Chapter 6
Measurement Techniques

> "When you only have a hammer, every problem begins to resemble a nail."
> —*Abraham Maslow*

In the previous chapters, we introduced the most common statistics that can be used to summarize measurements, that is, indices of central tendency and indices of dispersion, providing a basis for defining metrics as part of benchmarks. Furthermore, the statistical approaches for quantifying the variability and precision of measurements were introduced. This chapter looks at the different measurement techniques that can be used in practice to derive the values of common metrics. While most presented techniques are useful for performance metrics, some of them can also be applied generally for other types of metrics.

The chapter starts with a brief introduction to the basic measurement strategies, including event-driven, tracing, sampling, and indirect measurement. We then look at interval timers, which are typically used to measure the execution time of a program or a portion of it. Next, we introduce performance profiling, which provides means to measure how much time a system spends in different states. A performance profile provides a high-level summary of the execution behavior of an application or a system; however, this summary does not provide any information about the order in which events occur. Thus, at the end of the chapter, event-driven tracing strategies are introduced, which can be used to capture such information. We focus on call path tracing—a technique for extracting a control flow graph of an application. Finally, the chapter is wrapped up with an overview of commercial and open-source monitoring tools for performance profiling and call path tracing.

6.1 Basic Measurement Strategies

Measurement techniques are typically based on monitoring changes in the system state. Each change in the system state that is relevant for the measurement of a given

S. Kounev et al., *Systems Benchmarking*, https://doi.org/10.1007/978-3-030-41705-5_6

metric is referred to as an *event*. For example, an event could be a request arrival, a remote procedure call, a processor interrupt, a memory reference, a network access, a failure of a given system component, a rolled back database transaction, a detected denial of service attack, or a security breach. Four fundamental measurement strategies are distinguished: event-driven, tracing, sampling, and indirect measurement (Lilja, 2000).

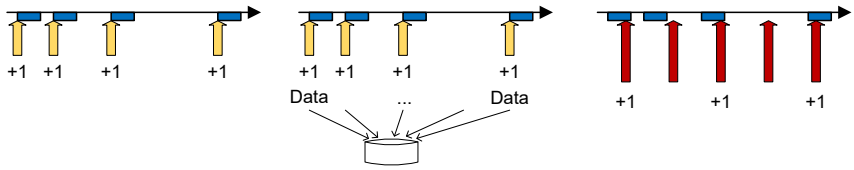
Event-driven strategies record information required to derive a given metric only when specified events of interest occur. The system may have to be instrumented to monitor the respective events and record relevant information. The term *instrumentation*, in this context, refers to the insertion of the so-called monitoring hooks in the code that observe and record relevant information about the events of interest. For example, counting the number of random disk accesses during the execution of a benchmark can be implemented by incrementing a counter in the respective I/O interrupt handling routine of the operating system and dumping the value of the counter at the end of the benchmark execution.

One important aspect of measurement strategies is how much *overhead* they introduce. The measurement overhead may or may not intrude upon the system behavior, and if it does, such intrusion may lead to a change in the observed behavior, a phenomenon often referred to as *perturbation*. The overhead of an event-driven strategy depends on the frequency of the events being monitored. If the events of interest occur very frequently, the overhead may be significant possibly leading to perturbation. In that case, the behavior of the system under test may change and no longer be representative of the typical or average behavior. Therefore, event-driven strategies are usually considered for events with low to moderate frequency.

Tracing strategies are similar to event-driven strategies; however, in addition to counting how often events of interest occur, they record further information about each event (e.g., information on the system state at the time of the event) required to derive a given metric of interest. For example, in addition to observing each random disk access, one may be interested in the specific files accessed and whether data is read or written. Depending on how much information is stored, tracing may introduce significant overhead. Moreover, the time required to store the additional information may significantly alter the behavior of the system under test.

Sampling strategies record relevant information about the system state in equidistant time intervals. The advantage of such strategies is that the overhead they introduce is independent of the frequency with which the respective events of interest occur. Instead, the overhead depends on the sampling frequency, which can be configured by the user. In contrast to the previous two strategies, sampling strategies do not observe every occurrence of the events of interest. They observe only a statistical sample of the execution behavior, which means that infrequent events may be completely missed. Thus, the sampling frequency should be configured to have the resolution necessary to obtain a representative sample of the events of interest. Given that only a statistical sample of the execution is observed, repetitive sampling-based measurements may produce different results. Sampling strategies are typically used for high frequency events where exact event counts are not required and a statistical summary is enough.

Figure 6.1 illustrates the three measurement strategies considered so far.



