SYSTEM MONITORING

The term *system* refers to a computer system that is composed of hardware and software for data processing. *System monitoring* collects information about the behavior of a computer system while the system is running. What is of interest here is run-time information that cannot be obtained by static analysis of programs. All collected information is essentially about system correctness or performance. Such information is vital for understanding how a system works. It can be used for dynamic safety checking and failure detection, program testing and debugging, dynamic task scheduling and resource allocation, performance evaluation and tuning, system selection and design, and so on.

COMPONENTS AND TECHNIQUES FOR SYSTEM MONITORING

System monitoring has three components. First, the jobs to be run and the items to be measured are determined. Then, the system to be monitored is modified to run the jobs and take the measurements. This is the major component. Monitoring is accomplished in two operations: triggering and recording (1). *Triggering,* also called *activation,* is the observation and detection of specified events during system execution. *Recording* is the collection and storage of data pertinent to those events. Finally, the recorded data are analyzed and displayed.

The selection and characterization of the jobs to be run for monitoring is important, because it is the basis for interpreting the monitoring results and guaranteeing that the experiments are repeatable. A collection of jobs to be run is called a *test workload* (2–4); for performance monitoring, this refers mainly to the *load* rather than the *work,* or *job.* A workload can be real or synthetic. A *real workload* consists of jobs that are actually performed by the users of the system to be monitored. A *synthetic workload,* usually called a *benchmark,* consists of batch programs or interactive scripts that are designed to represent the actual jobs of interest. Whether a workload is real or synthetic does not affect the monitoring techniques.

Items to be measured are determined by the applications. They can be about the entire system or about different levels of the system, from user-level application programs to operating systems to low-level hardware circuits. For the entire system, one may need to know whether jobs are completed normally and performance indices such as job completion time, called *turnaround time* in batch systems and *response time* in interactive systems, or the number of jobs completed per unit of time, called *throughput* (3). For application programs, one may be interested in how often a piece of code is executed, whether a variable is read between two updates, or how many messages are sent by a process. For operating systems, one may need to know whether the CPU is busy at certain times, how often paging occurs, or how long an I/O operation takes. For hardware circuits, one may need to know how often a cache element is replaced, or whether a network wire is busy.

Monitoring can use either event-driven or sampling techniques (3). *Event-driven monitoring* is based on observing changes of system state, either in software programs or hardware circuits, that are caused by events of interest, such as the transition of the CPU from busy to idle. It is often implemented as special instructions for interrupt–intercept that are inserted into the system to be monitored. *Sampling monitoring* is based on probing at selected time intervals, into either software programs or hardware circuits, to obtain data of interest, such as what kinds of processes occupy the CPU. It is often implemented as timer interrupts during which the state is recorded. Note that the behavior of the system under a given workload can be simulated by a simulation tool. Thus monitoring that should be performed on the real system may be carried out on the simulation tool. Monitoring simulation tools is useful, or necessary, for understanding the behavior of models of systems still under design.

Monitoring can be implemented using software, hardware, or both (1,3–5). *Software monitors* are programs that are inserted into the system to be monitored. They are triggered upon appropriate interrupts or by executing the inserted code. Data are recorded in buffers in the working memory of the monitored system and, when necessary, written to secondary storage. *Hardware monitors* are electronic devices that are connected to specific system points. They are triggered upon detecting signals of interest. Data are recorded in separate memory independent of the monitored system. *Hybrid monitors* combine techniques from software and hardware. Often, the triggering is carried out using software, and the data recording is carried out using hardware.

The data collected by a monitor must be analyzed and displayed. Based on the way in which results are analyzed and displayed, a monitor is classified as an on-line monitor or a batch monitor. *On-line monitors* analyze and display the collected data in real-time, either continuously or at frequent intervals, while the system is still being monitored. This is also called *continuous monitoring* (6). *Batch monitors* collect data first and analyze and display them later using a batch program. In either case, the analyzed data can be presented using many kinds of graphic charts, as well as text and tables.

ISSUES IN SYSTEM MONITORING

Major issues of concern in monitoring are at what levels we can obtain information of interest, what modifications to the system are needed to perform the monitoring, the disturbance of such modifications to the system behavior, and the cost of implementing such modifications. There are also special concerns for monitoring real-time systems, parallel architectures, and distributed systems.

Activities and data structures visible to the user process can be monitored at the application-program level. These include function and procedure calls and returns, assignments to variables, loopings and branchings, inputs and outputs, as well as synchronizations. Activities and data structures visi-

J. Webster (ed.), *Wiley Encyclopedia of Electrical and Electronics Engineering.* Copyright © 1999 John Wiley & Sons, Inc.

ble to the kernel can be monitored at the operating-system level; these include system state transitions, external interrupts, system calls, as well as data structures such as process control blocks. At the hardware level, various patterns of signals on the buses can be monitored. Obviously, certain high-level information cannot be obtained by monitoring at a lower level, and vice versa. It is worth noting that more often high-level information can be used to infer low-level information if one knows enough about all the involved components, such as the compilers, but the converse is not true, simply because more often multiple high-level activities are mapped to the same low-level activity.

In general, the software and hardware of a system are not purposely designed to be monitored. This often restricts what can be monitored in a system. To overcome these restrictions, modifications to the system, called *instrumentation,* are often required, for example, inserting interrupt instructions or attaching hardware devices. The information obtainable with a monitor and the cost of measurements determine the *measurability* of a computer system (3,7). At one extreme, every system component can be monitored at the desired level of detail, while at the other extreme, only the external behavior of the system as a whole can be monitored. When a low degree of detail is required, a *macroscopic* analysis, which requires measurement of global indices such as turnaround time and response time, is sufficient. When a high degree of detail is needed, a *microscopic* analysis, which requires, say, the time of executing each instruction or loading each individual page, must be performed.

