CHAPTER 3

MEASUREMENT/TESTING TECHNIQUE

The measurement technique of performance evaluation of computer and telecommunication systems is considered the most credible technique; however, it is the most expensive. The measurement or testing technique can be implemented on the real system or the prototype version of the system intended to be built. System performance measurement involves monitoring the system under study while it is being subjected to a particular set of workload or benchmark application programs. It is vital that the system performance analyst understand the following concepts before invoking the task of performance measurements: (a) system application, (b) performance metrics (measures) of interest to the analyst, (c) method of system instrumentation and how the system under study is monitored, (d) representative of workload to real applications, (e) methods of presenting performance results, and (e) techniques to design measurement monitors [1–21].

This chapter focuses on the fundamental concepts of the measurement technique, tracing, tools, as well as issues and solutions.

3.1 MEASUREMENT STRATEGIES

In the measurement technique, different kinds of performance metrics are usually needed depending on the nature of the system under test and application.

Fundamentals of Performance Evaluation of Computer and Telecommunication Systems,
By Mohammad S. Obaidat and Noureddine A. Boudriga

From the viewpoint of event type, these metrics can be categorized into the following classes [1–3, 7]:

- **Event-count metrics.** This class includes metrics that are simply counts of the number of times a specific event takes place, such as the number of packets that arrive with noise, number of cells discarded because of congestion or deadlock, and number of cache misses.
- **Profiles.** In computer systems, a profile represents an aggregate metric for characterizing the overall behavior of a program or a whole system. For example, degree of parallelism (DOP) represents the total number of active processors in a parallel computer system at each instant of time during the execution of a specific application program.
- **Auxiliary-event metrics.** Auxiliary metrics note the values of secondary system parameters when a specific event happens.

The strategy used to measure the performance metric of interest can be decided based on the event type classification discussed above. The chief strategies are as follows [2, 6, 17–21]:

- **Event-driven strategy.** This scheme records the needed information to calculate the metric whenever the events of interest occur. For instance, the desired metric may be the number of cells lost in a computer network's or the number of cache miss in a computer system. To find this number, the analyst should provide a way to record these events whenever they occur and update the appropriate counter. At the end of the session, a mechanism should be provided to dump the content of the counter. This strategy has the advantage that the overhead needed to monitor the event of interest is spent only when the event happens. However, this characteristic is considered as a drawback when the event occurs frequently.
- **Tracing strategy.** This scheme relies on recording more data than only a single event. This means that we need more storage space for this strategy compared with the event-driven scheme.
- **Indirect.** This scheme is used when the performance measure (metric) of interest cannot be measured directly. In such a case, the analyst should look for a metric that can be measured directly and from which the required metric can be derived.
- **Sampling.** This strategy relies on recording the system's state needed to find out the performance metric of interest. Clearly, the sampling frequency here determines the measurement overhead. The latter is determined by the resolution needed to sample the required events.

3.2 EVENT TRACING

In general, a trace consists of an ordered list of events and their related variables.

Such captured information is gathered using profiling tools, and it provides a summary representation of the overall execution of a task. The events of a trace can be a time-ordered list of all instructions executed by a program, sequence of cache addresses by a program, sequence of disk blocks referenced by a file system, and so on. Events may be represented at several levels of details, such as variable trace, procedure trace, or event trace. It is worth mentioning that tracing adds additional processing overhead. Thus, it is important to provide switches to enable or disable tracing as needed.

The traces can be investigated and analyzed to characterize the behavior of a program or a system. In simulation tasks such as cache memory simulation and communication network simulation, traces can be used to drive the simulation programs. Other examples include paging algorithms, central processing units (CPUs) scheduling schemes, and deadlock avoidance algorithms in parallel and distributed systems, among others. Because the size of such traces is usually large, trace compression is often used in such cases to expedite the process of simulation.