(a) Event-driven strategies (b) Tracing (c) Sampling

Fig. 6.1: Measurement techniques and strategies

Indirect measurement strategies are used in cases where the metric of interest cannot be measured directly by observing certain events. In such cases, other metrics that can be measured directly are used to derive or deduce the metric of interest. For example, based on the service demand law (see Chapter 7, Section 7.1.2), the service demand of requests at a given resource can be derived from measured throughput and utilization data.

6.2 Interval Timers

An *interval timer* is a tool for measuring the duration of any activity during the execution of a program. Interval timers are typically used in performance measurements to measure the execution time of a program or a portion of it. Most interval timers are implemented by using a counter variable incremented on each tick of a system clock. Interval timers are based on counting the number of ticks between the beginning and end of the respective activity whose duration needs to be measured. The clock ticks are counted by observing the counter variable at the respective points in the program execution (Figure 6.2).

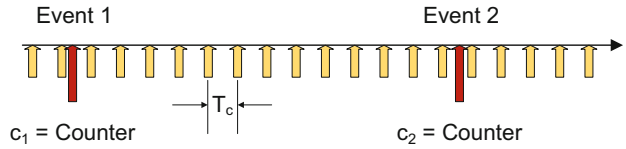


Fig. 6.2: Interval timers

More specifically, based on how an interval timer is implemented, we distinguish between *hardware timers* and *software timers* (Lilja, 2000). In hardware timers, the counter variable is incremented directly by a free-running hardware clock. The

counter is typically set to 0 when the system is powered up and its value shows the number of clock ticks that have occurred since then. In software-based timers, the counter variable is not directly incremented; instead, the hardware clock periodically generates a processor interrupt, and the respective interrupt-service routine triggers a process to increment the counter variable accessible to application programs. Depending on the timer implementation, the process of accessing and updating the counter variable may span several software layers (e.g., operating system, virtual machine, middleware).

Denote with T_c the period of time between two updates of the counter variable, referred to as *clock period* or *resolution* of the timer. If c_1 and c_2 are the values of the counter at the beginning and end of the activity whose duration needs to be measured, then the duration reported by the timer is $(c_2 - c_1)T_c$.

6.2.1 Timer Rollover

An important characteristic of an interval timer is the number of bits available for the counting variable, which determines the longest interval that can be measured using the timer. An n bit counter can store values between 0 and $(2^n - 1)$. Table 6.1 shows the longest interval that can be measured for different values of the resolution T_c and the counter size n.

A timer's counter variable is said to "roll over" to zero when its value transitions from the maximum value $(2^n - 1)$ to 0. If a timer's counter rolls over during an activity whose duration is being measured using the timer, then the value $(c_2 - c_1)T_c$ reported by the timer will be negative. Therefore, applications that use a timer must either ensure that roll over can never occur when using the timer or they should detect and correct invalid measurements caused by roll over.

Table 6.1: Length of time until timer rollover (Lilja, 2000)

Resolution (T_c)	Counter size in bits (n)		
	16	32	64
10 ns	655 μs	43 s	58.5 centuries
1 μs	65.5 ms	1.2 h	5,580 centuries
100 μs	6.55 s	5 days	585,000 centuries
1 ms	1.1 min	50 days	5,850,000 centuries

6.2.2 Timer Accuracy

The accuracy of measurements obtained through an interval timer generally depends on two factors: the timer resolution and the timer overhead.

The timer resolution T_c is the smallest time duration that can be detected by the timer. Given that the timer resolution is finite, there is a random *quantization error* in all measurements made using the timer (Lilja, 2000). This is illustrated in Figure 6.3, which shows an example of an interval timer reporting different duration of the same activity (e.g., execution of an operation with a fixed execution time) depending on its exact starting point. Repeated measurements of the same activity duration will lead to values $X \pm \Delta$. This quantization effect was already discussed in Chapter 4 (Section 4.2.1) in the context of random measurement errors.