Monitoring often interferes with the system behavior, since it may consume system resources, due to the time of performing monitoring activities and the space of storing collected data, which are collectively called the *overhead* of monitoring. A major issue in monitoring is to reduce the perturbation. It is easy to see that a macroscopic analysis incurs less interference than a microscopic analysis. Usually, sampling monitoring causes less interference than event-driven monitoring. In terms of implementation, software monitors always interfere and sometimes interfere greatly with the system to be monitored, but hardware monitors cause little or no interference.

Implementing monitoring usually has a cost, since it requires modification to the system to be monitored. Therefore, an important concern is to reduce the cost. Software monitors are simply programs, so they are usually less costly to develop and easier to change. In contrast, hardware monitors require separate hardware devices and thus are usually more difficult to build and modify.

Finally, special methods and techniques are necessary for monitoring real-time systems, parallel architectures, and distributed systems. Real-time systems have real-time constraints, so interference becomes much more critical. For parallel architectures, monitoring needs to handle issues arising from interprocessor communication and scheduling, cache behavior, and shared memory behavior. For distributed systems, monitoring must take into account ordering of distributed events, message passing, synchronization, as well as various kinds of failures.

MONITORING PRINCIPLES

A set of principles is necessary to address all the issues involved in monitoring. The major task is to determine the monitoring techniques needed based on the applications and the trade-offs. Methods and tools that facilitate monitoring are also needed.

Consider the major task. Given the desired information, one first needs to determine all levels that can be monitored to obtain the information. For each possibility, one determines all modifications of the system that are needed to perform the monitoring. Then one needs to assess the perturbation that the monitoring could cause. Finally, one must estimate the cost of the implementations. Clearly, unacceptable perturbation or cost helps reduce the possibilities. Then, one needs to evaluate all possibilities based on the following trade-offs.

First, monitoring at a higher level generally requires less modification to the system and has smaller implementation cost, but it may have larger interference with the system behavior. Thus one principle is to monitor at the highest level whose interference is acceptable. This implies that, if a software monitor has acceptable interference, one should avoid using a hardware monitor. Furthermore, to reduce implementation cost, for a system being designed or that is difficult to measure, one can use simulation tools instead of the real system if credibility can be established.

Second, macroscopic analysis generally causes less perturbation to the system behavior than microscopic analysis, and it often requires less modification to the system and has smaller cost. Therefore, a second principle is to use macroscopic analysis instead of microscopic analysis if possible. While sampling is a statistical technique that records data only at sampled times, event detection is usually used to record all potentially interesting events and construct the execution trace. Thus one should avoid using tracing if the desired information can be obtained by sampling.

Additionally, one should consider workload selection and data analysis. Using benchmarks instead of real workload makes the experiments repeatable and facilitates comparison of monitoring results. It can also reduce the cost, since running real jobs could be expensive or impossible. Thus using benchmarks is preferred, but a number of common mistakes need to be carefully avoided (4). Data analysis involves a separate trade-off: the on-line method adds time overhead but can reduce the space overhead. Thus even when monitoring results do not need to be presented in an on-line fashion, on-line analysis can be used to reduce the space overhead and, when needed, separate processors can be used to reduce also the time overhead.

Finally, special applications determine special monitoring principles. For example, for monitoring real-time systems, perturbation is usually not tolerable, but a full trace is often needed to understand system behavior. To address this problem, one may perform microscopic monitoring based on event detection and implement monitoring in hardware so as to sense signals on buses at high speed and with low overhead. If monitoring results are needed in an on-line fashion, separate resources for data analysis must be used. Of course, all these come at a cost.

To facilitate monitoring, one needs methods and tools for instrumenting the system, efficient data structures and algorithms for storing and manipulating data, and techniques for relating monitoring results to the source program to identify problematic code sections. Instrumentation of programs can be done via program transformation, by augmenting the source code, the target code, the run-time environment, the operating system, or the hardware. Often, combinations of these techniques are used. Efficient data structures and algorithms are needed to handle records of various execution information, by organizing them in certain forms of tables and linked structures. They are critical for reducing monitoring overhead. Additional information from the compiler and other involved components can be used to relate monitoring results with points in source programs. Monitoring results can also help select candidate jobs for further monitoring.

In summary, a number of trade-offs are involved in determining the monitoring techniques adopted for a particular application. Tools should be developed and used to help instrument the system, reduce the overhead, and interpret the monitoring results.

WORKLOAD SELECTION

To understand how a complex system works, one first needs to determine what to observe. Thus before determining how to monitor a system, one must determine what to monitor and why it is important to monitor them. This enables one to determine the feasibility of the monitoring, based on the perturbation and the cost, and then allows repeating and justifying the experiments.

Selecting candidate jobs to be run and measurements to be taken depends on the objectives of monitoring. For monitoring that is aimed at performance behavior, such as system tuning or task scheduling, one needs to select the representative load of work. For monitoring that is aimed at functional correctness, such as for debugging and fault-tolerance analysis, one needs to isolate the "buggy" or faulty parts.