Traces are also often used to verify simulation models and to analyze and tune resource management algorithms. However, it is worth pointing out here that trace-driven simulation has many advantages, including [1–10]: (a) better credibility, (b) fair comparison of alternative schemes; (c) less randomness and more close to real operating conditions; (d) more detailed, which helps to find the best trade off alternative; and (e) easier to validate.

The main drawbacks of trace-driven simulation include (a) single point validation, because traces give only one point of validation; (b) unnecessary high-level details as traces are generally long; (c) trace-driven simulation is complex as it requires a more detailed simulation of the system; (d) finiteness characteristic as a detailed trace of a minute or so may fill an entire disk and a simulation result based on such a few minutes may not represent the behavior of the system under study; and (e) poor representation, as traces may become obsolete faster than other forms of workload types.

Any tracing system consists of three key parts (a) the application program being traced that generates the traces; (b) the trace consumer, which is the program that uses the information (e.g., simulator); and (c) the disk file to store the traces. The latter may not be needed in cases where the traces are too large to fit in a disk. In such a case, the traces are used online. Figure 3.1 shows a simplified block diagram of a tracing system.

Traces can be generated using several techniques. These are as follows:

- **Software exception**. Some microprocessors have a mode of operation called the trace mode. A special control bit called the trace bit is used, and when it is enabled, tracing is performed. This is a sort of software exception, which slows down the program being executed.
- **Modification of source code**. In this technique, the source program to be traced is modified so that when it is compiled and executed, extra statements are executed to generate the required traces. This scheme

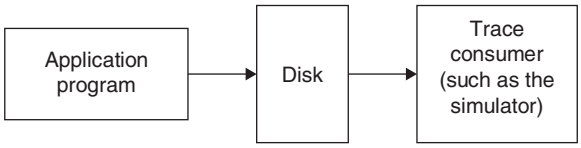

FIGURE 3.1. A simplified block diagram of a tracing system.

allows the performance analyst to trace specific events, which reduces the need to store a huge volume of traces. The main drawback of this scheme is that inserting the trace points is prone to error as it is a manual process.

- **Modification of microcode**. This approach was used when microprocessors were used to execute their instruction sets using interpretation. In such processors, it was possible to modify the microcode to generate traces of each executed instruction. However, in today's state-of-the-art microprocessors, there is no microcode, which limits the applicability of this scheme.
- **Emulation**. In this scheme, the emulation program is modified to trace the execution of the application program. Keep in mind that emulation enables us to execute a program by emulating a processor's instruction sent.
- **Modification of the executable code produced by the compiler**. In the technique, we add supplementary instructions at the beginning of each block in order to mark when the block is inserted.

One major concern when generating traces is the large volume of data produced in a very short time. This prompted performance analysts to devise schemes to compress the traces without affecting the degree of representation. The major techniques that can be used to reduce the size of information produced by tracing are as follows [2]:

- **Data compression**. This technique relies on applying data compression techniques to reduce the size of traces. It has the potential to reduce the size of traces by a factor of 25% of the original size. The tradeoff in using this scheme is the additional time needed to compress and uncompress the traces when traces are generated and consumed, respectively.
- **Compression of traces online**. The idea here is to consume the traces online as generated without the need to store them for later use. This saves the need to have a large disk system to store the traces; however, there is a possible problem in multitask environments. There is no guarantee that the next time the program is traced, the same succession of events will occur because system events happen asynchronously with respect to a traced program. Clearly, this is of great concern when repeatability in generating traces is needed.
- **Sampling of traces**. The main idea here is to save only a relatively small part of the trace sequences scattered throughout the collected or generated

trace data. Because such samples are considered statistically representative of the entire trace data, they can be used to drive the simulator. This has the advantage of storing a small part of the trace data. It is worth mentioning here that there is no theoretical basis to help the performance analysts decide on the size of each sample as well as the sampling rate. This technique has been used in the simulation of cache memory systems.

