**PERFORMANCE EVALUATION**

Because an OS is primarily a resource manager, it is important for operating systems designers, managers and users to be able to determine how effectively a particular system manages its resources. System performance measurements enable consumers to make informed decisions and help developers build more efficient systems. The performance of a system depends heavily on its hardware, operating system and the interaction between the two

**Self-ReviewQuestion**: (i) How do you suppose performance evaluation benefits consumers, developers and users? (ii) What resource of a computer system probably has the greatest impact on performance?

**Answers:** (i) Performance evaluation provides consumers with a basis of comparison to decide between different systems, developers with useful information on how to write software to efficiently use system components and users with information that can facilitate tuning the system to meet the requirements of a specific setting. (ii) The processor(s).

**IMPORTANT TRENDS AFFECTING PERFORMANCE ISSUES**

In early years of computer systems development, hardware was the dominant cost, so performance studies concentrated primarily on hardware issues. Now, hardware is relatively in expensive and prices continue to decline. Software complexity is increasing with the wide spread use of multithreading, multiprocessing, distributed systems, database management system, geographical user interfaces and various other application support systems. The software typically hides the hardware from the user, creating a virtual machine defined by the operating characteristics of the software. Cumbersome software causes poor performance, even on systems with powerful hardware, so it is important to consider a system’s software performance as well as its hardware performance.

The nature of performance evaluation itself is evolving. Raw and potentially misleading measures, such as clock speed and bandwidth, have become influential in the consumer market, because vendors engineer their products with an eye to these metrics. However, other aspects of performance evaluation are improving. For example, designers have developed more sophisticated benchmarks, acquired better trace data and engineered more comprehensive computer simulation models. New industry-standard benchmarks and “synthetic programs” are emerging. Unfortunately, there is still no consensus on these standards. Critics charge that performance results can be incorrect or misleading, because the performance evaluation techniques do not necessarily measure relevant features of a program.

**Self-Review Question**: (i) How has focus of performance studies shifted over the years? (ii) Give several examples of how operating systems can improve performance

**Answers:** (i) In the early years, hardware was the dominant cost, so performance studies focused on hardware issues. Today, it is recognized that sophisticated software can have a substantial effect on performance. (ii) Operating systems can improve disk and processor scheduling policies, implement more efficient thread-synchronization protocols, more effectively manage file systems, perform context switches more efficiently, improve memory management algorithms, etc.

**WHY PERFORMANCE MONITORING AND EVALUATION ARE NEEDED**

There are three common purposes for performance evaluation.

**. Selection evaluation –** the performance evaluator decides whether obtaining a computer system or application from a particular vendor is appropriate.

**. Performance projection** – the performance evaluator estimates the performance of a system that does not exist. It might be a completely new computer system, or an old system with a new hardware or software component.

**. Performance monitoring** – the evaluator accumulates performance data on an existing system or component to ensure that it is meeting its performance goals. Performance monitoring can also help estimate the impact of planned changes and provide system administrators with the data they need in order to make strategic decisions, such as whether to modify an existing process priority system or upgrade a hardware component.

In the early phase of a new system’s development, the vendor attempts to predict the nature of applications that will run on the system and the anticipated workloads these applications must handle. Once the vendor begins development and implementation of the new system, performance evaluation and prediction are used to determine the best hardware organization, the resource management strategies that should be implemented in the operating system and whether or not the evolving system meets its performance objectives. Once the product is released to the marketplace, the vendor must be prepared to answer questions from potential users about whether the system can handle certain applications with certain levels of performance. Users are often concerned with choosing an appropriate configuration of a system that services their needs.

When the system is installed at the user’s site, both the vendor and the user seek to obtain optimal performance. Administrators fine-tune the system to run at its best in the user’s operating environment. This process, called **system tuning,** can often cause dramatic performance improvements, once the system is adjusted to the idiosyncrasies of the user installation.

**SELF REVIEW**

1. When would an evaluator use performance projection in preference to selection evaluation?
2. How does performance monitoring facilitate system tuning?

**Ans: 1)** selection evaluation help an evaluator choose among existing systems. Performance projection help the evaluator predict the performance of systems that do not exist yet – or how anticipated modifications to existing systems would perform. **2)** An evaluator can monitor performance to determine which modification would most likely increase system performance.

**PERFORMANCE MEASURES**

By performance, we mean the efficiency with which a computer system meets its goals. Thus, performance is a relative rather than an absolute quantity, although we often talk of **absolute performance measures** such as the amount of time in which a given computer system can perform a specific computational task. However, whenever a performance measure is taken, it is normally to be used as a basis of comparison.

Performance is often “in the eye of the beholder.” For example, a young music student might find a performance of Beethoven’s Fifth Symphony thoroughly inspiring, whereas the conductor might be sensitive to the most minor flaws in the way a second violinist plays a certain passage. Similarly, the owner of a large airline reservation system might be pleased with the high utilization reflected by a large volume of reservations processed, whereas an individual user might be experiencing excessive delays on such a busy system.

Quantifying is difficult for some performance measures, such as ease **of use,** and simple for others, such as the speed of a disk-to-memory transfer. The performance evaluator must be careful to consider both types of measures, even though it might be possible to provide neat statistics only for the latter. Some performance measures, such as response time, are user oriented. Others, such as processor utilization, are system oriented.

Some performance results might be deceptive. For example, one operating system might focus on conserving memory by executing complicated page-replacement algorithms**,** whereas another might avoid these complex routines to save processor cycles for executing user programs. The former would appear more efficient on a system with a high clock speed; the latter, on a processor with a large main memory. In addition, some techniques permit the evaluator to measure the performance of small pieces of a system such as individual components or primitives. Although these tools can be useful for pinpointing specific weaknesses, they do not tell the entire story. The evaluator might discover that an operating system performs all of its primitives efficiently except one. This should not be a major concern, unless that inefficient primitive is used extensively. Without taking into account the frequency with which each primitive is used, measurements can be misleading. Similarly, programs designed to evaluate a system for a particular environment that do not resemble the applications for which the system is intended can yield spurious results.

Some common performance measures are:

**. Turnaround time –** this is the time from submission of a job until the system returns a result to the user.

**.response time –** this is an interactive system’s turnaround time, often defined as the time from a user’s pressing an Enter key or clicking a mouse until the system displays its response.

