systems. Consequently, in this case, the two sets of samples cannot be simply added together to form one larger sample set.

It is also important to note that this sampling procedure implicitly assumes that the interrupt occurs completely asynchronously with respect to any events in the program being profiled. Although the interrupts occur at fixed, predefined intervals, if the program events and the interrupt are asynchronous, the interrupts will occur at random points in the execution of the program being sampled. Thus, the samples taken at these points are completely independent of each other. This sample independence is critical to obtaining accurate results with this technique since any synchronism between the events in the program and the interrupt will cause some areas of the program to be sampled more often than they should, given their actual frequency of occurrence.

6.3.2 Basic-block counting

The sampling technique described above provides a statistical profile of the behavior of a program. An alternative approach is to produce an exact execution profile by counting the number of times each *basic block* is executed. A basic block is a sequence of processor instructions that has no branches into or out of the sequence, as shown in Figure 6.6. Thus, once the first instruction in a block begins executing, it is assured that all of the remaining instructions in the block will be executed. The instructions in a basic block can be thought of as a computation that will always be executed as a single unit.

A program's basic-block structure can be exploited to generate a profile by inserting into each basic block additional instructions. These additional instructions simply count the number of times the block is executed. When the program terminates, these values form a histogram of the frequency of the basic-block executions. Just like the histogram produced with sampling, this basic-block histogram shows which portions of the program are executed most frequently. In this case, though, the resolution of the information is at the basic-block level instead of the subroutine level. Since a basic block executes as an indivisible unit, complete instruction-execution-frequency counts can also be obtained from these basic-block counts.

```
1.   $37:     la       $25, __iob
2.            lw       $15, 0($25)
3.            addu     $9, $15, -1
4.            sw       $9, 0($25)
5.            la       $8, __iob
6.            lw       $11, 0($8)
7.            bge      $11, 0, $38
8.            move     $4, $8
9.            jal      __filbuf
10.           move     $17, $2
11.  $38:     la       $12, __iob
              . . .
```

Figure 6.6 A basic block is a sequence of instructions with no branches into or out of the block. In this example, one basic block begins at statement 1 and ends at statement 7. A second basic block begins at statement 8 and ends at statement 9. Statement 10 is a basic block consisting of only one instruction. Statement 11 begins another basic block since it is the target of an instruction that branches to label $38.

One of the key differences between this basic-block profile and a profile generated through sampling is that the basic-block profile shows the *exact* execution frequencies of all of the instructions executed by a program. The sampling profile, on the other hand, is only a statistical estimate of the frequencies. Hence, if a sampling experiment is run a second time, the precise execution frequences will most likely be at least slightly different. A basic-block profile, however, will produce exactly the same frequencies whenever the program is executed with the same inputs.

Although the repeatability and exact frequencies of basic-block counting would seem to make it the obvious profiling choice over a sampling-based profile, modifying a program to count its basic-block executions can add a substantial amount of run-time overhead. For instance, to instrument a program for basic-block counting would require the addition of at least one instruction to increment the appropriate counter when the block begins executing to each basic block. Since the counters that need to be incremented must be unique for each basic block, it is likely that additional instructions to calculate the appropriate offset for the current block into the array of counters will be necessary.

In most programs, the number of instructions in a basic block is typically between three and 20. Thus, the number of instructions executed by the instrumented program is likely to increase by at least a few percent and possibly as much as 100% compared with the uninstrumented program. These additional instructions can substantially increase the total running time of the program.

Furthermore, the additional memory required to store the counter array, plus the execution of the additional instructions, can cause other substantial perturbations. For instance, these changes to the program can significantly alter its memory behavior.

So, while basic-block counting provides exact profile information, it does so at the expense of substantial overhead. Sampling, on the other hand, distributes its perturbations randomly throughout a program's execution. Also, the total perturbation due to sampling can be controlled somewhat by varying the period of the sampling interrupt interval. Nevertheless, basic-block counting can be a useful tool for precisely characterizing a program's execution profile. Many compilers, in fact, have compile-time flags a user can set to automatically insert appropriate code into a program as it is compiled to generate the desired basic-block counts when it is subsequently executed.