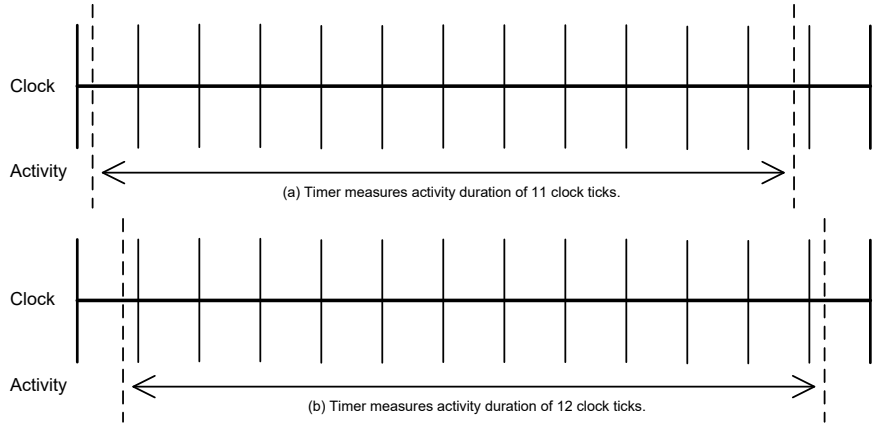


Fig. 6.3: Example of random quantization errors in timer measurements

Given that it is unlikely that the actual activity duration T_a is exactly a whole number factor of the timer's clock period, T_a will normally lie within the range $nT_c < T_a < (n + 1)T_c$, where T_c is the timer's clock period. Thus, the measured duration T_m reported by the timer will be the actual duration T_a rounded up or down by one clock period. The rounding is completely unpredictable, introducing random quantization errors into all measurements reported by the timer. The smaller the timer's clock period, the more accurate its measurements will be.

The second factor that affects the accuracy of a timer is its overhead. An interval timer is typically used like a stopwatch to measure the duration of a given activity. For example, the following pseudocode illustrates how a timer can be used within a program to measure the execution time of a critical section in a program:[1]

[1] A critical section is a section of code that accesses a shared resource (data structure or device) that must not be concurrently accessed by more than one thread of execution. Critical sections must be executed serially; that is, only one thread can execute a critical section at any given time.

```
time_start = read_timer();
<critical section to be measured>
time_end = read_timer();
elapsed_time = (time_end - time_start) * clock_period;
```

Figure 6.4 shows an exemplary time line of the execution. As we can see from the figure, the time we actually measure includes more than the duration of the critical section of which we are interested. This is because accessing the timer normally requires a minimum of one memory-read operation to read the value of the timer and one memory-write operation to store the read value. These operations must be performed at the beginning and end of the measured activity.

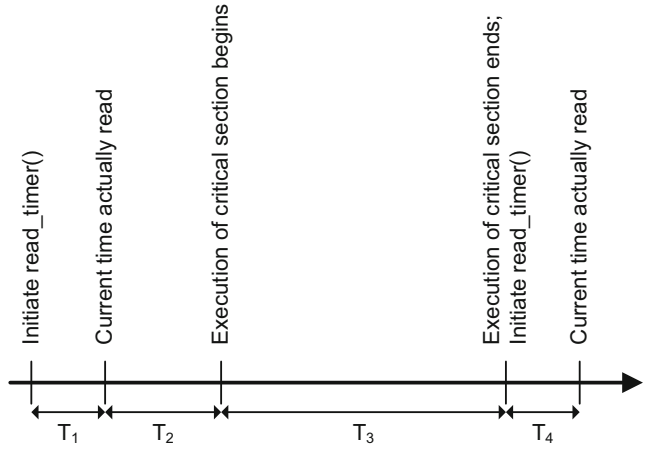


Fig. 6.4: Timer overhead

In Figure 6.4, T_1 and T_4 represent the time required to read the value of the timer's counter variable, whereas T_2 represents the time required to store the value that was obtained. The actual duration of the event we are trying to measure is given by $T_a = T_3$. However, due to the delays accessing the timer, we end up measuring $T_m = T_2 + T_3 + T_4$ instead. Thus, $T_a = T_m - (T_2 + T_4) = T_m - (T_1 + T_2)$, since $T_4 = T_1$. The value of $T_o = T_1 + T_2$ is referred to as *timer overhead* (Lilja, 2000).

If the activity being measured has a duration significantly higher than the timer overhead ($T_a >> T_o$), then the latter can simply be ignored. Otherwise, the timer overhead should be estimated and subtracted from the measurements. However, estimating the timer overhead may be challenging given that it often exhibits high variability in repeated measurements. We refer the reader to Kuperberg, Krogmann, et al. (2009) for a platform-independent method to quantify the accuracy and overhead of a timer without inspecting its implementation.

Generally, measurements of intervals with duration of the same order of magnitude as the timer overhead are not reliable. A rule of thumb is that for timer

measurements to be reliable, the duration of the activity being measured should be 100–1,000 times larger than the timer overhead.

Different timer implementations exhibit different overhead. Evaluating the quality of a given interval timer involves analyzing several properties, such as accuracy, overhead, and timer stability, all of which are platform-dependent. A composite metric, coupled with a benchmarking approach for evaluating the quality of timers, can be found in Kuperberg and Reussner (2011).