A real workload best reflects system behavior under actual usage, but it is usually unnecessarily expensive, complicated, or even impossible to use as a test workload. Furthermore, the test results are not easily repeated and are not good for comparison. Therefore, a synthetic workload is normally used. For monitoring the functional correctness of a system, a *test suite* normally consists of data that exercise various parts of the system, and monitoring at those parts is set up accordingly. For performance monitoring, the load of work, rather than the actual jobs, is the major concern, and the approaches below have been used for obtaining test workloads (3,4,8).

Addition instruction was used to measure early computers, which had mainly a few kinds of instructions. *Instruction mixes,* each specifying various instructions together with their usage frequencies, were used when the varieties of instructions grew. Then, when pipelining, instruction caching, and address translation mechanisms made computer instruction times highly variable, *kernels,* which are higher-level functions, such as matrix inversion and Ackermann's function, which represent services provided by the processor, came into use. Later on, as input and output became an important part of real workload, *synthetic programs,* which are composed of exerciser loops that make a specified number of service calls or I/O requests, came into use. For domain-specific kinds of applications, such as banking or airline reservation, *application benchmarks,* representative subsets of the functions in the application that make use of all resources in the system, are used. Kernels, synthetic programs, and application benchmarks are all called benchmarks. Popular benchmarks include the sieve kernel, the LINPACK benchmarks, the debit–credit benchmark, and the SPEC benchmark suite (4).

Consider monitoring the functional behavior of a system. For general testing, the test suite should have complete coverage, that is, all components of the system should be exercised. For debugging, one needs to select jobs that isolate the problematic parts. This normally involves repeatedly selecting more specialized jobs and more focused monitoring points based on monitoring results. For correctness checking at given points, one needs to select jobs that lead to different possible results at those points and monitor at those points. Special methods are used for special classes of applications; for example, for testing fault-tolerance in distributed systems, message losses or process failures can be included in the test suite.

For system performance monitoring, selection should consider the services exercised as well as the level of detail and representativeness (4). The starting point is to consider the system as a service provider and select the workload and metrics that reflect the performance of services provided at the system level and not at the component level. The amount of detail in recording user requests should be determined. Possible choices include the most frequent request, the frequency of request types, the sequence of requests with time stamps, and the average resource demand. The test workload should also be representative of the real application. Representativeness is reflected at different levels (3) at the physical level, the consumptions of hardware and software resources should be representative; at the virtual level, the logical resources that are closer to the user's point of view, such as virtual memory space, should be representative; at the functional level, the test workload should include the applications that perform the same functions as the real workload.

Workload characterization is the quantitative description of a workload (3,4). It is usually done in terms of workload parameters that can affect system behavior. These parameters are about service requests, such as arrival rate and duration of request, or about measured quantities, such as CPU time, memory space, amount of read and write, or amount of communication, for which system independent parameters are preferred. In addition, various techniques have been used to obtain statistical quantities, such as frequencies of instruction types, mean time for executing certain I/O operations, and probabilities of accessing certain devices. These techniques include averaging, histograms, Markov models, and clustering. *Markov models* specify the dependency among requests using a transition diagram. *Clustering* groups similar components in a workload in order to reduce the large number of parameters for these components.

TRIGGERING MECHANISM

Monitoring can use either event-driven or sampling techniques for triggering and data recording (3). Event-driven techniques can lead to more detailed and accurate information, while sampling techniques are easier to implement and have smaller overhead. These two techniques are not mutually exclusive; they can coexist in a single tool.

Event-Driven Monitoring

An *event* in a computer system is any change of the system's state, such as the transition of a CPU from busy to idle, the change of content in a memory location, or the occurrence of a pattern of signals on the memory bus. Therefore, a way of collecting data about system activities is to capture all associated events and record them in the order they occur. A software event is an event associated with a program's function, such as the change of content in a memory location or the start of an I/O operation. A hardware event is a combination of signals in the circuit of a system, such as a pattern of signals on the memory bus or signals sent to the disk drive.

Event-driven monitoring using software is done by inserting a special trap code or hook in specific places of the application program or the operating system. When an event to be captured occurs, the inserted code causes control to be transferred to an appropriate routine. The routine records the occurrence of the event and stores relevant data in a buffer area, which is to be written to secondary storage and/or analyzed, possibly at a later time. Then the control is transferred back. The recorded events and data form an *event trace.* It can provide more information than any other method on certain aspects of a system's behavior.

Producing full event traces using software has high overhead, since it can consume a great deal of CPU time by collecting and analyzing a large amount of data. Therefore, event tracing in software should be selective, since intercepting too many events may slow down the normal execution of the system to an unacceptable degree. Also, to keep buffer space limited, buffer content must be written to secondary storage with some frequency, which also consumes time; the system may decide to either wait for the completion of the buffer transfer or continue normally with some data loss.

In most cases, event-driven monitoring using software is difficult to implement, since it requires that the application program or the operating system be modified. It may also introduce errors. To modify the system, one must understand its structure and function and identify safe places for the modifications. In some cases, instrumentation is not possible when the source code of the system is not available.

Event-driven monitoring in hardware uses the same techniques as in software, conceptually and in practice, for handling events. However, since hardware uses separate devices for trigger and recording, the monitoring overhead is small or zero. Some systems are even equipped with hardware that makes event tracing easier. Such hardware can help evaluate the performance of a system as well as test and debug the hardware or software. Many hardware events can also be detected via software.

Sampling Monitoring

Sampling is a statistical technique that can be used when monitoring all the data about a set of events is unnecessary, impossible, or too expensive. Instead of monitoring the entire set, one can monitor a part of it, called a *sample.* From this sample, it is then possible to estimate, often with a high degree of accuracy, some parameters that characterize the entire set. For example, one can estimate the proportion of time spent in different code segments by sampling program counters instead of recording the event sequence and the exact event count; samples can also be taken to estimate how much time different kinds of processes occupy CPU, how much memory is used, or how often a printer is busy during certain runs.