6.4 Event tracing

The information captured through a profiling tool provides a summary picture of the overall execution of a program. An often-useful type of information that is ignored in this type of profile summary, however, is the time-ordering of events. A basic-block-counting profile can show the type and frequency of each of the instructions executed, for instance, but it does not provide any information about the order in which the instructions were executed. When this sequencing information is important to the analysis being performed, a program trace is the appropriate choice.

A *trace* of a program is a dynamic list of the events generated by the program as it executes. The events that comprise a trace can be any events that you can find a way to monitor, such as a time-ordered list of all of the instructions executed by a program, the sequence of memory addresses accessed by a program, the sequence of disk blocks referenced by the file system, the sizes and destinations of all messages sent over a network, and so forth. The level of detail provided in a trace is entirely determined by the performance analyst's ability to gather the information necessary for the problem at hand.

Traces themselves can be analyzed to characterize the overall behavior of a program, much as a profile characterizes a program's behavior. However, traces are probably more typically used as the input to drive a simulator. For instance, traces of the memory addresses referenced by a program are often used to drive cache simulators. Similarly, traces of the messages sent by an application program over a communication network are often used to drive simulators for evaluating changes to communication protocols.

6.4.1 Trace generation

The overall tracing process is shown schematically in Figure 6.7. A tracing system typically consists of two main components. The first is the application being traced, which is the component that actually *generates* the trace. The second main component is the trace *consumer*. This is the program, such as a simulator, that actually uses the information being generated. In between the trace generator and the consumer is often a large disk file on which to store the trace. Storing the trace allows the consumer to be run many times against an unchanging trace to allow comparison experiments without the expense of regenerating the trace. Since the trace can be quite large, however, it will not always be possible or desirable to store the trace on an intermediate disk. In this case, it is possible to consume the trace *online* as it is generated.

A wide range of techniques have been developed for generating traces. Several of these approaches are summarized below.

1. **Source-code modification.** Perhaps the most straightforward approach for generating a program trace is to modify the source code of the program to be traced. For instance, the programmer may add additional tracing statements to the source code, as shown in Figure 6.8. When the program is subsequently compiled and executed, these additional program statements will be executed, thereby generating the desired trace. One advantage of this approach is that the programmer can trace only the desired events. This can help reduce the volume of trace data generated. One major disadvantage is that inserting trace points is typically a manual process and is, therefore, very time-consuming and prone to error.
2. **Software exceptions.** Some processors have been constructed with a mode that forces a software exception just before the execution of each instruction. The