6.2.3 Measuring Short Intervals

Given that a timer's clock period T_c is the shortest time interval it can detect, the question arises how a timer can be used to measure the duration of intervals shorter than T_c. More generally, the quantization effect makes it hard to measure intervals that are not significantly larger than the timer's resolution.

In Chapter 4, Section 4.2.3.3, we presented an indirect measurement approach to estimate the execution time of a very short operation (shorter than the clock period of the used interval timer). The idea was to measure the total time for several consecutive repetitions of the operation and divide this time by the number of repetitions to calculate the mean time for one execution. We assume that the number of repetitions is chosen high enough, such that the resolution of the used interval timer can measure the cumulative times. By repeating this procedure n times, we obtain a sample of estimated times for one operation execution and can use this to derive a confidence interval for the mean execution time of the operation. As discussed in Chapter 4, Section 4.2.3.3, while this approach provides a workaround for the issue, the normalization has a penalty. On the one hand, the sample size is reduced, leading to loss of information. On the other hand, we obtain a confidence interval for the mean value of the aggregated operations, as opposed to the individual operations themselves. This leads to reducing the variation and thus the resulting confidence interval might be more narrow than it would have been if applied to the measured duration of single executions.

We now present an alternative approach to measure short intervals (Lilja, 2000). Assume that we would like to measure an interval of size T_a using a timer with a resolution (i.e., clock period) $T_c > T_a$. Short intervals are hard to measure even in cases where $T_a \approx n \times T_c$ for n a small integer.

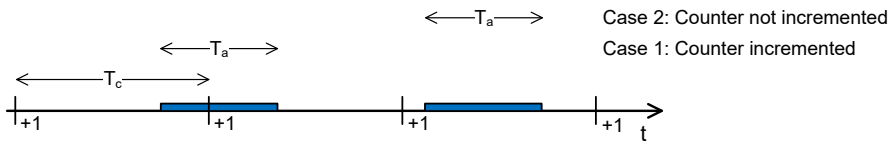


Fig. 6.5: Approximate measures of short intervals

There are two possible cases when measuring an interval of size $T_a < T_c$ (see Figure 6.5): (1) the measured interval begins in one clock period and ends in the next, that is, there is one clock tick during the measurement incrementing the timer's counter variable and (2) the measured interval begins and ends within the same clock period, that is, there is no clock tick during the measurement. Each measurement can thus be seen as a Bernoulli experiment. The outcome of the experiment is 1 with probability $p = T_a/T_c$ corresponding to the first case (counter is incremented during measurement) and 0 with probability $(1-p)$ corresponding to the second case (counter is not incremented during measurement). If we repeat this experiment n times and count the number of times the outcome is 1, the resulting distribution will be approximately Binomial. This is because we cannot assume that the n repetitions are independent, which is required for a true Binomial distribution. The approximation will be more accurate if we introduce a random delay between the successive repetitions of the Bernoulli experiment. If the number of times we get outcome 1 is k, then the ratio $\hat{p} = k/n$ will be a point estimate of p (see Chapter 4, Section 4.2.4). From this, we can derive an estimate for the duration of the measured interval as follows:

$$p \approx \frac{k}{n} \Rightarrow \frac{T_a}{T_c} \approx \frac{k}{n} \Rightarrow T_a \approx \frac{k}{n} T_c \tag{6.1}$$

Furthermore, as shown in Chapter 4, Section 4.2.4, the following approximate confidence interval for p can be derived:

$$P\left(\hat{p} - z_{\alpha/2}\sqrt{\frac{\hat{p}(1-\hat{p})}{n}} \leq p \leq \hat{p} + z_{\alpha/2}\sqrt{\frac{\hat{p}(1-\hat{p})}{n}}\right) \approx 1 - \alpha. \tag{6.2}$$

Multiplying both sides by T_c and considering that $pT_c = T_a$, we obtain the following confidence interval for T_a:

$$P\left(\hat{p}T_c - z_{\alpha/2}T_c\sqrt{\frac{\hat{p}(1-\hat{p})}{n}} \leq T_a \leq \hat{p}T_c + z_{\alpha/2}T_c\sqrt{\frac{\hat{p}(1-\hat{p})}{n}}\right) \approx 1 - \alpha. \tag{6.3}$$

6.3 Performance Profiling

Performance profiling is a process of measuring how much time a system spends in certain states that are of interest for understanding its behavior. A *profile* provides a summary of the execution behavior in terms of the fraction of time spent in different states, for example, the fraction of time spent executing a given function or method, the fraction of time the operating system is running in kernel mode, the fraction of time doing storage or network I/O, or the fraction of time a Java Virtual Machine is running garbage collection. It is often distinguished between *application profiling* and *systems profiling*, where the former stresses that a specific application is being profiled in the case of multiple applications running on the system under test. Application profiling normally aims to identify hotspots in the application code that

may be potential performance bottlenecks, whereas systems profiling typically aims to identify system-level performance bottlenecks. A profile may be used as a basis for performance tuning and optimization; for example, heavily loaded application components may be refactored and optimized or system configuration parameters such as buffer sizes, cache sizes, load balancing policies, or resource allocations may be tuned.