In general, sampling monitoring can be used for measuring the fractions of a given time interval each system component spends in its various states. It is easy to implement using periodic interrupts generated by a timer. During an interrupt, control is transferred to a data-collection routine, where relevant data in the state are recorded. The data collected during the monitored interval are later analyzed to determine what happened during the interval, in what ratios the various events occurred, and how different types of activities were related to each other. Besides timer interrupts, most modern architectures also include hardware performance counters, which can be used for generating periodic interrupts (9). This helps reduce the need for additional hardware monitoring.

The accuracy of the results is determined by how representative a sample is. When one has no knowledge of the monitored system, random sampling can ensure representativeness if the sample is sufficiently large. It should be noted that, since the sampled quantities are functions of time, the workload must be stationary to guarantee validity of the results. In practice, operating-system workload is rarely stationary during long periods of time, but relatively stationary situations can usually be obtained by dividing the monitoring interval into short periods of, say, a minute and grouping homogeneous blocks of data together.

Sampling monitoring has two major advantages. First, the monitored program need not be modified. Therefore, knowledge of the structure and function of the monitored program, and often the source code, is not needed for sampling monitoring. Second, sampling allows the system to spend much less time in collecting and analyzing a much smaller amount of data, and the overhead can be kept less than 5% (3,9,10). Furthermore, the frequency of the interrupts can easily be adjusted to obtain appropriate sample size and appropriate overhead. In particular, the overhead can also be estimated easily. All these make sampling monitoring particularly good for performance monitoring and dynamic system resource allocation.

IMPLEMENTATION

System monitoring can be implemented using software or hardware. Software monitors are easier to build and modify and are capable of capturing high-level events and relating them to the source code, while hardware monitors can capture rapid events at circuit level and have lower overhead.

Software Monitoring

Software monitors are used to monitor application programs and operating systems. They consist solely of instrumentation code inserted into the system to be monitored. Therefore, they are easier to build and modify. At each activation, the inserted code is executed and relevant data are recorded, using the CPU and memory of the monitored system. Thus software monitors affect the performance and possibly the correctness of the monitored system and are not appropriate for monitoring rapid events. For example, if the monitor executes 100 instructions at each activation, and each instruction takes 1 μs, then each activation takes 0.1 ms; to limit the time overhead to 1%, the monitor must be activated at intervals of 10 ms or more, that is, less than 100 monitored events should occur per second.

Software monitors can use both event-driven and sampling techniques. Obviously, a major issue is how to reduce the monitoring overhead while obtaining sufficient information. When designing monitors, there may first be a tendency to collect as much data as possible by tracing or sampling many activities. It may even be necessary to add a considerable amount of load to the system or to slow down the program execution. After analyzing the initial results, it will be possible to focus the experiments on specific activities in more detail. In this way, the overhead can usually be kept within reasonable limits. Additionally, the amount of the data collected may be kept to a minimum by using efficient data structures and algorithms for storage and analysis. For example, instead of recording the state at each activation, one may only need to maintain a counter for the number of times each particular state has occurred, and these counters may be maintained in a hash table (9).

Inserting code into the monitored system can be done in three ways: (1) adding a program, (2) modifying the application program, or (3) modifying the operating system (3). Adding a program is simplest and is generally preferred to the other two, since the added program can easily be removed or added again. Also, it maintains the integrity of the monitored program and the operating system. It is adequate for detecting the activity of a system or a program as a whole. For example, adding a program that reads the system clock before and after execution of a program can be used to measure the execution time.

Modifying the application program is usually used for event-driven monitoring, which can produce an execution trace or an exact profile for the application. It is based on the use of *software probes,* which are groups of instructions inserted at critical points in the program to be monitored. Each probe detects the arrival of the flow of control at the point it is placed, allowing the execution path and the number of times these paths are executed to be known. Also, relevant data in registers and in memory may be examined when these paths are executed. It is possible to perform sampling monitoring by using the kernel interrupt service from within an application program, but it can be performed more efficiently by modifying the kernel.

Modifying the kernel is usually used for monitoring the system as a service provider. For example, instructions can be inserted to read the system clock before and after a service is provided in order to calculate the turnaround time or response time; this interval cannot be obtained from within the application program. Sampling monitoring can be performed efficiently by letting an interrupt handler directly record relevant data. The recorded data can be analyzed to obtain information about the kernel as well as the application programs.

Software monitoring, especially event-driven monitoring in the application programs, makes it easy to obtain descriptive data, such as the name of the procedure that is called last in the application program or the name of the file that is accessed most frequently. This makes it easy to correlate the monitoring results with the source program, to interpret them, and to use them.

There are two special software monitors. One keeps *system accounting logs* (4,6) and is usually built into the operating system to keep track of resource usage; thus additional monitoring might not be needed. The other one is *program execution monitor* (4,11), used often for finding the performance bottlenecks of application programs. It typically produces an execution profile, based on event detection or statistical sampling. For event-driven precise profiling, efficient algorithms have been developed to keep the overhead to a minimum (12). For sampling profiling, optimizations have been implemented to yield an overhead of 1% to 3%, so the profiling can be employed continuously (9).