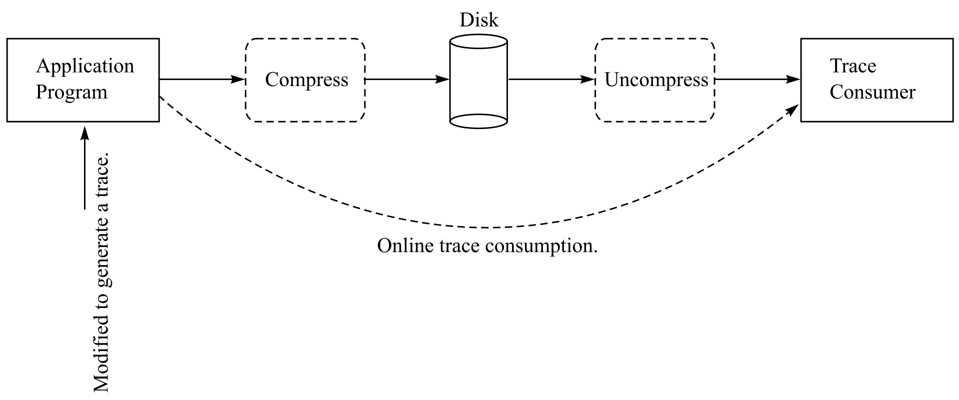


Figure 6.7 The overall process used to generate, store, and consume a program trace.

```
        sum_x = 0.0;
        trace(1);
        sum_xx = 0.0;
        trace(2);
        for (i = 1; i <= n; i++)
        trace(3);
                {
                sum_x += x[i];
                trace(4);
                sum_xx += (x[i]*x[i]);
                trace(5);
                }
        mean = sum_x / n;
        trace(6);
        var = ((n * sum_xx) - (sum_x * sum_x)) / (n * (n-1));
        trace(7);
        std_dev = sqrt(var);
        trace(8);
        z_p = unit_normal(1 - (0.5 * alpha));
        trace(9);
        half_int = z_p * std_dev / sqrt(n);
        trace(10);
        c1 = mean - half_int;
        trace(11);
        c2 = mean + half_int;
            trace(12);
```

(a) The original source program with calls to the tracing routine inserted.

```
trace(i)
{ print(i,time);}
```

(b) The trace routine simply prints the statement number, i, and the current time.

Figure 6.8 Program tracing can be performed by inserting additional statements into the source code to call a tracing subroutine at appropriate points.

exception-processing routine can decode the instruction to determine its operands. The instruction type, address, and operand addresses and values can then be stored for later use. This approach was implemented using the T-bit in Digital Equipment Corporation's VAX processor series and in the Motorola 68000 processor family. Executing with the trace mode enabled on these processors slowed down a program's execution by a factor of about 1,000.

3. **Emulation.** An emulator is a program that makes the system on which it executes appear to the outside world as if it were something completely different. For example, the Java Virtual Machine is a program that executes application programs written in the Java programming language by emulating the operation of a processor that implements the Java byte-code instruction set. This emulation obviously slows down the execution of the application program compared with direct execution. Conceptually, however, it is a straightforward task to modify the emulator program to trace the execution of any application program it executes.
4. **Microcode modification.** In the days when processors executed microcode to execute their instruction sets through interpretation, it was possible to modify the microcode to generate a trace of each instruction executed. One important advantage of this approach was that it traced every instruction executed on the processor, including operating-system code. This feature was especially useful for tracing entire systems, including the interaction between the application programs and the operating system. The lack of microcode on current processors severely limits the applicability of this approach today.
5. **Compiler modification.** Another approach for generating traces is to modify the executable code produced by the compiler. Similar to what must be done for generating basic-block counts, extra instructions are added at the start of each basic block to record when the block is entered and which basic block is being executed then. Details about the contents of the basic blocks can be obtained from the compiler and correlated to the dynamic basic-block trace to produce a complete trace of all of the instructions executed by the application program. It is possible to add this type of tracing facility as a compilation option, or to write a post-compilation software tool that modifies the executable program generated by the compiler.

These trace-generation techniques are by no means the only ways in which traces can be produced. Rather, they are intended to give you a flavor of the types of approaches that have been used successfully in other trace-generation systems. Indeed, new techniques are limited only by the imagination and creativity of the performance analyst.

6.4.2 Trace compression

One obvious concern when generating a trace is the execution-time slowdown and other program perturbations caused by the execution of the additional tracing instructions. Another concern is the volume of data that can be produced in a very short time. For example, say we wish to trace every instruction executed by a processor that executes at an average rate of 10^8 instructions per second. If each item in the trace requires 16 bits to encode the necessary information, our tracing will produce more than 190 Mbytes of data per uninstrumented second of execution time, or more than 11 Gbytes per minute! In addition to obtaining the disks necessary to store this amount of data, the input/output operations required to move this large volume of data from the traced program to the disks create additional perturbations. Thus, it is desirable to reduce the amount of information that must be stored.

6.4.2.1 Online trace consumption

One approach for dealing with these large data volumes is to consume the trace *online*. That is, instead of storing the trace for later use, the program that will be driven by the trace is run simultaneously with the application program being traced. In this way, the trace is consumed as it is generated so that it never needs to be stored on disk at all.