- **Abstract execution**. This scheme separates traces into the following two stages: (a) compiler analysis of the program to be traced to identify a small part of the entire trace that can be used later on to reproduce the entire trace and (b) execution of special trace-generation procedures to convert this small part of the trace data into the full trace data. Empirical studies have shown that this tracing scheme slows down the execution of the program being traced by an average factor of 6, which is not higher than other tracing schemes. Moreover, because this scheme records information only about the changes that actually occur during run time, it can reduce the volume of trace data stored by a factor of several hundreds.\

3.3 MONITORS

Monitors are defined as tools that are used to observe and record the activities of a system under testing and analysis. The main functions of a monitor used in performance evaluation are to: (a) observe the performance of the system, (b) collect performance statistics, (c) analyze the data collected, and (d) display the results if possible. Hardware monitors have gone through several generations of developments. State-of-the art hardware monitors are intelligent and have programmable devices as well as smart components such as processors and needed peripherals. Almost all monitors when embedded in the system under test cause overhead. It is desired to reduce the monitor overhead [1–5].

Monitors can be categorized based on the implementation level, trigger mechanism, and display capability. From point of view of the levels at which monitors are implemented, we have the following types of monitors: (a) hardware monitors, (b) software monitors, and (c) hybrid or firmware monitors. A brief description of each of them is given below.

3.3.1 Hardware Monitors

A hardware monitor is a device that consists of several components. It is attached to the system to be monitored and analyzed to collect information related to events of specific interest. Probes are usually used to connect the components of the hardware monitor to the system under test. In general, a hardware monitor consumes no system resources, and it has a lower overhead as compared with to a software monitor. Hardware monitors are usually faster

than software monitors. The basic building blocks of a hardware monitor are as follows:

- **Probes.** These are used to connect the monitor to the circuit or hardware points of interest of the system under test. A probe in general has high impedance.
- **Logic gates**. These components are needed to construct various functional units that help to indicate events that may increment counters or test conditions.
- **Counters.** These devices are needed to increment or decrement the occurrence of events of interest.
- **Timers.** These are needed for time stamping or triggering a sampling operation.
- **Comparators.** Such components are used to compare contents of counters and to test for specific conditions.
- **Storage device**. Almost all hardware monitors have built-in storage devices, such as tapes or compact disk (CD) drivers, to store observed data.

3.3.2 Software Monitors

These types of monitors are basically computer programs that are embedded in the operating system. They are meant to observe events in the operating system and higher level software, such as in databases and networks. It is essential to have the operating rate of the monitor high enough so that it can observe the needed events and collect the needed data properly. Also, the overhead should be small.

In general, software monitors have lower input rates, lower resolution, and higher overhead when compared with hardware monitors. However, they have higher input width and recording capacities than hardware monitors. Also, they are less expensive and easier to implement than hardware monitors. Because software monitors have high overhead, they should be designed so that their function can be easily disabled when needed using a simple flag.

In designing any software monitor, several issues should be considered. Among these are the following:

1. **Buffer size.** The size of the buffer memory should be optimal. This means that it should be large enough so that the rate of writing to the auxiliary storage is reduced, and at the same time, it should be small so that the time lost per writing operation is not too large.
2. **Number of buffers**. The minimum number of buffers should be two. This is because if there is only one buffer, then the monitoring (filling of buffer) and recording (emptying) processes cannot be performed simultaneously.

3. **Method of activation**. This refers to the way by which the software monitor's data collection procedure is triggered. Among these methods that are used in microprocessor systems are: (a) trace mode, (b) trap instruction(s), and (c) timer interrupt.

   The trace mode is available in most microprocessors. In this scheme, the execution of instructions is interrupted after each instruction, and control is passed on to a special procedure to collect data. This technique suffers from a high overhead.

   The trap instruction-based scheme relies on using a trap instruction in the middle of the code where it is inserted at selected points. When the trap instruction is executed, program execution is transferred to a special data collection procedure.