**. system reaction time –** in an interactive system, this is often defined as the time from a user’s pressing Enter or clicking a mouse until the first time slice of service is given to that user’s request.

These are probabilistic quantities, and in simulation and modeling studies of systems they are considered to be **random variables.**  A random variable is one that can assume a certain range of values, where each value has an associated probability of occurring. We discuss the **distribution of response times,** for example, because users experience a wide range of response times on a particular interactive system over some interval of operation. A probability distribution can meaningfully characterize this range.

When we talk of the **expected value** of a random variable, we are referring to its **mean** or average value. However, means can often be deceiving. A certain value can be produced by averaging a series of identical or nearly identical values, or it can be produced by averaging a wide variety of values, some much larger and some much smaller than the calculated mean. Therefore, other performance measures often employed are:

**. Variance in response times** (or of any of the random variables we discuss) – the variance of response times is a measure of **dispersion**. A small variance indicates that the response times experienced by users are generally close to the mean. A large variance indicates that some users are experiencing response time that differs widely from the mean. Some users could be receiving fast service, while others could be experiencing long delays. Thus the variance of response times is a measure of **predictability;** this can be an important performance measure for users of interactive systems.

**. Throughput** – this is the work-per-unit-time performance measurement.

. **Workload –** this is the measure of the amount of work that has been submitted to the system. Often, evaluators define an acceptable level of performance for the typical workload of a computing environment. The system is evaluated compared to the acceptable level.

**.Capacity –** this is a measure of the maximum throughput a system can attain, assuming that whenever the system is ready to accept more jobs, another job is immediately available.

**. Utilization –** this is the fraction of time that a resource is in use. Utilization can be a deceptive measure. Although a high-percent utilization seems desirable, it might be the result of inefficient usage. One way to achieve high processor utilization, for example, is to run a process that is in an infinite loop! Another view of processor utilization also yields interesting insights. We might view a processor at any moment as being idle, in user mode, or in kernel mode. When the processor is in user mode, it is performing operations on behalf of a user. When the processor is in kernel mode, it is performing task for the operating system. Some of this time, such as context-switching time, is pure overhead. This overhead component can become large in some systems. Thus, when we measure processor utilization, we must be concerned with how much of this usage is productive work on behalf of the users, and how much is system overhead. Strangely, “poor utilization” is actually a positive measure in certain kinds of systems, such as hard real-time systems, where the system resources must stand ready to respond immediately to incoming tasks, or lives could be at risk. Such systems focus on immediate response rather than resource utilization.

**Self review**

1. What is the difference between response time and system reaction time?
2. (T/F) when a processor spends most of its time in user mode, the system achieves efficient processor utilization.

**Ans: 1)** response time is the time required for the system to finish responding to a user request (from the time the request is submitted); system reaction time is the required for the system to begin responding to a user request (from the time the request is submitted).

**2)** false. If the processor is executing an infinite loop, this system is not using the processor efficiently.

**Performance evaluation techniques**

Now that we have considered some possible performance measures, we need some way of extracting them. In this section, we describe several important performance evaluation techniques. Some of the isolate different component of a system and report each component’s individual performance, allowing developers to identify areas of inefficiency. Others are oriented toward a system as a whole and allow consumers to make comparisons between systems. Still other techniques are application specific and therefore allow only indirect comparisons with other system.

**Tracing and profiling**

Ideally, a system evaluator would measure the performance of several systems, each executing in the same environment. However, this is usually not feasible, especially for corporations whose environments are complex and difficult to duplicate. Such a process can be invasive, compromising with the integrity of the environment and nullifying the results. When the performance of a system must be evaluated in a given environment, designers often use **trace** data. A trace is a record of system activity- typically a log of user and application requests to the operating system.

System evaluators can use trace data to characterize a partcular system’s execution environment by determining the frequency with which user-mode processes request particular kernel services. Before installing a new computing system, evaluators can test the new system using a workload derived from the trace data or using the trace itself. Trace data often can be modified to evaluate “what-if” scenarios. For example, a system administrator might need to determine how a new web site will affect web server performance. An existing trace can be modified to estimate how the system will handle its new load.

When operating systems execute in similar environment, standard traces can be developed and executed on those systems to compare performance. Trace data obtained from one installation, however, might not be applicable to another. Such data is, at best, an approximation of system activity at the other installation. In addition, user logs are considered proprietary to the system on which they were recorded; rarely are such logs distributed to the research community or to vendors. Consequently, there is a dearth of trace data available for comparison and evaluation.

Another method for capturing a computing system’s execution environment is profiling. **Profiles** record system activity while executing in kernel mode, which can include operations such as process scheduling, memory management and I/O management. For example, a profile might record which kernel operations are performed most often. Alternatively, a profile can simply log all function calls issued by the operating system. Profiles indicate which operating system primitives are most heavily used, enabling system administrators to identify potential targets for optimization and tuning. Evaluators most often employ other performance evaluation techniques in concert with profiles to determine the most effective ways to improve system performance.

**Self review**

1. How do evaluators use trace data?
2. Explain the difference between traces and profiles.

**Ans:**

**1)** Trace data permits evaluators to compare the performance of many different systems that operate in the computing environment. Trace data describes this environment so that evaluators can obtain performance results relevant to the systems’ intended use.

**2)** Traces record user requests, whereas profiles log all activity in kernel mode. Therefore, traces describe a computing environment by capturing the user demand for particular kernel services (without regard to the underlying system), and profiles capture operating system activity in a given environment.

**Timings and microbenchmarks**

**Timings** provide a means of performing quick comparisons of computer hardware. Early computer systems were often evaluated by their adds times or their memory-cycle times. Timings are useful for indicating the “raw horsepower” of a particular computer system, often in terms of number of **MIPS (millions of instructions per second)** or **BIPS (billions of instructions per second)** it executes. Some computers perform in the **TIPS (trillions of instructions per second)** range.

With the advent of **families of computers,** such as the IBM 360 series, first introduced in 1964, or the Intel Pentium series, a descendant of the Intel x86 series introduced in 1978, it had become common for hardware vendors to offer computers that enable a user to upgrade to faster processors (without replacing other computer components) as the user’s needs grow. The computers in a family are compatible in that they can run the same programs but at greater speeds as the user moves up in the family. Timings provide a convenient means for comparing the members of a family of computers.