A potential problem with online trace consumption in a multitasked (i.e. time-shared) system is the potential interdeterminate behavior of the program being traced. Since system events occur asynchronously with respect to the traced program, there is no assurance that the next time the program is traced the exact same sequence of events will occur in the same relative time order. This is a particular concern for programs that must respond to real-time events, such as system interrupts and user inputs.

This potential lack of repeatability in generating the trace is a concern when performing one-to-one comparison experiments. In this situation, the trace-consumption program is driven once with the trace and its output values are recorded. It is then modified in some way and then driven again with the same trace. If the identical input trace is used both times, it is reasonable to conclude that any change in performance observed is due to the change made to the trace-consumption program. However, if it cannot be guaranteed that the trace is identical from one run to the next, it is not possible to determine whether any change in performance observed is due to the change made, or whether it is due to a difference in the input trace itself.

6.4.2.2 Compression of data

A trace written to intermediate storage, such as a disk, can be viewed just like any other type of data file. Consequently, it is quite reasonable to apply a data-compression algorithm to the trace data as it is written to the disk. For example, any one of the large number of compression programs based on the popular Lempel–Ziv algorithm is often able to reduce the size of a trace file by 20–70%. Of course, the tradeoff for this data compression is the additional time required to execute the compression routine when the trace is generated and the time required to uncompress the trace when it is consumed.

6.4.2.3 Abstract execution

An interesting variation of the basic trace-compression idea takes advantage of the semantic information within a program to reduce the amount of information that must be stored for a trace. This approach, called *abstract execution*, separates the tracing process into two steps. The first step performs a compiler-style analysis of the program to be traced. This analysis identifies a small subset of the entire trace that is sufficient to later reproduce the full trace. Only this smaller subset is actually stored. Later, the trace-consumption program must execute some special trace-regeneration routines to convert this partial trace information into the full trace. These regeneration routines are automatically generated by the tracing tool when it performs the initial analysis of the program.

The data about the full trace that are actually stored when using the abstract-execution model consist of information describing only those transitions that may change during run-time. For example, consider the code fragment extracted from a program to be traced shown in Figure 6.9. The compiler-style analysis that would be performed on this code fragment would produce the control flow graph shown in Figure 6.10. From this control flow graph, the trace-generation tool can determine that statement 1 always precedes both statements 2 and 3. Furthermore, statement 4 always follows both statements 2 and 3. When this program is executed, the trace through this sequence of statements will be either 1–2–4, or 1–3–4. Thus, the only information that needs to be recorded during run-time is which of statements 2 and 3 actually occurred. The trace-regeneration routine is then able to later reconstruct the full trace using the previously recorded control flow graph.

Measurements of the effectiveness of this tracing technique have shown that it slows down the execution of the program being traced by a factor of typically 2–10. This slowdown factor is comparable to, or slightly better than, those of most other tracing techniques. More important, however, may be that, by recording information only about the changes that actually occur during run-time, this technique is able to reduce the size of the stored traces by a factor of ten to several hundred.

```
1. if (i > 5)
2.     then a = a + i;
3.     else b = b + 1;
4. i = i + 1;
```

Figure 6.9 A code fragment to be processed using the abstract execution tracing technique..

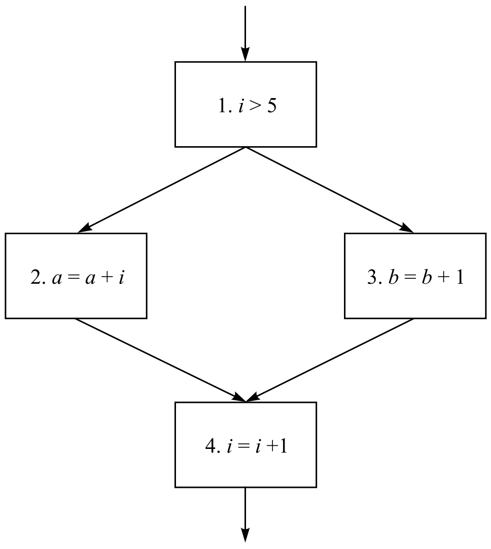


Figure 6.10 The control flow graph corresponding to the program fragment shown in Figure 6.9.

