Laboratory for Computer Science
Computer Systems Research Division

August 11, 1980
Request for Comments No. 194

PERFORMANCE MONITORING IN COMPUTER SYSTEMS: STRUCTURED APPROACH

by Liba Svobodova

Abstract

This paper discusses the characteristics and design of performance monitors from new perspectives. It argues that performance monitoring, similar to monitoring needed to ensure reliability, ought to be viewed as an ongoing process that is inseparable from the system operation. Performance monitors should be designed as part of the system, following the principles of structured system design and using the methods and tools developed for this purpose.

1. Introduction

As any complex system, a computer system ought to be equipped with a set of tools that assist with identifying existing or potential problems in the operation of the system. Some tools are needed to respond to the immediate state of the system if continued operation according to the normal algorithm would cause the system to fail. However, it is also important to provide tools that can record cumulative data pertaining to demand for and utilization of the system resources and assist in analyzing performance problems. For certain events, both types of tools are desirable. That is, in addition to an immediate response to a potentially dangerous situation, it is also desirable to make a record of such an event for later statistical summary and analysis. Of course, many such tools have been built. Performance monitors in the form of programs or stand-alone hardware devices have been available on the market for more than a decade. Special hardware circuits and diagnostics routines have been built into computer systems to detect and recover from errors. However, the work thus far has been lacking in two important aspects. One is the recognition of the link between monitoring performance (efficiency) of an error-free system and monitoring failures or problems (errors) that might turn into a failure. The other aspect is that while a lot of pioneering work has been done on the methodology of computer system design (e.g., structured programming, levels of abstractions, top-down hierarchical design, etc.), it has not affected significantly the design of tools needed to monitor system performance and the assessment of their role in the system design.

This paper is concerned with the characteristics and design of performance monitors,[1] in particular, with the incorporation of such tools into the computer system hardware and software. Performance monitoring is viewed not as an occasional activity that produces side effects, but as an ongoing process that is an inseparable part of the system operation. In this sense, performance monitoring is very

[1] The term "monitor" has gained a widespread use in software engineering to mean a special synchronization construct devised by Hoare [HOAR 74]. The monitors discussed in this paper are more observers than controllers, concerned with providing insight into the system behavior and the use of system resources rather than imposing constraints on the use of resources. However, as will be seen, construct similar to those used for synchronization can be utilized in the type of monitoring discussed in this paper.

similar to monitoring needed to ensure reliability. The paper argues about the need to design the monitoring tools with the system, applying the same design methods as used for the system itself. In fact, support for performance monitoring should be provided in programming languages; some examples of how performance monitors can be designed as abstractions, using only the mechanisms provided by a high level language, are included.

2. Performance and Reliability

In the context of the subject of this paper, that is, design and implementation of performance monitors, two observations about the relationship between performance monitoring and reliability monitoring can be made. First, it should be realized that reliability is in fact one of the aspects of system performance. In the early reports aimed at defining computer system performance, reliability was included as one of the performance measures [CALI 67, DRUM 69], however, reliability usually meant availability of the system as a whole. And, just as most performance studies concentrate strictly on evaluating the efficiency and utilization of an error free system, until recently, evaluation of the designs for fault-tolerant systems was not too concerned with the system efficiency. Yet there clearly exists a trade-off between the system reliability and efficiency, since the mechanisms needed to achieve fault-tolerance may represent a significant overhead. A different kind of relationship between reliability and efficiency is demonstrated in fail-soft systems, where the performance of the system degrades as a result of an occurrence of an unrecoverable fault, that is, performance of the system as a whole depends on the reliability of its components. Distributed systems present another example: reliability of the communication network and individual nodes will have significant effect on system performance.

Several recent studies have been concerned with system performance evaluation that takes errors into consideration [GELE 78, MEYE 79, BEAU 77, GARC 79]. In this paper, this subject is brought up mainly to emphasize that performance monitors should be able to observe both the normal and the abnormal system behavior.