**A microbenchmark** measures the time required to perform an operating system operation (e.g., process creation). Microbenchmarks are useful for measuring how a design change affects the performance a specific operation. **Microbenchmark suites** are programs that measure the performance of a number of important operative system primitives, such as memory operations, process creation and context-switch latency. Evaluators also use microbenchmarks to measure system performance for specific operations, such as read/write bandwidth (i.e., how much data the system can transfer per unit time during a read or write) and network connection latency.

Microbenchmarks describe how quickly the system performs a particular operation, not how often that operation is performed. Consequently, they do not measure important evaluation criteria such as throughput and utilization. Microbenchmarks, however, are useful in isolating which operations could be causing a system to perform poorly when coupled with information about how each operation is used.

Until the 1990s, no single microbenchmark suite demonstrated the effect of hardware system components on the performance of operating system primitives. In 1995 the **Imbenchmicrobenchmark suite**, which enabled evaluators to measure and compare system performance on a variety of UNIX platforms, was introduced. Although imbench provided useful performance evaluation data that allowed comparison across multiple platforms, it was inconsistent in the way that it reported statistical data – some tests returned results based on an average of runs while other used just one run of the microbenchmark. Imbench was also limited to performing measurements once per millisecond because it used coarse software timing mechanism; this is insufficient for measuring fast operations and timing hardware. Researchers at Harvard University addressed these limitations by creating the **microbenchmark suite,** which provides a standard and rigorous model for reporting statistics, enabling evaluators to more effectively analyze the relationship between operating system primitives and hardware components. Imbench and hbench represent different micro benchmark philosophies. Imbench focuses on portability, which permits evaluators to compare across different architectures; hbench focuses on the relationship between the operating system and its underlying hardware within a particular system.

**Self review**

1. How can results from timings and micro benchmarks be misleading? How are they useful?
2. Which performance measure can be combined with micro benchmarks to evaluate operating system performance?

**Ans: 1)** microbenchmarks measure the time required to perform specific primitives (e.g., process creation), and timings perform quick comparisons of hardware operation (e.g., add instructions). Neither of these measurements reflects system performance as a whole. However, microbenchmarks and timings can be useful in pinpointing potential areas of inefficiency and evaluating the effect of small modifications on system performance. **2)** profiles, microbenchmarks to evaluate operating system performance.

**Application Specific Evaluation**

Although “raw performance” is an important measure, many users are more interested on how well particular applications will perform on a particular system. Seltzer et al. describe a **vector-based methodology**  for calculating an application-specific evaluation of a system by combining trace and profile data with timings and microbenchmarks.

In this technique, an evaluator records the results of microbenchmarks for the operating system’s primitives. Next, the evaluator constructs a vector by placing the values corresponding to the microbenchmark results in the element of the vector; this is called the **system vector**. Next, the evaluator profiles the operating system while executing the target application. The evaluator constructs a second vector by inserting the relative demand for each operating system primitive in an element in the vector; this is called the **application vector.** Each element in the system vector describes how long the operating system needs to execute a particular primitive, and the corresponding entry in the application vector describes the application’s relative demand for that primitive. For example, if the first entry in the system vector records process creation performance, the first entry in the application vector records how many processes were created while executing a particular application (or group of applications). A characterization of the performance of a given system executing a particular application is calculated by

Where is the ith entry in the system vector, is the ith entry in the application vector and n is the size of both vectors.

This technique can be useful for comparing how efficiently different operating systems execute a particular application (or group of applications) by considering the demand an application places on each of a system’s primitives. The vector-based methodology can be used to select operating system primitives to tune to improve system performance.

Some application behaviors depend both on the particular application and on user input. For example, the type of application requests generated in a database system depends on its population of active users. Simply profiling the system without determining the typical stream of user requests can produce misleading results. In such cases, both the application and the user requests determine the system’s execution environment. Therefore, to provide a more accurate system evaluation, the vector-based methodology can be combined with a trace (seltzer et al. call this the **hybrid methodology).** In this case, the trace data allows the evaluator to construct the application vector while accounting for how the specific application and typical stream of user requests affect the demand for each operative system primitive.

A **kernel program** is another application-specific performance evaluation tool, although it is not often used. A kernel program can range from an entire program that is typical of one executed at an installation or a simple algorithm such as a matrix inversion. Using the manufacturer’s estimated instruction timings, execution of a kernel program is timed for a given machine. Machines are then compared on the basis of differences in the expected execution times. Kernel programs are actually “executed on paper” rather than being run on a particular computer. They are used in selection evaluation before a consumer purchases system being evaluated. [Note: “Kernel” here should not be confused with kernel of the operating system.]

Kernel programs give better results than either timings or microbenchmarks, they required manual effort to prepare and time. One key advantage of many kernel programs is that they are complete programs, which ultimately is what the user actually runs on the computer system under consideration.

Kernel programs can be helpful in evaluating certain software components of a system. For example, two different compilers might produce dramatically different code, and kernel programs can help an evaluator decide which one generates more efficient code.

**Self review**

1. What is a benefit of application-specific evaluation?
2. Why do we suppose kernel programs are rarely used?

**Ans: 1)** application-specific evaluation is useful in determining whether a system will perform well in executing the particular programs at a given installation. A drawback is that a system must be evaluated by each installation that is considering using it; the system’s designers cannot simply publish one set of performance results. **2)** Kernel programs require time and effort to. Also, they are “executed on paper” and can introduce human error. Often, it is easier to run the actual program or one similar to it, on the actual system that calculates the execution time for a kernel program.

**Analytical models**

**Analytical models** are mathematical representations of computer systems or their components. Many types of models are used; those of queuing theory and Markov processes are two of the more popular, because they are relatively manageable and useful.

For evaluators who are mathematically inclined, the analytic model can be relatively easy and modify. A large body of mathematical results exists that evaluators can apply in order to estimate the performance of a given computer system or component quickly and fairly accurately. However, several disadvantages of analytic modeling hinder its applicability. One is that evaluators must be highly skilled mathematicians; these people are rare in commercial computing environments. Another is that complex systems are difficult to model precisely; as systems become more complex, analytic modeling becomes useful.