6.4.2.4 Trace sampling

Trace sampling is another approach that has been suggested for reducing the amount of information that must be collected and stored when tracing a program. The basic idea is to save only relatively small sequences of events from locations scattered throughout the trace. The expectation is that these small samples will be statistically representative of the entire program's trace when they are used. For instance, using these samples to drive a simulation should produce overall results that are similar to what would be produced if the simulation were to be driven with the entire trace.

Consider the sequence of events from a trace shown in Figure 6.11. Each sample from this trace consists of k consecutive events. The number of events between the starts of consecutive samples is the *sampling interval*, denoted by P. Since only the samples from the trace are actually recorded, the total amount of storage required for the trace can be reduced substantially compared with storing the entire raw trace.

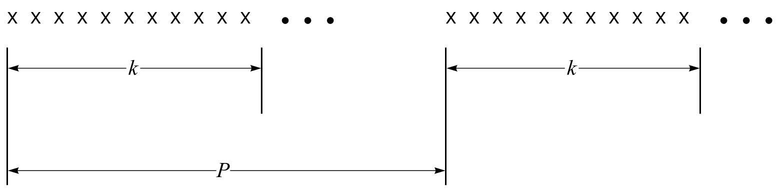


Figure 6.11 In trace sampling, k consecutive events comprise one sample of the trace. A new sample is taken every P events (P is called the sampling interval).

Unfortunately, there is no solid theoretical basis to help the experimenter determine how many events should be stored for each sample (k), or how large the sampling interval (P) should be. The best choices for k and P typically must be determined empirically (i.e. through experimentation). Furthermore, the choice of these parameters seems to be dependent on how the traces will be used. If the traces are used to drive a simulation of a cache to estimate cache-miss ratios, for instance, it has been suggested (see Laha *et al.* (1988)) that, in a trace of tens of millions of memory references, it is adequate to have several thousand events per sample. The corresponding sampling interval then should be chosen to provide enough samples such that 5–10% of the entire trace is recorded. These results, however, appear to be somewhat dependent on the size of the cache being simulated. The bottom line is that, while trace sampling appears to be a reasonable technique for reducing the size of the trace that must be stored, a solid theoretical basis still needs to be developed before it can be considered 'standard practice.'

6.5 Indirect and *ad hoc* measurements

Sometimes the performance metric we need is difficult, if not impossible, to measure directly. In this case, we have to rely on our ingenuity to develop an *ad hoc* technique to somehow derive the information indirectly. For instance, perhaps we are not able to directly measure the desired quantity, but we may be able to measure another related value directly. We may then be able to deduce the desired value from these other measured values.

For example, suppose that we wish to determine how much load a particular application program puts on a system when it is executed. We then may want to make changes to the program to see how they affect the system load. The first question we need to confront in this experiment is that of establishing a definition for the 'system load.'

There are many possible definitions of the system load, such as the number of jobs on the run queue waiting to be executed, to name but one. In our case, however, we are interested in how much of the processor's available time is spent executing our application program. Thus, we decide to define the average system load to be the fraction of time that the processor is busy executing users' application programs.

If we had access to the source code of the operating system, we could directly measure this time by modifying the process scheduler. However, it is unlikely that we will have access to this code. An alternative approach is to directly measure how much time the processor spends executing an 'idle' process that we create. We then use this direct measurement of idle time to deduce how much time the processor must have been busy executing real application programs during the given measurement interval.

Specifically, consider an ‘idle’ program that simply counts up from zero for a fixed period of time. If this program is the only application running on a single processor of a time-shared system, the final count value at the end of the measurement interval is the value that indirectly corresponds to an unloaded processor. If two applications are executed simultaneously and evenly share the processor, however, the processor will run our idle measurement program half as often as when it was the only application running. Consequently, if we allow both programs to run for the same time interval as when we ran the idle program by itself, its total count value at the end of the interval should be half of the value observed when only a single copy was executed.