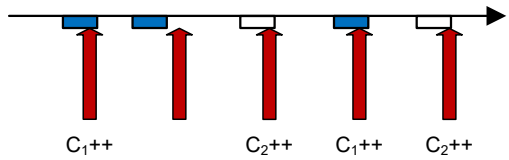


Fig. 6.6: Profiling implemented using sampling-based measurement

Performance profiling is normally implemented using a sampling-based measurement approach. The execution is periodically interrupted to inspect the system state and store relevant information about the states of interest (see Figure 6.6). Assume that there are k states of interest and the goal is to determine the fraction of time spent in each of them. Denote with C_i for $i = 1, 2, ..., k$ the number of times the system was observed to be in state i when interrupted during the profiling experiment. In that case, the interrupt service routine would simply check the current state and increment the respective element of an integer array used to store C_i. At the end of the experiment, a histogram of the number of times each state was observed would be available (see Figure 6.7).

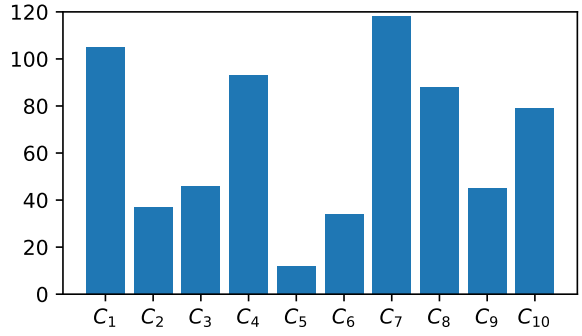


Fig. 6.7: Histogram of state frequencies

Assume that the system is interrupted n times to inspect its state. An estimate of the fraction of time the system spends in state i is given by $\hat{p}_i = C_i/n$. The confidence intervals for proportions that we derived in Chapter 4 (Section 4.2.4) can now be used

to obtain an interval estimate of the fraction of time p_i the system spends in state i. Applying Equation (4.31) from Chapter 4, Section 4.2.4, we obtain the following confidence interval for p_i:

$$P\left(\hat{p}_i - z_{\alpha/2}\sqrt{\frac{\hat{p}_i(1-\hat{p}_i)}{n}} \leq p_i \leq \hat{p}_i + z_{\alpha/2}\sqrt{\frac{\hat{p}_i(1-\hat{p}_i)}{n}}\right) \approx 1-\alpha. \tag{6.4}$$

We note that the above approach works under the assumption that the interrupts occur asynchronously with respect to any events in the system under test. This is important to ensure that the observations of the system state are independent of each other.

Example A Java program is run for 10 s and interrupted every 40 μs for profiling. The program was observed 36,128 times to execute method A. We apply Equation (6.4) to derive a 90% confidence interval for the time spent in method A.

$$\begin{aligned} &m = 36{,}128 \\ &n = 10\,s/40\,\mu s = 250{,}000 \\ &p = m/n = 0.144512 \\ &(c_1, c_2) = 0.144512 \mp 1.645\sqrt{\frac{0.144512(0.855488)}{250{,}000}} = (0.144, 0.146) \end{aligned} \tag{6.5}$$

We conclude with 90% confidence that the program spent 14.4–14.6% of its time in method A.

6.4 Event Tracing

A performance profile provides a high-level summary of the execution behavior of an application or system; however, this summary does not provide any information about the order in which events occur. Event-driven tracing strategies can be used to capture such information. A *trace* is a dynamic list of events generated by the application (or system under study) as it executes (Lilja, 2000). A trace may include any information about the monitored events of interest that is relevant for characterizing the application behavior. In the following, we introduce *call path tracing*, a technique for extracting a control flow graph of an application.

6.4.1 Call Path Tracing

Consider a system that processes transactions requested by clients.[2] An executed system transaction translates into a path through a control flow graph whose edges are basic blocks (Allen, 1970). A *basic block* is a portion of code within an application with only one entry point and only one exit point. A path through the control flow graph can be represented by a sequence of references to basic blocks. It is assumed that the system can be instrumented to monitor the so-called *event records*.