Today’s systems are often so complex that the modeler is forced to make many simplifying assumptions that can diminish the use of other techniques (e.g., micro benchmarks) in concert with analytic models. Sometimes the results of an evaluation using only analytic models might be invalidated by studies using other techniques. Often the different evaluations tend to reinforce one another, demonstrating the validity of the modeler’s conclusions.

**Self review**

1. Explain the relative merits of complex and simple analytic models.
2. What are some benefits of using analytic modeling?

**Ans: 1)** complex analytic models are more accurate, but it might not be possible to find a mathematical solution to model a system’s behavior. It is easier to represent the behavior of a system with a simpler model, but it might not accurately represent the system. 2)There is a large body of results that evaluators can draw from when creating models; analytic models can provide accurate and quick performance results; and they can be modified relatively easily when the system changes.

**Benchmarks**

**A benchmark** is a program that is executed to evaluate a machine. Commonly, a benchmark is a **production program** which is typical of many jobs, at the installation. The evaluator is thoroughly familiar with the performance of the benchmark on existing equipment, so when it is run on new equipment, the evaluator can draw meaningful conclusions. Several organizations, such as **Standard Performance Evaluation Corporation (SPEC;** [www.specbench.org](http://www.specbench.org)) and **Business Application Performance Corporation (BAPCo;** [www.bapco.com](http://www.bapco.com)), have developed industry. Standard benchmarks targeted for different systems (e.g., Web servers or personal computers). Evaluators can run these benchmarks to compare similar systems from different vendors.

One advantage of benchmarks is that many already exist, so the evaluator merely needs to choose them from known production programs or use industry-standard benchmarks. No timings are taken on individual instructions. Instead, the full program is run on the actual machine using the real data, so the computer does most of the work. The chance of human error is minimal because the benchmark time is measured by the computer itself. In environments such as multiprogramming, timesharing, multiprocessing, database, data communications and real-time systems, benchmarks can be particularly valuable, because they run on the actual machine in real circumstances. The effects of such complex systems can be experienced directly mislead of being estimated.

Several criteria should be considered when developing a benchmark. First, the results should be repeatable; specifically, every run of the benchmark program on a certain system should produce nearly the same result. Results do not have to be identical, and seldom are, because they can be affected by environment-specific details, such as an item is stored on disk. Second, benchmarks should accurately reflect the types of applications that will be executed on the system. Finally, the benchmark should be widely used so that more accurate comparisons can be made between systems. A good industry-standard benchmark will have all these properties; however, the latter two often lead to conflicting design decisions. A benchmark that is specific to a certain system might not be widely used; a benchmark that is designed to test multiple systems might not yield as accurate a result for specific system.

Benchmarks are useful in evaluating hardware as well as software, even under complex operating environments. They also are particularly useful in comparing the operation of a system before and after certain changes are made. Benchmarks are not useful, however, in predicting the effects of proposed changes, unless another system exists with the changes incorporated and on which the benchmarks can be run.

Benchmarks are probably the technique most widely used by organizations and consumers when determining which equipment to purchase from several computing vendors. Their popularity as tools for this purpose led to the need for industry-standard benchmarks. SPEC was founded in 1988 to promote the development of standard, relevant benchmarks. SPEC publishes a variety of benchmarks (often called **SPECmarks**) that can be used to evaluate the system ranging from servers to Java Virtual Machines and publishes performance results obtained using those SPECmarks for thousands of commercial systems. SPEC ratings can be useful for making an informed decision determining which computer components to purchase. However, one must first carefully determine the focus of each SPECmark test to evaluate particular platforms. For example, SPECweb measure systems on which Web servers typically execute and should not be used to compare systems that execute in different environments. In the environment of a particular Web server differs from a “typical” one, the SPEC rating might not be relevant. Furthermore, some of SPEC’s benchmarks have been criticized for narrow scope, especially by those who question the assumption that a “typical” workload can accurately approximate a real-world workload for a particular system. To combat such limitations, SPEC continually redesigns its benchmarks to improve their relevance to current systems.

Although SPEC produces some of the best-known benchmarks, there are a number of other popular benchmarks and benchmarking organizations. BAPCo produces several benchmarks, including the popular **SYSmark**benchmarks (for desktop systems, **MobileMark**(for systems installed on mobile devices) and **WebMark**(for internet performance). Other popular benchmarks include the **Transaction Processing Performance Council (TPC) benchmarks**, which target database systems, and the **Standard Application (SAP) benchmarks**, which evaluate a system’s scalability.

**Self review**

1. How can benchmarks be used to anticipate the effect of proposed changes to a system?
2. Why are there no universally accepted “standard” benchmarks?

**Ans: 1)** In general, benchmarks are effective only for determining the result after a change or for performance comparisons between systems, not for anticipating the effect of proposed changes to a system. However, if the proposed changes match the configuration of an existing system, running the benchmark on that system would be useful. **2)** benchmarks are real programs that are run on real machines, but each machine might contain a different set of hardware that runs a different mix of programs. Therefore, benchmark designers provide “typical” application mixes that are updated regularly to better approximate environments.

**Synthetic programs**

**Synthetic programs** (also called **synthetic benchmarks**) are similar to benchmarks, except that they focus on a specific component of the system, such as the I/O subsystem and memory subsystem. Unlike benchmarks, which are typical of real applications, evaluators construct synthetic programs for specific purposes. For example, a synthetic program might target an operating system component (e.g., the file system) or might be constructed to match the instruction frequency distribution of a large set of programs. One advantage of synthetic programs is that they can isolate specific components of a system rather than test the entire system.

Synthetic programs are useful in development environments. As new features become available, synthetic programs can be used to test that they are operational. Evaluators, unfortunately, do not always have sufficient time to code and debug synthetic programs, so they often seek existing benchmark programs that match the desired characteristics of a synthetic program as closely as possible. Evaluators can use synthetic programs with benchmarks and microbenchmarks for a thorough system evaluation. These three techniques provide different levels of abstraction (the system as a whole, a component of the system or a simple primitive) that, combined, give the evaluator an understanding of the performance both of the entire system and of individual parts of the system.