This immediately leads to the second point, the similarity of the tools needed to monitor system performance and those needed to ensure reliability. The most basic common characteristic is that all these tools have to have access to certain set of events, that is, must know that a relevant event occurred. Thus in both cases the system must be designed such that relevant events can be detected, and the notification of their occurrence can be passed to the appropriate monitor(s). In respect to the second step, several recent programming languages include powerful and flexible exception handling mechanisms that can detect various (possibly harmful) conditions and signal their occurrence to a special kind of monitor, the exception handler [Levi 77, LISK 77B, GOOD 75]. The concept of exception handling is sufficiently powerful to support more general performance monitoring; an interesting example of using exception handling mechanisms for management of a shared buffer pool was presented by Levin [LEVI 77]. Of course, to be able to monitor performance of the hardware components, the hardware must provide some aid in conveying the needed information to the programs running on the hardware; similar aid is needed in order to handle hardware faults and other errors detected by the hardware.

However, there are more important aspects of monitoring tools than their structure. I am particularly concerned with the philosophy underlying the use of monitoring tools, and the acceptance that they are an important and inseparable part of the system. I believe that performance monitoring should be done in a similar spirit as the monitoring needed for reliable system operation. This point will be elaborated throughout the rest of this paper.

3. Temporary vs. Permanent Instrumentation

To understand what types of tools are needed, it is instructive to summarize how the performance of a system is influenced. Basically, the performance is determined at three different levels: the conceptual design of the system, the implementation of the system, and the adjustment of the system to a particular application. The last stage includes the initial selection of a system and subsequent tuning. The first stage cannot utilize any measurement tools, since there is not yet any system to measure. The tools needed here are predictive tools, either analytical or simulators. During implementation extensive temporary instrumentation may be needed; the measurements are concerned primarily with the efficiency and correctness of the implementation. Much of this instrumentation will not be used during the normal operation of the system. Tuning, however, is an ongoing process, since the needs and expectations of the users will change over time. Performance data should be collected continuously although the collected data may not be always used. Such measurements can be compared to maintenance of backup files needed to recover from a system crash. Such backup files will be used to reconstruct the state of the system should a crash destroy the current online version of some file, but since such an event should be very rare, it can be said that the backup files are mostly unused. However, it is too late to start worrying about backup when the trouble is detected. Similarly, although much of the collected performance data will not be used, when a performance problem is finally detected, having a record of the earlier symptoms may be very helpful. Thus, tuning ought to be supported by permanent instrumentation. Permanent instrumentation is also needed for continuous dynamic control of allocation of system resources. One step further are self-tuning application systems that know, based on the measurement data collected during their use, how to reconfigure for best performance. Consequently, it is necessary to:

1) provide tools for (temporary) measurement during
system implementation

2) provide tools for (permanent) continuous
monitoring during system use.

Of course, these two classes of tools may overlap, that is, a tool needed during implementation may also prove useful during the production phase of the system. However, it is important to make this distinction, because these two classes represent different considerations in assessing the side effects of the tool.

The side effects of a performance monitor have always been a strong argument in the discussions of hardware vs software tools. The possible side effects come in two flavors:

1) Performance side effects: the tool uses some of the
system resources, thus reducing the effective performance of
the system as seen by the user.
2) Functional side effects: the tools changes the
functional behavior of the system, in particular, introduces
errors.

Part of the performance side effects actually represents an accountable overhead, that is, overhead that can be measured and discounted from the data obtained from the measured system. The unaccountable part is due to subtle interactions of the monitor with the monitored activities in their competition for system resources; such interactions perturb the measured system in ways that are not fully understood.