   The timer interrupt-based scheme is provided by the operating system to transfer control of execution to a special data collection procedure at fixed intervals. One interesting characteristic of this scheme is that it has low overhead because the overhead is independent of the event rate.
4. **Enable/Disable.** Because of the overhead incurred when the software monitor is enabled, it is desired to disable the software monitor easily when monitoring is no longer needed. Moreover, such an on/off capability permits debugging of the code.
5. **Overflow management**. There is always a possibility that the monitor's buffer reaches the overflow state. To avoid losing traces, designers of software monitors should provide a mechanism by which the monitoring process is stopped whenever the buffer overflows. The choice between overwriting the buffer or not depends on the application and needs of the analyst. The goal here is to detect the occurrence of buffer overflow.
6. **Programming language**. To reduce overhead and latency, it is desired to write the monitor code in low-level languages such as assembly language. However, because the monitor is embedded in the system software, therefore, it is better to have both written in the same programming language.
7. **Monitor priority**. To minimize the effect of the monitor's on system's operation, it should be given a low priority, especially if the monitor is run asynchronously. However, if it is required to observe important timely observations, then the monitor should be given a high priority.

3.3.3 Hybrid Monitors

This type of monitors is also called by other authors as firmware monitors. These monitors are often used for applications where speed and timing consideration prevent the use of software monitors, and difficulty of accessing probe points prevents the use of hardware monitors. They are popular to monitor networks where existing interfaces can be easily microprogrammed

to monitor the flow of traffic. Furthermore, they can be used to generate address profiles of microcode, which are used to optimize the code to improve the speed of execution.

Hybrid monitor use software and hardware means for their operation. When implemented properly, hybrid monitors have the potential of providing the high-resolution characteristics of hardware monitors and the data reduction capabilities of software monitors.

It is worth mentioning in this context the distributed-system monitors that are used to monitor distributed and parallel systems. In such a case, the monitor itself must be distributed and should be made of several components that are supposed to work concurrently. The main functions of a distributed and parallel-system monitor are:

1. Gathering of data from individual system's components.
2. Collection of data from different observers.
3. Analysis of data collected using special statistical routines that summarize the data characteristics.
4. Preparation of performance reports.
5. Interpretation of performance reports using either human experience or expert systems.
6. Based on result's interpretation, a special entity is supposed to make decisions to set or change system parameters or configurations. The component that performs this task is often called the manager, and it is implemented in advanced monitors with automated monitoring and control features [1–4].

3.4 PROGRAM OPTIMIZERS

Program execution monitors or program execution analyzers (program optimizers) are of great interest to performance evaluation analysts. Monitoring the execution of a computer program is needed for the following reasons:

1. To locate the execution path of a code
2. To determine the time spent in various sections of a program
3. To locate the most frequent or most time-consuming segment of the program
4. To test the relationship between the variables and parameters of a program
5. To establish the adequacy of a test run of the program

In general, programs to be monitored are chosen depending on the following criteria: (a) frequency of use, (b) time criticality, and (c) resource demand.

The chief issues in designing a program execution (optimizer) monitor are pretty much similar to these for software monitors. In addition there are several issues that are specific to the optimizer monitors, including [1, 4] the following:

1. **Frequency and time histogram**. Almost all program monitors produce an execution profile with different levels of hierarchy, such as summaries by modules, for each module by procedures, and for each procedure by statement. Monitors have the capability to scale up or down the amount of detail.
2. **Measurement units**. An execution program divides the program into smaller units called modules, procedures, high-level language statements, or machine instructions. Data associated with each unit are noted and shown in the final report. Lower level reports such as machine instruction profiles may be too comprehensive for certain applications.
3. **Measurement methods**. We can have two basic measurement schemes: sampling and tracing. The sampling scheme makes use of the system's timer convenience, and it records states at cyclic intervals. If the elapsed time sampling approach is used, then the program may be in a wait state, until an input/output (I/O) operation or some other event is completed. The tracing scheme uses either explicit loops or the trace mode of a microprocessor.
4. **Instrumentation means**. In this scheme, instrumentation can be added before compilation, during compilation, before linking (after compilation), or during run time. To instrument a source code, a high-level routine call statement is added at a strategic location in the program. This call statement transfers control to the monitor procedure that it is supposed to collect data [2].