Although no longer used much, the **Whetstone** and the **Dhrystone** are examples of classic synthetic programs. The Whetstone measures how well systems handle floating point calculations, and was thus helpful in evaluating scientific programs. The Dhrystone measures how effectively an architecture runs systems programs. Because the Dhrystone consumes only a small amount of memory, its effectiveness is particularly sensitive to the size of a processor’s cache; if the Dhrystone’s data and instruction can fit in the cache, it will execute much faster than if the processor must access main memory while it executes. In fact, because the Dhrystone fits in most of today’s processor caches, in effect it measures a processor’s clock speed and provides no insight into how a system manages memory. One popular synthetic program used extensively today is **WinBench 99,**whichtestsa system’s graphics, disk and video subsystems in a Microsoft Windows environment. Other popular synthetic benchmarks include **IOStone**(which tests file systems),**Hartstone**(for real-time systems) and **STREAM** (for the memory subsystem).

**Self review**

1. Explain why synthetic programs are useful for development environments.

2. Should synthetic programs alone be used for a performance evaluation? Why?

**Ans: 1)** Synthetic programs can be written fairly quickly and can test specific features for correctness. **2)** No, synthetic programs are “artificial” programs used to test specific components or to characterize a large set of programs (but not one in particular). Therefore, although producing valuable results, they do not necessarily describe how the entire system will perform when executing real programs. It is usually a good idea to use a variety of performance evaluation techniques.

**Simulation**

**Simulation** is a technique in which an evaluator develops a computerized model of the system being evaluated. The evaluator tests the model, which presumably reflects the target system, to infer performance data about this system.

With simulation it is possible to prepare a model of a system that does not exist and then run the model to see how the system might behave in certain circumstances. Of course, the simulation must eventually be **validated** against the real system to improve that it is accurate. Simulations can highlight problems early in a system’s development cycle. Computerized simulation industries, because of the severe consequences of building systems that fail.

**. Event-driven simulators** – These are controlled by events that are made to occur in the simulator according to probability distributions.

**. Script-driven simulators** – These are controlled by data carefully manipulated to reflect the anticipated environment of the simulated system; evaluators derive this data from empirical observations.

Simulation requires considerable expertise on the part of evaluator and can consume substantial computer time. Simulators generally produce huge amounts of data that must be carefully analyzed. However, once simulators are developed, they can be reused effectively and economically.

Just as with analytic models, it is difficult to model a complex system exactly with a simulation. Several common errors cause most inaccuracies. Bugs in simulators will, of course, produce erroneous performance results. Deliberate omissions, resulting from the necessity of simplifying a simulation, can invalidate results as well. This is usually more of a problem for simple simulators. However, complex simulators suffer from the third problem-lack of detail. Complex simulators attempts to model all parts of the system, but inevitably, they do not model all details exactly. The attendant errors also can hinder a simulator’s effectiveness. Therefore, to achieve the most accurate performance results, it is important to validate a simulation against the real system.

**Self review**

1. Which produces more consistent results, event-driven or script-driven simulators?
2. (T/F) complex simulators are always more effective than simpler ones.

**Ans: 1)** Script-driven simulators produce nearly the same result each run because the system always uses the same inputs. Because event-driven simulators dynamically generate input based on probabilities, the results are less consistent. **2)** false. Although complex simulators attempt to model a system more completely, they still might not model it accurately.

**Performance Monitoring**

Performance monitoring is the collection and analysis of information regarding system performance for existing systems. It can help determine useful performance measures, such as throughput, response times and predictability. Performance monitoring can locate inefficiencies quickly and help system administrators decide how to improve system performance.

Users can monitor performance using software or hardware techniques. Software monitors might distort performance readings, because the monitors themselves consume system resources. Familiar examples of performance monitoring software and Microsoft Windows’s Task Manager and the Linux proc file system. Hardware monitors are generally more costly, but they have little or no impact on system performance. Many of today’s processors maintain several counting registers useful for performance monitoring that record events including clockticks, TLB misses and memory operations ( such as a write to main memory).

Monitors generally produce huge volumes of data that must be analyzed, possibly requiring extensive computer resources. However, they indicate precisely how the system is functioning, and this information can be extremely valuable. This is particularly true in development environments in which key design decisions might be based on the operation of the system. Instruction execution traces, or module execution traces, can reveal which areas of a system are most frequently used. A module execution trace might show, for example, that a small subset of modules is being used a large percentage of the time. If designers concentrate their optimization efforts on these modules, they might be able to improve system performance without expending effort and resources on infrequently used portions of the system. Figure 14.1 summarizes performance evaluation techniques.

**Self review**

1. Why do software performance monitors influence a system more than hardware performance monitors?
2. Why is performance monitoring important?

**Ans:1)** Software-based performance monitors must compete for system resources that would otherwise be allocated to programs that are being evaluated. This can result in inaccurate performance measurements. Hardware performance monitors operate in parallel with other system hardware, so that measurement does not affect system performance. **2)** Performance monitoring enables administrators to identify inefficiencies, using data describing how the system is functioning.

**Technique Description**

|  |  |
| --- | --- |
| Trace | Record of real application requests to the operating system, which identifies a system’s workload. |
| Profile | Record of kernel activity taken during a real session. Profiles indicate the relative usage of operating system primitives. |
| Timing | Raw measure of hardware performance, which can be used for quick comparisons between related systems. |
| Micro benchmarks | Raw measure of how quickly an operating system performs an isolated operation. |
| Application-specific evaluation | Evaluation that determines how efficiently a system executes a particular application. |
| Analytic modeling | Technique in which an evaluator builds and analyzes a mathematical model of a computer system. |
| Benchmark | Program typical of one that will be run on the given system, used for comparisons between systems. |
| Synthetic program | Programs that isolates the performance of a particular operating system component. |
| Simulation | Technique in which a computer model of the system is evaluated. The results of the simulation must be validated against the actual system once the system is built. |
| Performance monitoring | Ongoing evaluation of a system once it is installed, allowing administrators to assess whether it is meeting its demands and to determine which areas of its performance require improvement. |

***Summary of performance evaluation techniques.***

**Bottlenecks and saturation**

Operating systems manage collections of resources that interface and interact in complex ways. Occasionally, resources become **bottlenecks,** limiting the system’s overall performance, because they perform their designated tasks slowly relative to other resources. While other system resources might have excess capacity, the bottlenecks cannot pass jobs or processes to these other resources fast enough to keep them busy.