Now, the temporary measurements most probably will have performance side effects: once the instrumentation is removed, the system is different. Since the performance should be better without the tool, this should not be too serious. In the case of permanent instrumentation, it does not make sense to talk about side effects, since the monitor is an unseparable part of the system. That is, the system is not perturbed by monitoring, it only may be more costly to run such systems. Again, an analogy with the monitoring of the system reliability is appropriate. Reliability mechanisms represent additional cost, both because of the additional hardware and because of the additional execution time needed to perform the necessary checks. But they are an important part of the system, and nobody thinks about them as a "perturbation" of some ideal, pure system. Of course, for permanent monitoring, it is desirable that the monitoring tools be inexpensive, since they will be used continuously; this consideration may not be that important in the case of temporary instrumentation. In both cases, however, it is important that monitoring does not cause any functional errors, either when the monitor is actively used, or when it is turned off. The SMT, a measurement system designed at UC Berkeley, performed some integrity checks before inserting measurement traps specified by the user [FERR 74], but it is still a dangerous method that violates the principles of good system design. Since permanent instrumentation is designed with the system, it can be made to undergo the same rigorous tests as the rest of the system. The subject of the "structured" design of performance monitors will be investigated in Section 5. The next section presents the basic measurement concepts that are independent of the monitor implementation.

4. Basic Concepts of Performance Measurement

Performance monitors usually are classified according to their implementation as hardware, software, and hybrid monitors. These divisions are becoming more and more vague, as more hardware monitors utilize various software and more systems provide hardware support for software monitors. Rather than discussing the differences in the characteristics of performance monitors that are a result of a particular implementation, I believe that it is more important to understand their common base.

The system behavior is observable through changes in the system state. A change in the system state marks either the beginning or the end of a period of some activity (or inactivity) of a system component (a hardware component, a software component, or a process). A change in the system state is called an event: an event is a transition from one system state to another that occurs when certain conditions are satisfied.

To be able to extract information about the changes in the system state, the system must be instrumented. The form of instrumentation may depend on what type of information is desired about individual components and their interaction. In particular, one may be interested only in how often a certain operation is performed (that is, how often a particular state is entered). In this situation, it is necessary only to count the occurrences of the event that marks the entering of that state. Some examples of this type of measurement are: count the seeks performed on each disk device; count the invocations of a specified procedure, measure the frequency of page faults, measure the frequency of memory parity errors. The frequency of an event can be obtained by dividing the accumulated count of the occurrences by the length of the time period during which the counting took place.

A more difficult situation involves timing of observed activities. The simplest case is to determine what percentage of time, on the average, the system spends in a particular state. Measurement of the CPU or the I/O channel utilization and concurrent use (overlap) of the CPU and I/O channel belong to this category. Finally, the most difficult problem is to time individual instances of a specific activity, that is, to measure, each time a specific state is entered, the length of time for which the system remains in that state. An example of this type of problem is a measurement of the distribution of the CPU service times, where the service time is the length of time for which a particular computation can utilize the CPU, once it has allocated the CPU. Another example is measurement of time between errors (such as memory parity errors or disk head crashes).

In all three cases, it is assumed that it is possible to detect the appropriate changes in the system state. Actually, for the second category, it is not necessary to detect exactly when the state changes. These types of measurements are often done by sampling the system state. The system state is interrogated at random time points, random in the sense that these time points are not related to the measured activities.[1] Each such observation either will or will not find the system in the state corresponding to the measured activity. The percentage of time spent in this state can then be determined as a ratio of the successful observations (the system was in the state of interest) to the total number of observations.[2] It should be pointed out that hardware performance monitors perform timing this way. Since the sampling rate is normally very high (1MHz - 10MHz), it is possible to convert the number of observations into time with very

[1] Since, due to the variability and unpredictability of the workload, the occurrence and duration of most system activities can be viewed as random variables, sampling can be performed at regular time intervals.