Definition 6.1 (Event Record) An *event record* is defined as a tuple $e = (l, t, s)$, where l refers to the beginning or end of a basic block, t is a timestamp, and s identifies a transaction. The event record indicates that l has been reached by s at time t.

In order to *trace* individual transactions, a set of event records has to be obtained at run time. The set of gathered event records then has to be: (1) partitioned and (2) sorted. The set of event records is partitioned in equivalence classes $[a]_{\mathcal{R}}$ according to the following equivalence relation:

Definition 6.2 $\mathcal{R}$ is a relation on event records: Let $a = (l_1, t_1, s_1)$ and $b = (l_2, t_2, s_2)$ be event records obtained through instrumentation. Then, a relates to b, that is, $a \sim_{\mathcal{R}} b$, if and only if $s_1 = s_2$.

Sorting the event records of an equivalence class in chronological order leads to a sequence of event records that can be used to derive a call path trace. We refer to Briand et al. (2006), Israr et al. (2007), and Anderson et al. (2009) where call path traces are transformed, for example, to UML sequence diagrams.

To reduce the overhead of monitoring system transactions, there exist two orthogonal approaches: (1) *quantitative throttling*—throttling the number of transactions that are actually monitored—and (2) *qualitative throttling*—throttling the level of detail at which transactions are monitored. Existing work on (1) is presented, for example, in Gilly et al. (2009). The authors propose an adaptive time slot scheduling for the monitoring process. The monitoring frequency depends on the load of the system. In phases of high load, the monitoring frequency is throttled. An example of an approach based on (2) is presented in Ehlers and Hasselbring (2011); this approach supports adaptive monitoring of requests; that is, monitoring probes can be enabled or disabled depending on what information about the requests should be monitored.

When extracting call path traces, one is typically interested in obtaining control flow statistics that summarize the most important control flow information in a compact manner. In the rest of this section, we describe the typical control flow statistics of interest by looking at an example.

---

[2] The term *transaction* here is used loosely to refer to any unit of work or processing task executed in the system, for example, an HTTP request, a database transaction, a batch job, a web service request, or a microservice invocation. Transactions, in this context, are also commonly referred to as *requests* or *jobs*.

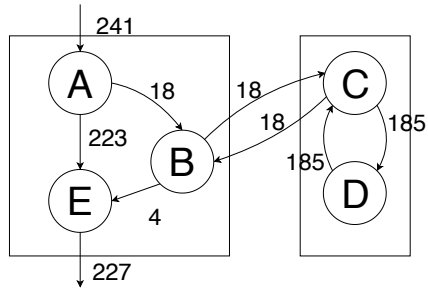


Fig. 6.8: Example call path

Consider the example call path shown in Figure 6.8. We have two different components, depicted as rectangles. The first component contains the basic blocks A, B, and E; the second component contains the remaining basic blocks C and D. An arrow between two basic blocks denotes that the control flow is handed over from one node to another (e.g., by calling a method or service).

The numbers next to the arrows indicate the amount of event records that took the respective explicit call path. The two components could, for example, correspond to two different web servers, communicating over the Internet, while offering certain method interfaces in the form of A, B, C, D, and E. As another example, the components could also correspond to two methods, with A, B, C, D, and E each being a portion of code executed when the methods are called. Here, the first method calls the second method and is then blocked until the second method returns the control flow back to the first method. The granularity of a basic block depends on the specific use case, but also on the capabilities of the tracing tool.

In our example, Figure 6.8 shows 241 event records that enter the first component and trigger execution of basic block A. The latter contains a branch, where 223 of all transactions are directly forwarded to basic block E, and 18 transactions are forwarded to basic block B. For each of those 18 incoming transactions, B is assumed to issue an external call to basic block C in the second component. Basic block C contains a loop that triggers 185 executions of basic block D for each of the 18 transactions. C aggregates the returned information for each of the 18 requests and sends the response back to B. B implements a filtering step based on the information provided by C and therefore again implements a branching, where only four of the 18 received transactions are forwarded to E. Finally, E processes and returns all transactions received by both A and B.

From the described example, we can outline four basic types of information obtainable by call path tracing:

- Call frequencies,
- Branching probabilities,
- Loop iteration counts, and
- Processing times and response times.

We now discuss each of these in more detail.

6.4.1.1 Call Frequencies

By tracing the control flow of transactions between the different basic blocks, it is easy to simply count the frequencies of ingoing and outgoing transactions for each block. Figure 6.8 shows the frequencies at the edges connecting the basic blocks. We usually distinguish between internal and external calls. An *external* call is a call between two different components. In Figure 6.8 components are depicted as rectangles—basic blocks A, B, and E form one component, and basic blocks C and D form another component. Hence, the call from B to C can be seen as an external call.