3.5 ACCOUNTING LOGS

Accounting logs provide interesting useful information about the usage and performance of the system; therefore, many analysts consider them software monitors. It is recommended that before creating a monitor, the analyst should benefit from the data provided by accounting logs.

Accounting logs can be used without extra efforts as they are built into the system. Data collected using accounting logs are accurate as they represent real operation with little overhead. However, accounting log analysis programs are not generally provided. For a more accurate analysis, a system analyst should develop additional analysis programs, including statistical analysis programs.

In general, the precision of accounting logs is not high, and most of them contain no system-level information including device utilization, queueing time, or queue length. In addition to the queueing time, the elapsed time includes the

service time of resources. Moreover, almost all accounting logs do not record the time spent waiting for user inputs; therefore, this time cannot be distinguished from the queueing time. The typical information provided by an accounting log include the program name, program start time, program end time, CPU time used by the program, number of disk reads and writes, number of terminal reads and writes, and so on.

The main usages of accounting logs are to know: (a) usage of resources, (b) programs that users should be trained to use more efficiently, (c) programs that need better code optimization, (d) which application programs are I/O bound, (e) programs that have poor locality of reference, (f) number of jobs that can be run at the same time without performance degradation, and (g) programs that provide the best opportunity for better human interface [1, 2, 4].

3.6 SUMMARY

This chapter presented the main aspects of the measurement/testing technique of performance evaluation of computer and telecommunication systems. In particular, we studied the strategies to be followed when invoking into a measurement task, which include the event-driven, tracing, and sampling strategies. We also studied the main performance metrics that depend on the applications and objectives of the performance evaluation study. Then we studied event tracing and its significance, drawbacks, applications, approaches, components, and methods of trace compression. Software, hardware, and hybrid monitors have been investigated along with their functions, types, and design issues. We also have shed some light on program optimizers or program execution analyzers. In particular, we reviewed the main issues in their design such as frequency and time histogram, measurement unit and methods, and instrumentation means. The last section covered in this chapter is accounting logs. Although accounting logs are not accurate, they provide a rough estimate of the performance of resources, especially their use. Accounting logs provide interesting useful information about the usage and performance of the system; therefore, many analysts consider them software monitors. It is recommended that before creating a monitor, the analyst should benefit from the data provided by accounting logs. Also, we addressed the main applications and characteristics of accounting logs.

REFERENCES

[1] M. S. Obaidat, and G. I. Papdimitriou (Eds.), ''Applied System Simulation: Methodologies and Applications,'' Springer, New York, 2003.

[2] R. Jain, ''The Art of Computer Systems Performance Analysis,'' Wiley, New York, 1991.

[3] K. Kant, "Introduction to Computer System Performance Evaluation," McGraw Hill, New York, 1992.

[4] D. J. Lilja, "Measuring Computer Performance," Cambridge University Press, Cambridge, UK, 2000.

[5] M. S. Obaidat, "Advances in Performance Evaluation of Computer and Tele communications Networking," *Computer Communication Journal* Vol. 25, No. 11 12, pp. 993 996, 2002.

[6] M.S. Obaidat, "ATM Systems and Networks: Basics Issues, and Performance Modeling and simulation," Simulation: Transactions of the Society for Modeling and Simulation International, Vol. 78, No. 3, pp. 127 138, 2003.

[7] M.S. Obaidat, "Performance Evaluation of Telecommunication Systems: Models Issues and Applications," Computer Communications Journal, Vol. 34, No. 9, pp. 753 756, 2001.

[8] M. Ghanbari, C. J. Hughes, M. C. Sinclair, and J. P. Eade, "Principles of Performance Engineering for Telecommunication and Information Systems," IEE, Herts, UK, 1997.