A bottleneck tends to develop at a resource when the traffic there begins to approach its capacity. We say that the resource becomes **saturated** – i.e., processes competing for its attention begin to interfere with one another, because one must wait for others to complete using the resource. A classic example is a virtual memory system that is thrashing (see chapter 11, Virtual Memory Management). This occurs in paged systems when main memory becomes overcommitted and the working set of the various active processes cannot be maintained simultaneously in main memory. An active process interferes with another process’s use of memory by forcing the system to flush some of the other process’s working set to secondary storage to make room for its own working set. Saturation can be dealt with by reducing the load on the system – e.g., thrashing can be eliminated by temporarily suspending less critical, noninteractive processes, if such processes are available.

How can bottlenecks be detected? Quite simply, each resource’s request queue should be monitored. When a queue begins to grow quickly, then the **arrival rate** of requests at that resource must be larger than its **service rate**, so the resource has become saturated.

Isolation of bottlenecks is an important part of tuning a system. The bottlenecks can be removed by increasing the capacity of the resources, by adding more resources of that type at the point in the system, or, again, by decreasing the load on those resources. Removing a bottleneck does not always improve throughput, however, because other bottlenecks might also exist in the system. Tuning a system often involves identifying and eliminating bottlenecks until system performance reaches satisfactory levels.

**Self review**

1. Why is it important to identify bottleneck in a system?
2. Thrashing is due to the saturation of what resource? How would an operating system detect thrashing?

**Ans: 1)** identifying bottlenecks allows designers to focus on optimizing sections of a system that are degrading performance. **2)**main memory. The operating system would notice a high page recapture rate – the pages being paged out to make room for incoming pages would themselves quickly be paged back into main memory.

**Feedback loops**

**A feedback loop** is a technique in which information about the current state of the system can affect arriving requests. These requests can be rerouted if the feedback indicates that the system might have difficulty servicing them. Feedback can be negative, in which case arrival rates might decrease, or positive, in which case arrival rates might increase. Although we divide this section to examine positive and negative feedback situations separately, the do not represent two different techniques. Rather, a request rate at a particular resource might cause either negative or positive feedback (or neither).

**Negative feedback**

In **negative feedback** situations, the arrival rate of new requests might decrease as a result of information being fed back. For example, a motorist pulling into a gas station and observing that several cars are waiting at each pump might decide to drive down the street to less crowded station.

In distributed systems, spooled output can often be printed by any of several equivalent print servers. If the queue behind one server is too long, the job may be placed in a less crowded queue.

Negative feedback contributes to **stability** in queuing systems. If the scheduler assigns arriving jobs indiscriminately to a busy device, for example, the queue behind that device might grow indefinitely (even though other devices might be underutilized).

**Self review**

1. Explain how negative feedback could improve read performance in level one (mirrored) RAID.
2. How does negative feedback contributes to system stability?

**Ans: 1)** if one of the disks in a mirrored pair contains a large queue of read requests, some of these requests could be sent to the other disk in the pair if it contains a smaller queue. **2)** Negative feedback prevents one resource from being overwhelmed while other identical resources lie idle.

**Positive feedback**

In **positive feedback** situations, the arrival rate of new requests might increase as a result of information being fed back. A classic example occurs in paged virtual memory multiprocessor systems. Suppose the operating system detects that a processor is underutilized. The system might inform the process scheduler to admit more processes to that processor’s queue, anticipating that this would place the greater load on the processor. As more processes are admitted, the amount of memory that can be allocated to each process decreases and page faults might increase (because the working sets of all active processes might not fit in memory). Processor utilization will actually decrease as the system thrashes. A poorly designed operating system might then decide to admit even more processes. Of course, this would cause further deterioration in processor utilization.

Operating systems designers must be cautious when designing mechanisms to respond to positive feedback to prevent such unstable situation from developing. The effects of each incremental change should be monitored to see whether it results in the anticipated improvement. If a change causes performance to deteriorate, this signals to the operating system that it might be entering an unstable range and that resource-allocation strategies should be adjusted in order to resume stable operation.

**Self review**

1. In some large server systems, users communicate requests to a “front-end” server. This server accepts the user’s request and sends it to a “back-end” server for processing. How could a front-end server balance request loads among a set of equivalent back-end servers using feedback loops?
2. This section describes how positive feedback can intensify thrashing. Suggest one possible solution to this problem.

**Ans: 1)** Back-end servers with a long queue of request might send negative feedback to the front-end server, and back-end servers that are idle might send positive feedback. The front-end server can use these feedback loops to send incoming requests to underloaded instead of overloaded servers.**2)** the system can monitor the number of page recaptures and refuse to admit further processes beyond a certain threshold.

**Performance Techniques in Processor Design**

A system’s performance depends heavily on the performance of its processors. Comceptually, a processor can be divided into its **instruction set**, which is the set of machine instructions it can perform, and its implementation, which is the physical hardware. An instruction set might be composed of a few basic instructions that perform only simple functions, such as loading a value from memory into a register or adding two numbers. Alternatively, instruction sets can contain an abundance of more complex instructions such as those that solve a specified polynomial equation. The penalty for providing such a large number of complex instructions is more complex hardware, which increases processor cost and might reduce performance for simpler instructions; the benefit is that these complex routines can be performed quickly.

The **Instruction Set Architecture (ISA)** of a processor is an interface that describes the processor, including its instruction set, number of registers and memory size. The ISA is the hardware equivalent to An operating system API.

Although a particular ISA does not specify the hardware implementation, the elements of an ISA directly affect how the hardware is constructed and therefore significantly impact performance. Approaches to ISAs have evolved over the years. This section investigates these approaches and evaluates how ISA design decisions affect performance.

**Self review**

1. What trade-offs must be weighed when including single instructions that perform complex routines in an instruction set?
2. Why is the choice of an ISA important?

**Ans: 1)** the penalty for adding these instructions is more complex hardware, which might slow the execution of other more frequently used instructions; the benefit is faster execution of these complex routines. **2)**the ISA specifies a programming interface between hardware and low-level software; therefore, it affects how easily code can be generated for the processor and how much memory that code occupies. Also, the ISA directly affects processor hardware, which influences cost and performance.