[2] Note that sampling can be used also for reliability purposes. For example, in some systems individual components are periodically queried to find out whether they are "alive".

high precision. Also, a hardware tool, if it can time duration of an activity in this way, can also detect the corresponding changes in the system state. As a result, this method in the context of hardware tools is not usually viewed as sampling. However, if the measurements are performed by programs executing within the measured system, the sampling rate has to be kept relatively low; it is often preferrable to use a direct timing method described below.

The direct timing method is very simple. A clock is read when the measured activity begins and again when it ends. The difference of these two readings is the duration of that instance of the measured activity. This method, of course, requires that the events marking the beginning and the end of each such occurrence are detectable. In addition, it assumes the existence of a clock that has sufficient precision and is easily accessible to the monitor for the particular activity.

5. Use of Abstraction in a Design of Performance Monitors

It has been long advocated that systems should be built as a hierarchy of levels where each level represents a specific abstraction of the system that can be understood without having to know the details of the lower levels (the implementation). Several recent programming languages provide powerful abstraction mechanisms that separate the specification of the behavior of a system as seen by the system users from the internal organization (implementation) of the system. That is, not only is the user encouraged to look at only those features that are relevant to his task, but may actually be prevented from getting into the internals of the system directly. Microprocessors on a chip can also be viewed as abstractions that cannot be "opened" to investigate their internal structure and behavior. Thus it is important that each such abstraction has built-in tools for performance monitoring.

The use of abstractions has another impact on performance monitoring: at each level, performance ought to be measured in terms of the functions (services) provided by the abstraction. Such a service-oriented approach has been used earlier in computer performance evaluation, in particular, by defining workload in terms of requests for logical services rather than hardware resources [MORG 73].

Finally, the same way more elaborate and powerful systems are built from lower level components, complex performance monitors ought to be built methodically from more basic tools. An example of a structured design of a performance monitor can be found in [JUER 78]; this monitor was developed as an integral part of a layered operating system. However, such facility should also be available to the system users, in particular, implementors of application systems.

To summarize, abstraction mechanisms ought to be exploited in two ways:

1) in the design and implementation of monitors

2) in the instrumentation of the system to be monitored.

A programming language can be of great help in the design and implementation of performance monitors. In fact, this paper was inspired by the current research in programming languages, in particular, languages that support the notion of abstract objects. An abstract object is defined completely by its possible external behavior, that is, by a set of operations allowed on the object and their effects. Further, abstraction mechanisms such as the cluster in CLU [LISK 77A] or form in Alphard [WULF 76] define a class of objects of a particular type. Specific instances of a type can be then created, all of which inherit the same behavioral properties. All objects of the same type are manipulated by the same (shared) set of procedures that form the type manager. Now, all objects of the same type can be monitored in the same way, that is, monitoring functions can be defined as part of the type manager. Moreover, in addition to monitoring each object individually, it is also possible to monitor the whole class of objects. In particular, the creation/destruction rate of objects of a specific type is of interest.

As an example, consider a FIFO queue: as an abstract object with operations:

create () returns (queue)	- creates an empty queue
enqueue (q:queue,x:element)	- inserts the new element at the tail of the queue
dequeue (q:queue) returns(x)	- removes and returns the

first element in the queue

length (q:queue) returns(int) - returns an integer number which is the number of elements in the queue

destroy (q:queue) - destroys the queue

Now assume that a specific instance q of a queue was created and we want to monitor its length. We may be interested in the mean length, the distribution of the lengths, or the occasions when the queue length exceeds some limit. In addition, these measures could be based just on the number of occurrences, or include the time duration. Thus we would like to have some language construct that allows us to specify:

1) which object is to be monitored

2) what properties of this object are to be monitored.

3) how the specified properties are to be monitored.

For example, to specify that we want to get the mean length of the queue q with respect to time, the following construct could be used:[1]

queue$monitor(q, length: mean/time).