Hardware Monitoring

With *hardware monitoring,* the monitor uses hardware to interface to the system to be monitored (5,13–16). The hardware passively detects events of interest by snooping on electric signals in the monitored system. The monitored system is not instrumented, and the monitor does not share any of the resources of the monitored system. The main advantage of hardware monitoring is that the monitor does not interfere with the normal functioning of the monitored system and rapid events can be captured. The disadvantage of hardware monitoring is its cost and that it is usually machine dependent or at least processor dependent. The snooping device and the signal interpretation are bus and processor dependent.

In general, hardware monitoring is used to monitor the run-time behavior of either hardware devices or software modules. Hardware devices are generally monitored to examine issues such as cache accesses, cache misses, memory access times, total CPU times, total execution times, I/O requests, I/O grants, and I/O busy times. Software modules are generally monitored to debug the modules or to examine issues such as the bottlenecks of a program, the deadlocks, or the degree of parallelism.

A hardware monitor generally consists of a probe, an event filter, a recorder, and a real-time clock. The probe is high-impedance detectors that interface with the buses of the system to be monitored to latch the signals on the buses. The signals collected by the probe are manipulated by the event filter to detect events of interest. The data relevant to the detected event along with the value of the real-time clock are saved by the recorder. Based on the implementation of the event filter, hardware tools can be classified as fixed hardware tools, wired program hardware tools, and stored program hardware tools (5,13).

With *fixed hardware tools,* the event filtering mechanism is completely hard-wired. The user can select neither the events to be detected nor the actions to be performed upon detection of an event. Such tools are generally designed to measure specific parameters and are often incorporated into a system at design time. Examples of fixed hardware tools are timing meters and counting meters. *Timing meters* or *timers* measure the duration of an activity or execution time, and *counting meters* or *counters* count occurrences of events, for example, references to a memory location. When a certain value is reached in a timer (or a counter), an electronic pulse is generated as an output of the timer (or the counter), which may be used to activate certain operations, for instance, to generate an interrupt to the monitored system.

Wired-program hardware tools allow the user to detect different events by setting the event filtering logic. The event filter of a wired-program hardware tool consists of a set of logic elements of combinational and sequential circuits. The interconnection between these elements can be selected and manually manipulated by the user so as to match different signal patterns and sequences for different events. Thus wired-program tools are more flexible than fixed hardware tools.

With *stored-program hardware tools,* filtering functions can be configured and set up by software. Generally, a stored-program hardware tool has its own processor, that is, its own computer. The computer executes programs to set up filtering functions, to define actions in response to detected events, and to process and display collected data. Their ability to control filtering makes stored-program tools more flexible and easier to use. *Logical state analyzers* are typical examples of stored-program hardware tools. With a logical state analyzer, one can specify states to be traced, define triggering sequences, and specify actions to be taken when certain events are detected. In newer logical state analyzers, all of this can be accomplished through a graphical user interface, making them very user-friendly.

Hybrid Monitoring

One of the drawbacks of the hardware monitoring approach is that as integrated circuit techniques advance, more functions are built on-chip. Thus desired signals might not be accessible, and the accessible information might not be sufficient to determine the behavior inside the chip. For example, with increasingly sophisticated caching algorithms implemented for on-chip caches, the information collected from external buses may be insufficient to determine what data need to be stored. Prefetched instructions and data might not be used by the processor, and some events can only be identified by a sequence of signal patterns rather than by a single address or instruction. Therefore passively snooping on the bus might not be effective. Hybrid monitoring is an attractive compromise between intrusive software monitoring and expensive nonintrusive hardware monitoring.

Hybrid monitoring uses both software and hardware to perform monitoring activities (5,16–18). In hybrid monitoring, triggering is accomplished by instrumented software and recording is performed by hardware. The instrumented program writes the selected data to a hardware interface. The hardware device records the data at the hardware interface along with other data such as the current time. Perturbation to the monitored system is reduced by using hardware to store the collected data into a separate storage device.

Current hybrid monitoring techniques use two different triggering approaches. One has a set of selected memory addresses to trigger data recording. When a selected address is detected on the system address bus, the monitoring device records the address and the data on the system data bus. This approach is called *memory-mapped monitoring.* The other approach uses the coprocessor instructions to trigger event recording. The recording unit acts as a coprocessor that executes the coprocessor instructions. This is called *coprocessor monitoring.*

With memory-mapped monitoring, the recording part of the monitor acts like a memory-mapped output device with a range of the computer's address space allocated to it (5,16,17). The processor can write to the locations in that range in the same way as to the rest of the memory. The system or program to be monitored is instrumented to write to the memory locations representing different events. The recording section of the monitor generally contains a comparator, a clock and timer, an overflow control, and an event buffer. The clock and timer provide the time reference for events. The resolution of the clock guarantees that no two successive events have the same time stamp. The comparator is responsible for checking the monitored system's address bus for designated events. Once such an address is detected, the matched address, the time, and the data on the monitored system's data bus are stored in the event buffer. The overflow control is used to detect events lost due to buffer overflow.

With coprocessor monitoring, the recording part is attached to the monitored processor through a coprocessor interface, like a floating-point coprocessor (18). The recorder contains a set of data registers, which can be accessed directly by the monitored processor through coprocessor instructions. The system to be monitored is instrumented using two types of coprocessor instructions: data instructions and event instructions. Data instructions are used to send event-related information to the data registers of the recorder. Event instructions are used to inform the recorder of the occurrence of an event. When an event instruction is received by the recorder, the recorder saves its data registers, the event type, and a time stamp.