Similarly, if three applications are executed simultaneously and equally share the processor for the same measurement interval, the final count value in our idle program should be one-third of the value observed when it was executed by itself. This line of thought can be further extended to n application programs simultaneously sharing the processor. After calibrating the counter process by running it by itself on an otherwise unloaded system, it can be used to indirectly measure the system load.

Example. In a time-shared system, the operating system will share a single processor evenly among all of the jobs executing in the system. Each available job is allowed to run for the *time slice* T_s. After this interval, the currently executing job is temporarily put to sleep, and the next ready job is switched in to run. Indirect load monitoring takes advantage of this behavior to estimate the system load. Initially, the load-monitor program is calibrated by allowing it to run by itself for a time T, as shown in Figure 6.12(a). At the end of this time, its counter value, n, is recorded. If the load monitor and another application are run simultaneously so that in total two jobs are sharing the processor, as shown in Figure 6.12(b), each job would be expected to be executing for half of the total time available. Thus, if the load monitor is again allowed to run for time T, we would expect its final count value to be $n/2$. Similarly, running the load monitor with two other applications for time T would result in a final count value of $n/3$, as shown in Figure 6.12(c). Consequently, knowing the value of the count after running the load monitor for time T allows us to deduce what the average load during the measurement interval must have been ◇

6.6 Perturbations due to measuring

One of the curious (and certainly most annoying!) aspects of developing tools to measure computer-systems performance is that instrumenting a system or pro-

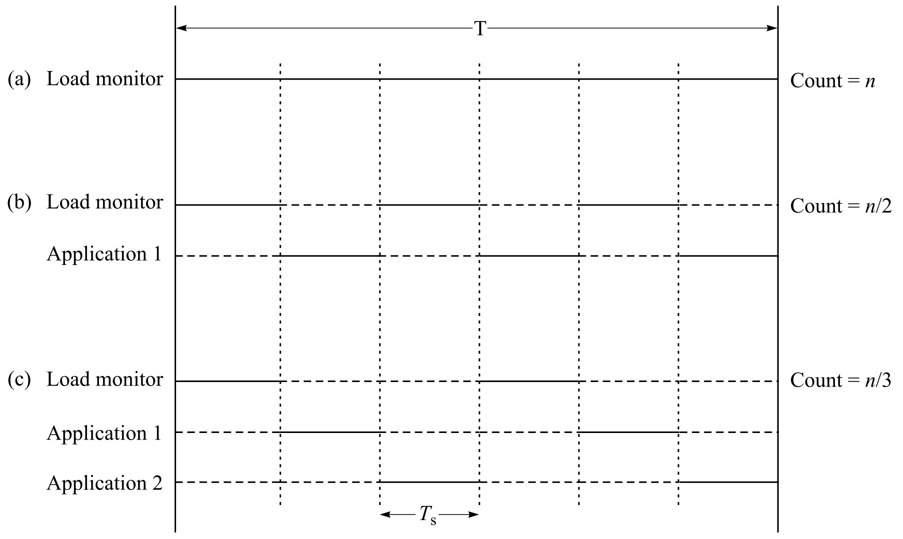


Figure 6.12 An example of using an indirect measurement technique to estimate the average system load in a time-shared system. The solid lines indicate when each application is running.

gram changes what we are trying to measure. Obtaining more information, or obtaining higher resolution measurements, for instance, requires more instrumentation points in a program. However, more instrumentation causes there to be more perturbations in the program than there are in its uninstrumented execution behavior. These additional perturbations due to the additional instrumentation then make the data we collect less reliable. As a result, we are almost always forced to use insufficient data to infer the behavior of the system in which we are interested.