**Complex Instruction Set Computing (CISC)**

Until the mid-1980s, the clear trend was to incorporate frequently used sections of code into single machine-language instructions, with the hope of making these functions faster to execute and easier to code in assembly language. The logic was appealing, judging by the number of ISAs that reflected these greatly expanded instruction sets – an approach that has been named **complex instruction set computing (CISC)**, partly echoing the popular term reduced instruction set computing (RISC), which we discuss in the next section.

CISC processors originated when most systems were written in assembly language. An instruction set including single instructions that each perform several operations enabled assembly-language programmers to write their programs with fewer lines of code, thus improving programmer productivity. CISC continued to be attractive when high-level languages became widely used for writing operating systems (such as the use of C and C++ in UNIX source code), because special-purpose instructions were added to fit well with the needs of optimizing compilers. Optimizing compilers alter the structure of compiled code (but not the semantics) to achieve higher performance for a particular architecture. CISC instructions mirrored the complex operations of high-level languages rather than the simple operations a processor could execute in one or two clock cycles. These complex instruction sets tended to be implemented via microprogramming. Microprogramming introduces a layer of programming below a computer’s machine language; this layer specifies the actual primitive operations that a processor must perform, such as fetching an instruction from memory (see Section 2.9 for a further description). In these CISC architectures, the machine-language instructions were interpreted, so that complex instructions were performed as a series of simpler microprogrammed instructions.

CISC processors became popular largely in response to the decreasing cost of hardware coupled with the increasing cost of developing software with assembly language. CISC processors attempt to move most of the complexity from the software to the hardware. As a side effect, CISC processors also reduce the size of programs, saving memory and easing debugging. Another characteristic of some CISC processors is a trend toward reducing the number of general-purpose registers, to decrease cost and increase space available to other CISC structures, such as the instruction decoder.

One powerful technique to increase performance that developed during the CISC era was pipelining. A pipeline divides a processor’s datapath (i.e., the portion of the processor that performs operations on data) into discrete stages. For every clock cycle, a maximum of one instruction can occupy each stage, allowing a processor to perform operations on several instructions simulations simultaneously. In the early1960s, IBM developed the first pipelined processor, the IBM 7030 (nicknamed “stretch”). The 7030’s pipeline consisted of four stages: instruction fetch, instruction decode, operand fetch and execution. While the processor was executing one instruction, it fetched the operands for the next instruction, decoded another instruction and fetched a fourth instruction. After each clock cycle began, each instruction in the pipeline would move forward one stage; this permitted the 7030 to process up to four instructions at once, which improved performance significantly.

As processor manufacturing technology has improved, chip size and memory bandwidth have become less of a concern. Moreover, advance compilers can easily perform many optimization techniques previously delegated to the implementation of the CISC instruction set. However, CISC processors still are popular in many personal computers; they also can be found in several high-performance computers. Intel Pentium ([www.intel.com/products/desktop/processors/pentium4/](http://www.intel.com/products/desktop/processors/pentium4/)) and Advanced Micro Devices (AMD) Athlon ([www.amd.com/us-en/processors/productinformation/o,,30\_118\_756,00.html](http://www.amd.com/us-en/processors/productinformation/o,,30_118_756,00.html)) processors are CISC processors.

**Self review**

1. (T/F) The widespread use of high-level programming languages eliminated the usefulness of complex instructions.
2. What was the motivation behind the CISC processor-design philosophy?

**Ans: 1)** False. Additional complex instructions were created to fit well with the needs of optimizing compilers. **2)** Early operating systems were written primarily in assembly code, so complex instructions simplified programming, because each instruction performed several operations.

**Reduced Instruction Set Computing (RISC)**

Many of the advantages of CISC processors were rendered unimportant by advances in hardware technology and compiler design. In addition, designers realized that the microcode required to interpret complex instructions slowed the execution of the simpler instructions. Various studies that a large majority of programs generated by popular compilers used only a small portion of their target processors’ instruction set. For example, an IBM study observed that the 10 most frequently executed instructions of the hundreds in the System/370 architecture (Fig.14.2) accounted for two-thirds of the instruction executions on that machine. The additional instructions incorporated into CISC instruction sets to improve software development time were being used only infrequently. The simplest instructions to implement, such as register-memory transfers (loads and stores), were the most commonly used.

Another IBM study, a static analysis of assembly-language programs written for the IBM Series/1 computer, provided more evidence that large CISC instruction sets might be inefficient. The study observed that programmers tended to generate “semantically equivalent instruction sequences” when working with the rich machine language of the IBM Series/1. The authors concluded that since it is difficult to detect such sequences automatically, either instruction sets should be made sparser, or programmers should restrict their use of instructions. These observations provided compelling motivation for the notion of **reduced instruction set computing (RISC).** This processor-design philosophy emphasizes that computer architects can optimize performance by concentrating their efforts on making common instructions, such as branches, loads and stores, execute efficiently.

RISC processors execute common instructions efficiently by including relatively few instructions, most of which are simple and can be performed quickly (i.e., in one clock cycle).this means that much of the programming complexity is moved from the hardware to the compiler, which permits RISC processor design to be simple and to optimize for a small set of instructions. Furthermore, RISC control units are implemented in hardware, which reduces execution overhead compared to microcode. RISC instructions, which occupy fixed-length words in memory, are faster and easier to decode than variable-length CISC instructions. Additionally, RISC processors attempt to reduce number of accesses to main memory by providing many high-speed general-purpose registers in which programs can perform data manipulation.

Providing only simple, uniform-length instructions permits RISC processors to make better use of pipelines than CISC processors do. Pipelined execution in CISC architectures can be slowed by the longer processing times of complex instructions; these have the effect of idling sections of the pipeline handling simplex instructions.