DATA ANALYSIS AND PRESENTATION

The collected data are voluminous and are usually not in a form readable or directly usable, especially low-level data collected in hardware. Presenting these data requires automated analyses, which may be simple or complicated, depending on the applications. When monitoring results are not needed in an on-line fashion, one can store all collected data, at the expense of the storage space, and analyze them off-line; this reduces the time overhead of monitoring caused by the analysis. For monitoring that requires on-line data analysis, efficient on-line algorithms are needed to incrementally process the collected data, but such algorithms are sometimes difficult to design.

The collected data can be of various forms (4). First, they can be either qualitative or quantitative. *Qualitative data* form a finite category, classification, or set, such as the set {busy, idle} or the set of weekdays. The elements can be ordered or unordered. *Quantitative data* are expressed numerically, for example, using integers or floating-point numbers. They can be discrete or continuous. It is easy to see that each kind of data can be represented in a high-level programming language and can be directly displayed as text or numbers.

These data can be organized into various data structures during data analysis, as well as during data collection, and presented as tables or diagrams. Tables and diagrams such as line charts, bar charts, pie charts, and histograms are commonly used for all kinds of data presentation, not just for monitoring. The goal is to make the most important information the most obvious, and concentrate on one theme in each table or graph; for example, concentrate on CPU utilization over time, or on the proportion of time various resources are used. With the advancement of multimedia technology, monitored data are now frequently animated. Visualization helps greatly in interpreting the measured data. Monitored data may also be presented using hypertext or hypermedia, allowing details of the data to be revealed in a step-by-step fashion.

A number of graphic charts have been developed specially for computer system performance analysis. These include Gantt charts and Kiviat graphs (4).

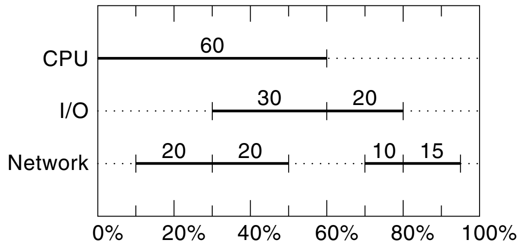

Figure 1. A sample Gantt chart for utilization profile.

Gantt charts are used for showing system resource utilization, in particular, the relative duration of a number of Boolean conditions, each denoting whether a resource is busy or idle. Figure 1 is a sample Gantt chart. It shows the utilization of three resources: CPU, I/O channel, and network. The relative sizes and positions of the segments are arranged to show the relative overlap. For example, the CPU utilization is 60%, I/O 50%, and network 65%. The overlap between CPU and I/O is 30%, all three are used during 20% of the time, and the network is used alone 15% of the time.

A Kiviat graph is a circle with unit radius and in which different radial axes represent different performance metrics. Each axis represents a fraction of the total time during which the condition associated with the axis is true. The points corresponding to the values on the axis can be connected by straight-line segments, thereby defining a polygon. Figure 2 is a sample Kiviat graph. It shows the utilization of CPU and I/O channel. For example, the CPU unitization is 60%, I/O 50%, and overlap 30%. Various typical shapes of Kiviat graphs indicate how loaded and balanced a system is. Most often, an even number of metrics are used, and metrics for which high is good and for which low is good alternate in the graph.

APPLICATIONS

From the perspective of application versus system, monitoring can be classified into two categories: that required by the user of a system and that required by the system itself. For example, for performance monitoring, the former concerns the utilization of resources, including evaluating performance, controlling usage, and planning additional resources, and the latter concerns the management of the system itself, so as to allow the system to adapt itself dynamically to various factors (3).

From a user point of view, applications of monitoring can be divided into two classes: (1) testing and debugging, and (2) performance analysis and tuning. Dynamic system management is an additional class that can use techniques from both classes.

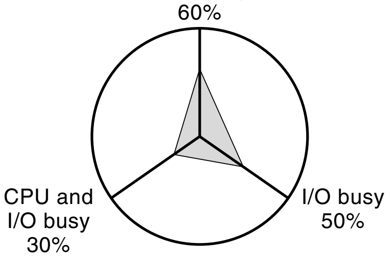

Figure 2. A sample Kiviat graph for utilization profile.

Testing and Debugging

Testing and debugging are aimed primarily at system correctness. Testing checks whether a system conforms to its requirements, while debugging looks for sources of bugs. They are two major activities of all software development. Systems are becoming increasingly complex, and static methods, such as program verification, have not caught up. As a result, it is essential to look for potential problems by monitoring dynamic executions.

Testing involves monitoring system behavior closely while it runs a test suite and comparing the monitoring results with the expected results. The most general strategy for testing is bottom-up: unit test, integration test, and system test. Starting by running and monitoring the functionality of each component separately helps reduce the total amount of monitoring needed. If any difference between the monitoring results and the expected results is found, then debugging is needed.

Debugging is the process of locating, analyzing, and correcting suspected errors. Two main monitoring techniques are used: single stepping and tracing. In single-step mode, an interrupt is generated after each instruction is executed, and any data in the state can be selected and displayed. The user then issues a command to let the system take another step. In trace mode, the user selects the data to be displayed after each instruction is executed and starts the execution at a specified location. Execution continues until a specified condition on the data holds. Tracing slows down the execution of the program, so special hardware devices are needed to monitor real-time operations.

Performance Evaluation and Tuning

A most important application of monitoring is performance evaluation and tuning (3,4,8,13). All engineered systems are subject to performance evaluation. Monitoring is the first and key step in this process. It is used to measure performance indices, such as turnaround time, response time, throughput, and so forth.