The calls triggered by a basic block can be easily derived by dividing the number of outgoing edges by the number of incoming edges as measured by call path tracing.

6.4.1.2 Branching Probabilities

Branching probabilities describe the probability of entering each branch transition for every entry of a branch. In Figure 6.8, basic block A represents a branch between forwarding an incoming transaction to block E, or forwarding it to block B. Determining the branching probabilities of a given block is very important for analyzing the performance of a given control flow. For example, Figure 6.8 exhibits very different behavior for transactions forwarded directly to E compared to transactions forwarded to B first. Note that it is also possible to have more than two branch transitions, for example, three, four, or more different actions to take for any specific transaction. In order to extract the respective branching probabilities, one can divide the number of transactions of each particular branch transition by the number of total entries into the branch.

6.4.1.3 Loop Iteration Counts

Similarly to branching probabilities, loop iteration counts are important parameters when analyzing the control flow of an application. Loop iteration counts quantify, how often a specific basic block is entered due to the execution of a loop as part of a transaction. This behavior can be seen at basic block D in Figure 6.8, where basic block C calls basic block D in a loop. The loop iteration counts can be quantified by dividing the number of loop iterations (i.e., sum of loop body repetitions by all transactions) by the number of loop entries (i.e., number of transactions reaching the beginning of the loop).

6.4.1.4 Processing Times and Response Times

The processing time of an individual basic block, as well as of an entire transaction (i.e., the transaction response times), can be easily determined based on the timestamps of the event records corresponding to the beginning and end of the considered basic block and transaction, respectively.

In addition to the above described control flow statistics, tracing tools typically also report transaction throughput and resource utilization data. This allows one to determine further parameters such as *service demands*—also referred to as *resource demands*—of the individual basic blocks or entire transactions. The service/resource demand of a transaction at a given system resource is defined as the average total service time of the transaction at the resource over all visits to the resource. The term resource demand will be introduced more formally in Chapter 7, Section 7.1. Chapter 17 presents a detailed survey and systematization of different approaches to the statistical estimation of resource demands based on easy to measure system-level and application-level metrics. Resource demands can be considered at different levels of granularity, for example, for individual basic blocks or for entire transactions.

6.4.2 Performance Monitoring and Tracing Tools

A number of commercial and open-source monitoring tools exist that support the extraction of call path traces and estimation of the call path parameters discussed above.

Commercial representatives are, for example, Dynatrace,[3] New Relic,[4] AppDynamics,[5] or DX APM.[6] Commercial tools normally have several advantages including product stability, available customer support as well as integrated tooling for analysis and visualization, providing fast and detailed insights into execution behavior.

In addition, many open-source and academic tools are available, such as inspectIT Ocelot,[7] Zipkin,[8] Jaeger,[9] Pinpoint,[10] or Kieker.[11] Open-source tools are often limited in their applicability, supported programming languages, and tooling support; however, they have the advantage of flexibility, extensibility, and low cost. For example, the Kieker framework (Hoorn et al., 2012) has been heavily used and extended

[3] https://www.dynatrace.com

[4] https://newrelic.com

[5] https://www.appdynamics.com

[6] https://www.broadcom.com/products/software/aiops/application-performance-management

[7] https://www.inspectit.rocks

[8] https://zipkin.io

[9] https://www.jaegertracing.io

[10] https://naver.github.io/pinpoint

[11] http://kieker-monitoring.net

over the past 10 years by performance engineers both from industry and academia. Some examples of academic works employing Kieker for research purposes include (Brosig et al., 2011; Grohmann et al., 2019; Spinner et al., 2015; Walter, 2018).

6.5 Concluding Remarks

This chapter introduced different measurement techniques that can be used in practice to derive the values of common metrics. While most presented techniques are useful for performance metrics, some of them can also be applied generally for other types of metrics. The chapter started with a brief introduction to the basic measurement strategies, including event-driven, tracing, sampling, and indirect measurement. We then looked at interval timers, which are typically used to measure the execution time of a program or a portion of it. We discussed in detail several issues related to interval timers, that is, timer rollover, timer accuracy, and strategies for measuring short intervals. Next, we looked at performance profiling, which provides means to measure how much time a system spends in different states. A performance profile provides a high-level summary of the execution behavior of an application or a system; however, this summary does not provide any information about the order in which events occur. Thus, at the end of the chapter, event-driven tracing strategies were introduced, which can be used to capture such information. A trace is a dynamic list of events generated by the application (or system under study) as it executes; it may include any information about the monitored events of interest that is relevant for characterizing the application behavior. We focused on call path tracing—a technique for extracting a control flow graph of the application. Finally, the chapter was wrapped up with an overview of commercial and open-source monitoring tools that support the extraction of call path traces and the estimation of call path statistics, such as call frequencies, branching probabilities, loop iteration counts, and response times.