As elements are enqueued and dequeued, the monitor measures the time intervals t between the changes in the queue length and accumulates the sum

[1] The examples are given in a CLU-like syntax. In particular, dtype$op denotes an operation "op" for an object of type "dtype" and x:dtype specifies a parameter "x" of type "dtype".

$$\sum_{i=1}^{k} l_i \; t_i$$

where l_i is the length of q during t_i and k is the number of changes in the queue length observed thus far. To get the measurement data, another language construct is needed:

queue$stat(q, length).

The queue$monitor operation encloses the specified queue in an envelope within which the relevant operations are intercepted and where the measurement data are accumulated. Figure 1 illustrates this situation. This model is similar to a model supported by some synchronization constructs that control access to shared objects, namely a serializer [HEWI 77]. However, the mapping from the specification of the measurement request to the implementation of the monitor is, in general, non-trivial. In the example presented here, monitoring of the length requires interception of the enqueue and dequeue operations and the use of a clock to measure the time between subsequent invocations of these operations on the specified queue. The possibility of synthesizing code from a specification of an abstraction is a subject of much research, but for most types of objects (systems), the performance monitors must be designed and implemented as part of the type manager.

Finally, let us return to the case of monitoring the length of the queue beyond some limit. Again, it would be possible to count how many times an element is added beyond this limit or time for how long the queue length exceeds the limit. The third possibility is to alert the program that uses the queue, that is, to signal an occurrence of this event. Thus, it should also be possible to set the monitor in the following way:

queue$monitor(q, length: limit=n/signal) where n is an integer number.

In this case, a facility similar to exception handling would propagate the signal to a monitor provided by the program that uses this particular queue. However, it should be possible to resume the operation on the queue after an occurrence of the monitored event has been properly recorded. The exception handling mechanisms in CLU do not permit such a resumption: CLU supports only a termination model [LISK 77B]. However, the resumption model has been developed for other languages and advocated by many researchers [LEVY 77, GOOD 75].

Finally, the abstractions presented here need to be placed into proper perspective with regard to the discussion in Section 3 about temporary vs. permanent instrumentation. The monitor construct is intended for selective permanent instrumentation, that is, a monitor is initiated when a specific queue is created and disappears only when that queue is destroyed. Of course, the monitor call could be removed from (or inserted into) an existing program, thus turning it into a "temporary" instrumentation, but this will not affect the correctness of the implementation of the queue and the operations performed on the queue. It is this last property that is considered most important.

6. Basic Support

Thus far, the discussion of the language support assumed that the ability to monitor performance of various abstract types is built into the type manager and that the language provides constructs for selectively starting the monitoring of a specific object and retrieving the measurement data. The signal option provides an escape from the standard monitor supported by the type manager: the user of the abstract type can implement its own monitor that will be triggered by the signal. However, some tools are needed also to support the construction of the monitors.

The discussion of the instrumentation and to some extent the discussion in the last section have shown that at the bottom level, monitoring consists of rather primitive activities: event detection, event counting, direct timing, sampling. In addition, measurement data need to be accumulated in various ways. To aid in design and implementation of performance monitors, the basic system (that is, the hardware, the operating system, and the programming support) ought to come equipped with a set of basic tools that support these primitive activities:

clocks

counters

accumulators

mechanisms for event detection and signalling.

Some suggestions concerning the hardware support can be found in [SVOB 76]. Here the discussion will concentrate primarily on the language support.

The clock deserves a special attention. Many present system have a built-in high-resolution program-readable hardware clock that is incremented with strict regularity (driven by crystal oscillator), independently of the instruction execution and memory accesses. This clock may be used to measure the elapsed time, and it measures elapsed time with high accuracy. However, this clock is an unsuitable tool for a programmer who wants to measure the amount of CPU time used by a particular program in a multiprogramming environment. In such an environment, the execution of a program can be interrupted by a higher priority task, by a completion of an I/O operation, or, in a time-sharing system, because the respective task has exceeded its allocated time quantum. For such measurements, the system should provide for each multiprogrammed task a virtual clock that is incremented (automatically, by the hardware) only when the corresponding task is actually using the processor. Finally, to be able to use this virtual clock to time activities specified in a higher level language, it is necessary to incorporate it into a tool accessible from that language.