Monitoring results can be used for performance evaluation and tuning in as least the following six ways (4,6). First, monitoring results help identify heavily used segments of code and optimize their performance. They can also lead to the discovery of inefficient data structures that cause excessive amount of memory access. Second, monitoring can be used to measure system resource utilization and find performance bottleneck. This is the most popular use of computer system monitoring (6). Third, monitoring results can be used to tune system performance by balancing resource utilization and favoring interactive jobs. One can repeatedly adjust system parameters and measure the results. Fourth, monitoring results can be used for workload characterization and capacity planning; the latter requires ensuring that sufficient computer resources will be available to run future workloads with satisfactory performance. Fifth, monitoring can be used to compare machine performance for selection evaluation. Monitoring on simulation tools can also be used in evaluating the design of a new system. Finally, monitoring results can be used to obtain parameters of models of systems and to validate models. They can also be used to validate models, that is, to verify the representativeness of a model. This is done by comparing measurements taken on the real system and on the model.

Dynamic System Management

For a system to manage itself dynamically, typically monitoring is performed continuously, and data are analyzed in an on-line fashion to provide dynamic feedback. Such feedback can be used for managing both the correctness and the performance of the system.

An important class of applications is dynamic safety checking and failure detection. It is becoming increasingly important as computers take over more complicated and safety-critical tasks, and it has wide applications in distributed systems, in particular. Monitoring system state, checking whether it is in an acceptable range, and notifying appropriate agents of any anomalies are essential for the correctness of the system. Techniques for testing and debugging can be used for such monitoring and checking.

Another important class of applications is dynamic task scheduling and resource allocation. It is particularly important for real-time systems and service providers, both of which are becoming increasingly widely used. For example, monitoring enables periodic review of program priorities on the basis of their CPU utilization and analysis of page usage so that more frequently used pages can replace less frequently used pages. Methods and techniques for performance monitoring and tuning can be used for these purposes. They have low overhead and therefore allow the system to maintain a satisfactory level of performance.

MONITORING REAL-TIME, PARALLEL, AND DISTRIBUTED SYSTEMS

In a sequential system, the execution of a process is deterministic, that is, the process generates the same output in every execution in which the process is given the same input. This is not true in parallel systems. In a parallel system, the execution behavior of a parallel program in response to a fixed input is indeterminate, that is, the results may be different in different executions, depending on the race conditions present among processes and synchronization sequences exercised by processes (1). Monitoring interference may cause the program to face different sets of race conditions and exercise different synchronization sequences. Thus instrumentation may change the behavior of the system. The converse is also true: removing instrumentation code from a monitored system may cause the system to behave differently.

Testing and debugging parallel programs are very difficult because an execution of a parallel program cannot easily be repeated, unlike sequential programs. One challenge in monitoring parallel programs for testing and debugging is to collect enough information with minimum interference so the execution of the program can be repeated or replayed. The execution behavior of a parallel program is bound by the input, the race conditions, and synchronization sequences exercised in that execution. Thus data related to the input, race conditions, and synchronization sequences need to be collected. Those events are identified as process-level events (1). To eliminate the behavior change caused by removing instrumentation code, instrumentation code for process-level events may be kept in the monitored system permanently. The performance penalty can be compensated for by using faster hardware.

To monitor a parallel or distributed system, all the three approaches—software, hardware, and hybrid—may be employed. All the techniques described above are applicable. However, there are some issues special to parallel, distributed, and real-time systems. These are discussed below.

To monitor single-processor systems, only one event-detection mechanism is needed because only one event of interest may occur at a time. In a multiprocessor system, several events may occur at the same time. With hardware and hybrid monitoring, detection devices may be used for each local memory bus and the bus for the shared memory and I/O. The data collected can be stored in a common storage device. To monitor distributed systems, each node of the system needs to be monitored. Such a node is a single processor or multiprocessor computer in its own right. Thus each node should be monitored accordingly as if it were an independent computer.

Events generally need to be recorded with the times at which they occurred, so that the order of events can be determined and the elapsed time between events can be measured. The time can be obtained from the system being monitored. In single-processor or tightly coupled multiprocessor systems, there is only one system clock, so it is guaranteed that an event with an earlier time stamp occurred before an event with a later time stamp. In other words, events are totally ordered by their time stamps. However, in distributed systems, each node has its own clock, which may have a different reading from the clocks on other nodes. There is no guarantee that an event with an earlier time stamp occurred before an event with a later time stamp in distributed systems (1).

In distributed systems, monitoring is distributed to each node of the monitored system by attaching a monitor to each node. The monitor detects events and records the data on that node. In order to understand the behavior of the system as a whole, the global state of the monitored system at certain times needs to be constructed. To do this, the data collected at each individual node must be transferred to a central location where the global state can be built. Also, the recorded times for the events on different nodes must have a common reference to order them. There are two options for transferring data to the central location. One option is to let the monitor use the network of the monitored system. This approach can cause interference to the communication of the monitored system. To avoid such interference, an independent network for the monitor can be used, allowing it to have a different topology and different transmission speed than the network of the monitored system. For the common time reference, each node has a local clock and a synchronizer. The clock is synchronized with the clocks on other nodes by the synchronizer.

The recorded event data on each node can be transmitted immediately to a central collector or temporarily stored locally and transferred later to the central location. Which method is appropriate depends on how the collected data will be used. If the data are used in an on-line fashion for dynamic display or for monitoring system safety constraints, the data should be transferred immediately. This may require a high-speed network to reduce the latency between the system state and the display of that state. If the data are transferred immediately with a high-speed network, little local storage is needed. If the data are used in an off-line fashion, they can be transferred at any time. The data can be transferred after the monitoring is done. In this case, each node should have mass storage to store its local data. There is a disadvantage with this approach. If the amount of recorded data on nodes is not evenly distributed, too much data could be stored at one node. Building a sufficiently large data storage for every node can be very expensive.