[9] K. Hwang, and Z. Xu, "Scalable Parallel Computing," McGraw Hill, New York, 1998.

[10] M. Arlitt, and T. Jin, "Workload Characterization of the 1998 World Cup Web Site," HP Technical Report 1999 35R1, Hewlett Packard, 1999.

[11] M. Arlitt, and C. Williamson, "Internet Web Servers: Workload Characterization and Performance Implications," IEEE/ACM Transactions on Networking, Vol. 5, No. 5, pp. 631 645, 1997.

[12] L. K. John, and A. M. G. Maynard, (Eds.)" Workload Characterization of Emerging Applications," Kluwer Academic Publisher, Dordrecnt, The Netherlands, 2001.

[13] J. L. Hennessy, and D. A. Patterson, "Computer Architecture: A Quantitative Approach," 3rd edition Morgan Kaufmann, New York, 2003.

[14] J. Banks, J. S. Crason II, B. L. Nelson, and D. Nicol, "Discrete Event System Simulation," 3rd edition, Prentice Hall, Upper Saddle River, 2001.

[15] S. M. Ross, "Simulation," 2nd edition, Harcourt Academic Press, San Diego, CA, 1997.

[16] M. S. Obaidat, "Performance Evaluation of the IMPS Multiprocessor System," Journal of Computer and Electrical Engineering, Vol. 15, No. 4, pp. 121 130, 1989.

[17] The Standard Performance Evaluation Corporation: http://www.spec.org.

[18] http://imls.lib.utexas.edu/redesign/slideshow/tsld009.html.

[19] NAS Parallel Benchmarks: http://science.nas.nasa.gov/software/npb/.

[20] Parkbench Parallel Benchmarks: http://www.netlib.org/parbench/.

[21] Transaction Processing Council (TPC) Benchmarks: http://www.tpc.org/.

EXERCISES

1. What are the strategies that can be used for the measurement technique of performance evaluation?

2. Compare and contrast hardware and software monitors.
3. What are the main applications of accounting logs?
4. Which programs must be chosen for I/O optimization? Explain.
5. Choose an IEEE 802.11 wireless local area network (WLAN), review published articles related to its performance evaluation, and make a list of the benchmarks used in these articles.
6. Choose a multiprocessor computer system architecture. Review the related published articles on its performance evaluation, and make a list of the used performance metrics.
7. Select a measurement study of the performance evaluation of a computer system or a communication network in which hardware monitors are used in the study. Explain how useful such monitors are for providing accurate and real measurement about the behavior of the system. Discuss whether you can replace the hardware monitor by a software monitor, and give the advantages and disadvantages for doing so.
8. A workstation uses a 500-MHz processor with a claimed 100-MIPS rating to execute a given program mix. Assume a one-cycle delay for each memory access.
   a. What is the effective cycle per instruction (CPI) of this machine?
   b. Suppose that the processor is being upgraded with a 1000-MHz clock. However, the speed of the memory subsystem remains unchanged, and consequently, two clock cycles are needed per memory access. If 30% of the instructions require one memory access and another 5% require two memory accesses per instruction, what is the performance of the upgraded processor with a compatible instruction set and equal instruction counts in the given program mix?
9. A linear pipeline processor has eight stages. It is required to execute a task that has 600 operands. Find the speedup factor, S_k, assuming that the CPU runs at 1.5 GHz. Note that the speedup factor of a liner pipeline processor is defined by the following expression: Sk = speedup = (time needed by a one-stage pipeline processor to do a task)/(time needed by k-stage processor to do the same task) = T_1/T_k.
10. Devise an experiment to find out the performance metrics for an IEEE 802.3 local area network (LAN)
    a. The throughput of the network as a function of the number of nodes in the LAN.
    b. The average packet delay as a function of the number of nodes in the LAN.
    c. The throughput-delay relationship.