The (virtual) clock can be viewed as an abstract object with the operations:

create()returns (clock)

read(c:clock)returns(time)

destroy(c:clock)

The operations create and destroy ought to be privileged operations that can be executed only as part of creation/destruction of a process. A virtual clock can be built using the real clock, with the precision of the real clock. Its accuracy, however, will depend on how well it is possible to detect that a task is being interrupted and how the time for which the task has been interrupted is discounted; an implementation of such a virtual clock on conventional hardware is not always straightforward, and may be even impossible.

The basic measurement tool that ought to be given to the programmer is an interval timer based on the virtual clock. The interval timer is again an abstract object with the operations:

create () returns (interval_timer)	- creates a new interval timer
start (IT: interval_timer)	- starts the specified timer at time 0
read (IT: interval_timer) returns (time)	- returns the time since the last reading (or since start)
destroy (IT: interval_timer)	- destroys the specified interval timer

A process can create and use any number of these timers.

Now, since the underlying implementation of the read operation on the virtual clock (clock$read) is a procedure call, read does not return the value of the clock instantaneously, but it adds some overhead. The implementation of the interval timer ought to adjust for this overhead, that is, the overhead ought to be subtracted from the time interval obtained by two subsequent readings of the virtual clock. It is assumed that the overhead has been measured beforehand. Since the overhead depends on the speed of the underlying hardware and the language implementation, the interval timer needs to be "adjusted" if either one is changed.

A measurement experiment aimed at determining the characteristics of the virtual clock in the Multics system is described in the Appendix. The results of this experiment demonstrate that such tools cannot be always trusted. The abstract interval timer proposed here should shield the users from all implementation aspects, that is, the users should not have to worry about the overhead of the tool or the possible inaccuracies due to interference from other processes.

Conclusions

The intention of this paper was to convey a new way of thinking about performance monitoring, in particular, to bring attention to the relationship between performance monitoring and reliability monitoring, and to the impact of computer system design principles on a design of performance monitors.

The main shortcoming of this paper is that it does not present a concrete example of a design of a performance monitor that would prove the claims. This is where the future work shall focus.

Appendix

Study of the Multics Virtual Clock

The Multics system was instrumented early in its implementation cycle [SALT 70]. The Multics users at MIT can look at a variety of system performance aspects by invoking high level commands from their terminals. To measure performance of their own programs, the lower level abstractions (e.g., components that accumulate measured data) can be used directly. One of the most important tools provided to the programmer is the virtual clock supported in PL/1, the most widely used language on Multics. The virtual clock can be read and reset by PL/1 callable procedures.

A computer performance measurement laboratory course developed at MIT, studied the Multics system. The first few experiments assigned to the students are concerned with the accuracy of the Multics measurement tools. Accuracy has meaning only with respect to a specific measurement task; it is the objectivity of the results with respect to the real world. Low accuracy may be caused by interference of the tool with the measured system, by low resolution of the tool (that is, the tool may be unable to detect and record events at the rate required by the measurement experiment), or the tool may be simply unsuitable for that type of measurement. Thus before implementing or applying measurement tools, it is necessary to specify carefully what is to be measured.

The measurement of the virtual clock is concerned with two characteristics:

1) the overhead of the PL/1 built-in function

2) the accuracy of the virtual clock supported by the
operating system.

For most measurements, the first item can be considered negligible. However, the accuracy of the virtual clock is a problem. Figure 2a shows results of an experiment where a very short program (in fact, the PL/1 procedure that reads the clock) was timed 10,000 times. The virtual clock was reimplemented in 1978; also, it is now provided as a PL/1 built-in function. The latter step changed the overhead. The former step drastically improved the accuracy, as demonstrated in Figure 2b.