In monitoring real-time systems, a major challenge is how to reduce the interference caused by the monitoring. Real-time systems are those whose correctness depends not only on the logical computation but also on the times at which the results are generated. Real-time systems must meet their timing constraints to avoid disastrous consequences. Monitoring interference is unacceptable in most real-time systems (1,14), since it may change not only the logical behavior but also the timing behavior of the monitored system. Software monitoring generally is unacceptable for real-time monitoring unless monitoring is designed as part of the system (19). Hardware monitoring has minimal interference to the monitored system, so it is the best approach for monitoring real-time systems. However, it is very expensive to build, and sometimes it might not provide the needed information. Thus hybrid monitoring may be employed as a compromise.

CONCLUSION

Monitoring is an important technique for studying the dynamic behavior of computer systems. Using collected run-time information, users or engineers can analyze, understand, and improve the reliability and performance of complex systems. This article discussed basic concepts and major issues in monitoring, techniques for event-driven monitoring and sampling monitoring, and their implementation in software monitors, hardware monitors, and hybrid monitors. With the rapid growth of computing power, the use of larger and more complex computer systems has increased dramatically, which poses larger challenges to system monitoring (20,21,22). Possible topics for future study include:

- New hardware and software architectures are being developed for emerging applications. New techniques for both hardware and software systems are needed to monitor the emerging applications.
- The amount of data collected during monitoring will be enormous. It is important to determine an appropriate level for monitoring and to represent this information with abstractions and hierarchical structures.
- Important applications of monitoring include using monitoring techniques and results to improve the adaptability and reliability of complex software systems and using them to support the evolution of these systems.
- Advanced languages and tools for providing more user-friendly interfaces for system monitoring need to be studied and developed.

BIBLIOGRAPHY

1. J. J. P. Tsai et al., *Distributed Real-Time Systems: Monitoring, Visualization, Debugging and Analysis,* New York: Wiley, 1996.
2. D. Ferrari, Workload characterization and selection in computer performance measurement, *IEEE Comput.,* **5** (7): 18–24, 1972.
3. D. Ferrari, G. Serazzi, and A. Zeigner, *Measurement and Tuning of Computer Systems,* Englewood Cliffs, NJ: Prentice-Hall, 1983.
4. R. Jain, *The Art of Computer Systems Performance Analysis,* New York: Wiley, 1991.
5. P. McKerrow, *Performance Measurement of Computer Systems,* Reading, MA: Addison-Wesley, 1987.
6. G. J. Nutt, Tutorial: Computer system monitors, *IEEE Comput.,* **8** (11): 51–61, 1975.
7. L. Svobodova, *Computer Performance Measurement and Evaluation Methods: Analysis and Applications,* New York: Elsevier, 1976.
8. H. C. Lucas, Performance evaluation and monitoring, *ACM Comput. Surv.,* **3** (3): 79–91, 1971.
9. J. M. Anderson et al., Continuous profiling: Where have all the cycles gone, *Proc. 16th ACM Symp. Operating Syst. Principles,* New York: ACM, 1997.
10. C. H. Sauer and K. M. Chandy, *Computer Systems Performance Modelling,* Englewood Cliffs, NJ: Prentice-Hall, 1981.
11. B. Plattner and J. Nievergelt, Monitoring program execution: A survey, *IEEE Comput.,* **14** (11): 76–93, 1981.
12. T. Ball and J. R. Larus, Optimally profiling and tracing programs, *ACM Trans. Program. Lang. Syst.,* **16**: 1319–1360, 1994.
13. D. Ferrari, *Computer Systems Performance Evaluation,* Englewood Cliffs, NJ: Prentice-Hall, 1978.
14. B. Plattner, Real-time execution monitoring, *IEEE Trans. Softw. Eng.,* **SE-10**: 756–764, 1984.
15. B. Lazzerini, C. A. Prete, and L. Lopriore, A programmable debugging aid for real-time software development, *IEEE Micro,* **6** (3): 34–42, 1986.
16. K. Kant and M. Srinivasan, *Introduction to Computer System Performance Evaluation,* New York: McGraw-Hill, 1992.
17. D. Haban and D. Wybranietz, Real-time execution monitoring, *IEEE Trans. Softw. Eng.,* **SE-16**: 197–211, 1990.
18. M. M. Gorlick, The flight recorder: An architectural aid for system monitoring, *Proc. ACM/ONR Workshop Parallel Distributed Debugging,* New York: ACM, May 1991, pp. 175–183.
19. S. E. Chodrow, F. Jahanian, and M. Donner, Run-time monitoring of real-time systems, in R. Werner (ed.), *Proc. 12th IEEE Real-Time Syst. Symp.,* Los Alamitos, CA: IEEE Computer Society Press, 1991, pp. 74–83.
20. R. A. Uhlig and T. N. Mudge, Trace-driven memory simulation: A survey, *ACM Comput. Surg.,* **29**(2): 128–170, 1997.
21. M. Rosenblum et al., Using the SimOS machine simulator to study complex computer systems, *ACM Trans. Modeling Comput. Simulation,* **7**: 78–103, 1997.
22. D. R. Kaeli et al., Performance analysis on a CC-PUMA prototype, *IBM J. Res. Develop.,* **41**: 205–214, 1997.

YANHONG A. LIU
Indiana University
JEFFREY J. P. TSAI
University of Illinois