To further confound the situation, performance perturbations due to instrumentation are nonlinear and nonadditive. They are nonlinear in the sense that doubling the amount of instrumentation in a program will not necessarily double its impact on performance, for instance. Similarly, instrumentation perturbation is nonadditive in the sense that adding more instrumentation can cancel out the perturbation effects of other instrumentation. Or, in some situations, additional instrumentation can multiplicatively increase the perturbations.

For example, adding code to an application program to generate an instruction trace can significantly change the spatial and temporal patterns of its memory accesses. The trace-generation code will cause a large number of extra store instructions to be executed, for instance, which can cause the cache to be effectively flushed at each trace point. These frequent cache flushes will then increase the number of caches missed, which will substantially impact the overall performance. If additional instrumentation is added, however, it may be possible that the additional memory locations necessary for the instrumentation could change the pattern of conflict misses in the cache in such a way as to actually improve the cache performance perceived by the application. The bottom line is that the effects of adding instrumentation to a system being tested are entirely unpredictable.

Besides these direct changes to a program's performance, instrumenting a program can cause more subtle indirect perturbations. For example, an instrumented program will take longer to execute than will the uninstrumented program. This increase in execution time will then cause it to experience more context switches than it would have experienced if it had not been instrumented. These additional context switches can substantially alter the program's paging behavior, for instance, making the instrumented program behave substantially differently than the uninstrumented program.

6.7 Summary

Event-driven measurement tools record information about the system being tested whenever some predefined event occurs, such as a page fault or a network operation, for instance. The information recorded may be a simple count of the number of times the event occurred, or it may be a portion of the system's state at the time the event occurred. A time-ordered list of this recorded state information is called a trace. While event-driven tools record all occurrences of the defined events, sampling tools query some aspect of the system's state at fixed time intervals. Since this sampling approach will not record every event, it provides a statistical view of the system. Indirect measurement tools are used to deduce some aspect of a system's performance that it is difficult or impossible to measure directly.

Some perturbation of a system's behavior due to instrumentation is unavoidable. Furthermore, and more difficult to compensate for, perhaps, is the unpredictable relationship between the instrumentation and its impact on performance. Through experience and creative use of measurement techniques, the performance analyst can try to minimize the impact of these perturbations, or can sometimes compensate for their effects.

It is important to bear in mind, though, that measuring a system alters it. While you would like to measure a completely uninstrumented program, what you actually end up measuring is the instrumented system. Consequently, you must always remain alert to how these perturbations may bias your measurements and, ultimately, the conclusions you are able to draw from your experiments.

6.8 For further reading

There is an extensive body of literature dealing with program tracing and a very large variety of tools has been developed. Although the following references only begin to scratch the surface of this field, they should provide you with some useful starting points.

- The Lempel–Ziv data-compression algorithm, on which many data compression programs have been based, is described in

  Terry A. Welch, 'A Technique for High Performance Data Compression,' *IEEE Computer*, Vol. 17, No. 6, June 1984, pp. 8–19.
- The abstract-execution idea, which was developed by James Larus, is described in the following papers, along with some related ideas. These papers also provide a good summary of the program-tracing process in general.

  James R. Larus, 'Efficient Program Tracing,' *IEEE Computer*, Vol. 26, No. 5, May 1993, pp. 52–61.

  James R. Larus, 'Abstract Execution: A Technique for Efficiently Tracing Programs,' *Software Practices and Experience,* Vol. 20, No. 12, December 1990, pp 1241–1258.

  Thomas Ball and James R. Larus, 'Optimally Profiling and Tracing Programs,' *ACM SIGPLAN-SIGACT Principles of Programming Languages (POPL)*, January 1992, pp. 59–70.
- This paper talks about some of the problems encountered when trying to trace applications running on multiprocessor systems, and describes the various types of perturbations that can occur due to tracing.