References

BEAU 77 Beaudry, D.M., "Performance-Related Reliability Measures for Computing Systems," Proc. of the 7th IEEE Symposium on Fault-Tolerant Computing, June 1977, pp. 16-21.

CALI 67 Calingaert, P., "System Performance Evaluation: Survey and Appraisal," CACM, Vol. 15, No. 3 (March 1967), pp. 185-190.

DRUM 69 Drummond, M.E., "A Perspective on System Performance Evaluation," IBM Systems Journal, Vol. 8, No. 4 (1969), pp. 252-263.

FERR 74 Ferrari, D., Liv, M., "A General-Purpose Software Measurement Tool," Proc. Second SIGMETRICS Symposium on Measurement and Evaluation, September 1974, pp. 94-103.

GARC 79 Garcia-Molina, H., "Performance of Update Algorithms for Replicated Data in a Distributed Database," Technical Report No. STAN-CS-79-744, Department of Computer Science, Stanford University, June 1979.

GELE 78 Gelenbe, E., Derochette, D., "Performance of Rollback Recovery Systems under Intermittent Failures," CACM, Vol. 21, No. 5 (June 1978), pp. 493-499.

GOOD 75 Goodenough, J.B., "Exception Handling: Issues and a Proposed Notation," CACM, Vol. 18, No. 12 (December 1975), pp. 683-696.

HEWI 77 Hewitt, C., Atkinson, R., "Parallelism and Synchronization in Actor Systems," Proc. of Conference on Principles of Programming Languages, January 1977, pp. 267-280.

HOAR 74 Hoare, C.A.R., "Monitors: An Operating System Structuring Concept," CACM, Vol. 17, No. 5 (October 1974), pp. 549-557.

JUER 78 Juergens, J., Pantele, E.F., "Collection and Output of Performance Measurement Data in a Layered Operating System," Proc. of CIPS Session '78, May 1978, pp. 142-146.

LEVI 77 Levin, R., "Program Structures for Exceptional Condition Handling," Ph.D. Thesis, Department of Computer Science, Carnegie-Mellon University, June 1977.

LISK 77A Liskov, B., Snyder, A., Atkinson, R., Schaffert, C., "Abstraction Mechanisms in CLU," CACM, Vol. 20, No. 8 (August 1977), pp. 564-576.

LISK 77B Liskov, B., Snyder, A., "Structured Exception Handling," Computation Structures Group Memo 155, Laboratory for Computer Science, M.I.T., December 1977.

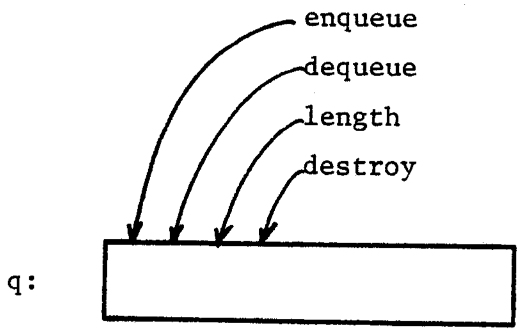
MEYE 79 Meyer, J.F., Furchtgott, D.G., Wu, L.T., "Performability Evaluation of the SIFT Computer," Proc. of the 9th IEEE Symposium on Fault Tolerant Computing, June 1979, pp. 43-50.

MORG 73 Morgan, D.E., Campbell, J.A., "An Answer to a User's Plea?," Proc. of 1st ACM SIGME Symposium on Measurement and Evaluation, February 1973, pp. 112-120.