References

Allen, F. E. (1970). “Control Flow Analysis”. *ACM SIGPLAN Notices*, 5(7). ACM: New York, NY, USA, pp. 1–19 (cited on p. 141).

Anderson, E., Hoover, C., Li, X., and Tucek, J. (2009). “Efficient Tracing and Performance Analysis for Large Distributed Systems”. In: *Proceedings of the 2009 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 2009)*. (London, UK). IEEE Computer Society: Washington, DC, USA, pp. 1–10 (cited on p. 141).

Briand, L. C., Labiche, Y., and Leduc, J. (2006). "Toward the Reverse Engineering of UML Sequence Diagrams for Distributed Java Software". *IEEE Transactions on Software Engineering*, 32(9). IEEE Computer Society: Washington, DC, USA, pp. 642–663 (cited on p. 141).

Brosig, F., Huber, N., and Kounev, S. (2011). "Automated Extraction of Architecture-Level Performance Models of Distributed Component-Based Systems". In: *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. (Oread, Lawrence, Kansas). IEEE Computer Society: Washington, DC, USA, pp. 183–192 (cited on p. 145).

Ehlers, J. and Hasselbring, W. (2011). "Self-Adaptive Software Performance Monitoring". In: *Software Engineering 2011 – Fachtagung des GI-Fachbereichs Softwaretechnik*. Ed. by R. Reussner, M. Grund, A. Oberweis, and W. Tichy. Gesellschaft für Informatik e.V.: Bonn, Germany, pp. 51–62 (cited on p. 141).

Gilly, K., Alcaraz, S., Juiz, C., and Puigjaner, R. (2009). "Analysis of Burstiness Monitoring and Detection in an Adaptive Web System". *Computer Networks*, 53(5). Elsevier North-Holland, Inc.: Amsterdam, The Netherlands, pp. 668–679 (cited on p. 141).

Grohmann, J., Eismann, S., Elflein, S., Kistowski, J. von, Kounev, S., and Mazkatli, M. (2019). "Detecting Parametric Dependencies for Performance Models Using Feature Selection Techniques". In: *Proceedings of the 27th IEEE International Symposium on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2019)*. (Rennes, France). IEEE Computer Society: Washington, DC, USA (cited on p. 145).

Hoorn, A. van, Waller, J., and Hasselbring, W. (2012). "Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis". In: *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering (ICPE 2012)*. (Boston, Massachusetts, USA). ACM: New York, NY, USA, pp. 247–248 (cited on p. 144).

Israr, T., Woodside, M., and Franks, G. (2007). "Interaction Tree Algorithms to Extract Effective Architecture and Layered Performance Models from Traces". *Journal of Systems and Software*, 80(4). Elsevier Science Inc.: Amsterdam, The Netherlands, pp. 474–492 (cited on p. 141).

Kuperberg, M., Krogmann, M., and Reussner, R. (2009). "TimerMeter: Quantifying Properties of Software Timers for System Analysis". In: *Proceedings of the 6th International Conference on Quantitative Evaluation of SysTems (QEST 2009)*. (Budapest, Hungary). IEEE: Piscataway, New Jersey, USA, pp. 85–94 (cited on p. 136).

Kuperberg, M. and Reussner, R. (2011). "Analysing the Fidelity of Measurements Performed with Hardware Performance Counters". In: *Proceedings of the 2nd ACM/SPEC International Conference on Performance Engineering (ICPE 2011)*. (Karlsruhe, Germany). ACM: New York, NY, USA, pp. 413–414 (cited on p. 137).

Lilja, D. J. (2000). *Measuring Computer Performance: A Practitioner's Guide*. Cambridge University Press: Cambridge, UK (cited on pp. 132–137, 140).

Spinner, S., Casale, G., Brosig, F., and Kounev, S. (2015). "Evaluating Approaches to Resource Demand Estimation". *Performance Evaluation*, 92. Elsevier Science: Amsterdam, The Netherlands, pp. 51–71 (cited on p. 145).

Walter, J. C. (2018). "Automation in Software Performance Engineering Based on a Declarative Specification of Concerns". PhD thesis. Würzburg, Germany: University of Würzburg (cited on p. 145).