**Opcode instruction % of execution**

|  |  |  |
| --- | --- | --- |
| BC | Branch condition | 20.2 |
| L | Load | 15.5 |
| TM | Test Under Mask | 6.1 |
| ST | Store | 5.9 |
| LR | Load Register | 4.7 |
| LA | Load Address | 4.0 |
| LTR | Test Register | 3.8 |
| BCR | Branch Register | 2.9 |
| MVC | Move Characters | 2.1 |
| LH | Load Half Word | 1.8 |

*Figure 14.2 |the 10 most frequently executed instructions on IBM’s System/370 architecture. (courtesy of International Business Machines Corporation.)*

Additionally, pipelines on CISC machines often contain many stages and require complex hardware to support instructions of various lengths. Pipelines in RISC architectures contain few stages and are relatively straightforward to implement because most instructions require one cycle.

A simple and clever optimizing technique used in many RISC architectures is called **delayed branching.** When a conditional branch is executed, the next sequential instruction might or might not be executed next, depending on the evaluation of the condition. Delayed branching enables this next sequential instruction to enter execution anyway; then, if the branch is not taken, this next instruction can be well along, if not completely finished. RISC optimizing compilers often rearrange machine instructions so that a useful calculation ( one that must be executed regardless of whether the branch is taken) is placed immediately after the branch. Because branching occurs more frequently than most people realize (as often as every fifth instruction on some popular architectures), this can yield considerable performance gains.Lilja provides a thorough analysis of delayed branching and several other techniques that can greatly reduce the so-called **branch penalty** in pipelined architectures.

The most significant trade-off in the RISC design philosophy in that it is simple, reduced instruction set and register-rich architecture increase context-switching complexity. RISC architectures must save a large number of registers to main memory during a context switch; this reduces context-switching performance compared to CISC architectures due to an increased number of accesses to main memory. Because context switching is pure overhead and occurs frequently, this can significantly impact system performance.

In addition to increased context-switch time, there are other drawbacks to the RISC design. One interesting study tested the effect of instruction complexity on performance; the study selected three increasingly complex instruction sunsets of the VAX. The researchers reached several conclusions.

**1.**programs written in the simplest of the instruction subsets required as much as 2.5 times the memory of equivalent complex instruction set programs.

**2.** the cache miss ratio was considerably larger for programs written in the simplest subset.

**3.**the bus traffic for programs written in the simplest subset was about double that of programs written in the most complex subset.

These results and others like them indicate that RISC architectures can have negative consequences.

Floating point operations execute faster in CISC architectures. Additionally, CISC processors perform better for graphics or scientific program, which repeatedly execute complex instructions; these programs tend to perform better on CISC machines with optimized complex instructions than on comparable RISC machines.

Figure 14.3 provides a summary comparison between the RISC and CISC design philosophies. Examples of RISC processors include the ultraSPARC

***Category Characteristics of CISC characteristi-***

***processors cs of RISC***

processors

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

**Self Review**

1. Why do RISC processors exploit pipelines better than CISC processors?
2. Why context switching does require more overhead on RISC than on CISC processors?

**Ans: 1)** RISC instructions are of fixed length and generally require one cycle to execute, so it is easy to overlap instructions such that most stages in the pipeline are performing meaningful work. The variable-length instructions of CISC processors have the effect of idling sections of the pipeline not needed for simple instructions. **2)** RISC processors contain many more registers

Revision Exercises

1 Why are RISC programs generally longer than their CISC equivalents? Given this, why might RISC programs execute faster than their CISC equivalents?

2 Prepare a research paper surveying contemporary studies that compare RISC performance to CISC performance. Describe the strengths and weaknesses of each design philosophy in today’s systems.

3 A weakness of the RISC approach is a dramatic increase in context-switching overhead. Give a detailed explanation of why this is so. Write a paper on context switching. Discuss the various approaches that have been used. Suggest how context switching might be handled efficiently in RISC-based systems

4. You are a member of a performance evaluation team working for a computer manufacturer. You have been given the task of developing a generalized synthetic program to facilitate the evaluation of a completely new computer system with an innovative instruction set.

a) Why might such a program be useful?

b) What features might you provide to make your program a truly general evaluation tool?

5 On one computer system, the processor contains a BIPS meter that records how many billion instructions per second the processor is performing at any instant in time. The meter is calibrated from 0 to 4 BIPS in increments of 0.1 BIPS. All of the workstations to this computer are currently in use. Explain how each of the following situations might occur.

a) The meter reads 3.8 BIPS and the terminal users are experiencing good response times.

b) The meter reads 0.5 BIPS and the terminal users are experiencing good response times

c) The meter reads 3.8 BIPS and the terminal users are experiencing poor response times.

d) The meter reads 0.5 BIPS and the terminal users are experiencing poor response time

6 What Performance evaluation techniques are most applicable in each of the following situations? Explain your answers

a) An insurance company has a stable workload consisting of a large number of lengthy batch-processing production runs. Because of a merger, the company must increase its capacity by 50 %. The company wishes to replace its equipment with a new computer system.

b) The insurance company described in a) wishes to increase capacity by purchasing some additional memory and channels.

c) A computer company is designing a new, ultrahigh speed computer system and wishes to evaluate several alternative designs.

d) A consulting firm that specializes in commercial data processing gets a large military contract requiring extensive mathematical calculations. The company wishes to determine if its existing computer equipment will process the anticipated load of mathematical calculations

e) Management in charge of a multicomputer network needs to locate bottlenecks as soon as they develop and to reroute traffic accordingly.

f) A system programmer suspects that one of the software modules is being called upon more frequently than originally anticipated. The programmer wants to confirm this before devoting substantial effort to recording the module to make it execute more efficiently.

g) An operating systems vendor needs to test all aspects of the system before selling its product commercially.

h) Explain why it is important to monitor and evaluate the performance of a system’s software as well as its hardware.

i) When a user logged in, some early timesharing systems printed the total number of logged in users.

1. Why was this information useful to the user?

2. In what circumstances might this not have been a useful indication of load?

3 What factors tended to make this a highly reliable indication of system load on a time sharing system that supported many users?

j) Briefly discuss each of the following purposes for performance evaluation.

1 Selection evaluation

2 Performance projection

3 performance monitoring

k) What is system tuning? Why is it important

l) in discussing random variables, why can mean values sometimes be deceiving? What other performance measure is useful in describing how closely the values of a random variables cluster about it mean?

m) Simulation is viewed by many as the most widely applicable performance evaluation technique.

1 Give several reasons for this.

2 Even though simulation is widely applicable, it is not as widely used as one might expect. Give several reasons why.