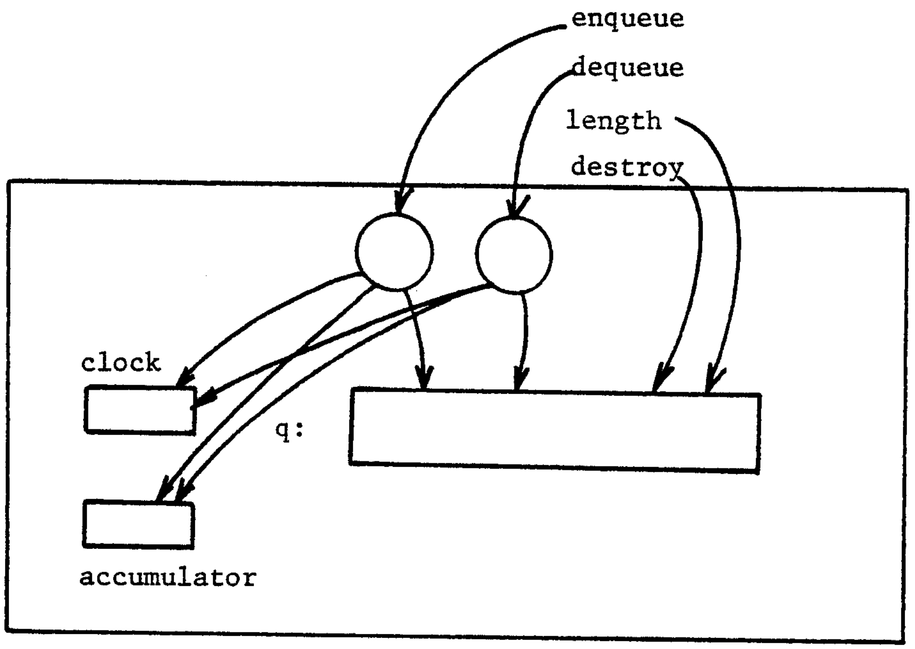
SALT 70 Saltzer, J.H., Gintell, J.W., "The Instrumentation of Multics," CACM, Vol. 13, No. 8 (August 1970), pp. 495-500.

SVOB 76 Svobodova, L., "Computer System Measurability," COMPUTER, Vol. 9, No. 6 (June 1976), pp. 9-17.

WULF 76 Wulf, W.A., London, R.L., Shaw, M., "An Introduction to the Construction and Verification of Alphard Programs," IEEE Trans. on Software Engineering, Vol. SE-2, No. 4 (December 1976), pp. 253-265.



a. Existing queue -- not monitored



b. Monitored queue

Figure 1: Monitoring the length of q:queue

```
trials =  10000                       221            1
                                      222            1
   56         3010                    230            3
   57         6542                    231            2
   58          263                    232            1
   59           38                    233            1
   60            2                    235            2
   61            6                    236            3
   62            4                    237            5
   63            2                    238            4
   64            2                    239            8
   65            1                    240            1
   67            2                    242            2
   68            1                    243            2
   69            1                    244            1
   70            2                    245            1
   71            3                    246            4
   72            3                    247            1
   74            1                    250            1
   77            2                    251            2
   78            4                    255            1
   83            3                    260            1
   84            5                    264            1
   85            4                    265            3
   86            1                    266            3
   87            1                    267            5
   88            1                    268            3
   90            3                    269            2
   91            1                    270            2
   92            2                    271            2
   95            1                    273            1
  100            1                    280            1
  171            1                    448            1
  201            1
  202            2
  207            1                  mean      =   58.48
  208            2
  209            1                  variance  =  315.58
  212            1
  215            1                  maximum   =   448
  217            1
  218            5
```

in µsec

a. Old implementation

```
trials = 10000

     16            2
     17            1
     18            1
     19            1
     21           32
     22         6825
     23         3061
     24           25
     25            3
     26            1
     27            3
     28            4
     29            4
     30            3
     31            6
     32            5
     33            3
     34            6
     35            1
     36            1
     38            2
     40            1
     41            1
     44            1
     55            2
     56            1
     57            1
     58            1
     60            1
     64            1

mean      =   22.37

variance  =    1.59

maximum   =   64
```

in µsec

b. New implementation

Figure 2: Results of a vitrual clock measurement experiment