  Allen D. Malony and Daniel A. Reed, 'Performance Measurement Intrusion and Perturbation Analysis,' *IEEE Transactions on Parallel Distributed Systems*, Vol. 3, No. 4, July 1992, pp. 433–450.
- Paradyn is an interesting set of performance tools for parallel- and distributed-computing systems. The following paper provides a good overview of these tools:

  Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. 'The Paradyn Parallel Performance Measurement Tools,' *IEEE Computer*, Vol. 28, No. 11, November 1995, pp. 37–46.
- The idea behind the indirect-load-measurement technique was presented in

  Edward D. Lazowska, John Zahorjan, David R. Cheriton, and Willy Zwaenepoel, 'File Access Performance of Diskless Workstations,' *IEEE Transactions on Software Engineering*, Vol. 4, No. 3, August 1986, pp. 238–268.

- The SimOS tool, described in the following paper, is an interesting example of how to trace an entire computer system, including both the application program and the operating system:

  Mendel Rosenblum, Stephen A. Herrod, Emmett Witchel, and Anoop Gupta, 'Complete Computer Simulation: The SimOS Approach,' *IEEE Parallel and Distributed Technology*, Fall 1995.
- The idea of sampling traces to reduce the amount of trace information that must be collected and stored is described in

  Subhasis Laha, Janak H. Patel, and Ravishankar K. Iyer, 'Accurate Low-Cost Methods for Performance Evaluation of Cache Memory Systems,' *IEEE Transactions on Computers*, Vol. 37, No. 11, November 1998, pp. 1325–1336.

6.9 Exercises

1. Determine the maximum time between rollovers for the interval timer available on your system.
2. What are the most important differences between tracing and basic-block counting?
3. Develop a technique for measuring the time a processor spends waiting for input/output requests.
4. Develop a technique for determining the associativity of a cache.
5. Measure the overhead of the interval timer on your system.
6. How would you measure the average number of jobs running on a time-shared system?
7. What interrupt period is needed to ensure that each of the 12 subroutines of a program that runs for 30 s has a 99% chance of having at least ten samples? Assume that each subroutine executes for at least 5% of the total time.
8. Use a program counter-sampling tool to compare the differences in performance between two versions of some appropriate benchmark program. Repeat your comparisons using a basic-block-counting tool. Compare and contrast the results you obtain when using these two different types of tools to profile the execution of this program. For instance, what are the fundamental differences between the techniques used by the basic-block-counting tool and those used by the sampling tool? How do the differences between these two tools affect your comparisons of the two versions of the benchmark program?
9. Compare the time penalties and the storage requirements of the various trace-compression techniques.

10. Are there any pathological situations in which these trace compression techniques can backfire and actually expand the input data set?
11. Devise an experiment to determine the following parameters of a computer system with a data cache:
    (a) the memory delay observed by the processor on a data-cache hit, and
    (b) the memory delay observed by the processor on a data-cache miss.

    Then

    (c) construct a simple model of the average memory delay observed by a program, given its hit-or-miss ratio. Use the parameters you measured above.
12. Write a test program that you can use to control the miss-ratio obtained in a system with a data cache. Use this program to validate the model of the average memory delay developed above. That is, measure the execution time of your test program and compare it with the time predicted by your model. What simplifications and approximations are you implicitly making? How could you improve your model or your test program? *Hint*: think about measuring the time required to scan through a large array with a fixed stride (the stride is the number of elements between successive references to the array – a stride of one accesses every element sequentially, a stride of two accesses every second element, a stride of three accesses every third element, and so on). By varying the stride, you should be able to determine the cache-block size. Then, knowing the block size, you can determine the miss ratio.
13. Section 6.5 discussed a technique for indirectly measuring the system load.
    (a) Write a program to perform this counting process.
    (b) Calibrate your counter by running two copies of it simultaneously. Show the results of your calibration with appropriate confidence intervals.
    (c) Use your counter process to determine how the load on a system varies over the course of day on a large time-shared system. For instance, you might try measuring the system load for 1 min every hour on each of several different days. Plot this system load as a function of time. Include appropriate error bars for each of the data points on your plot to give an indication of the variance in your measurements. (These error bars are simply the end-points of the confidence interval for each measured data point. Note that you must repeat the experiment several times to obtain enough independent measurements to generate a confidence interval.)