A Flexible Measurement Tool For Software Systems

P. Deutsch
C. A. Grant
University of California, Berkeley

Abstract

Implementers of large programming systems have come to realize the desirability of a flexible facility for measuring these programs. The applications of such a facility are tracing, monitoring for special conditions, and gathering statistics for performance evaluation.

Our work will discuss a particular measurement tool, called the Informer, now being used to measure the time-sharing operating system of the SDS-940 at Berkeley. The Informer provides an environment in which user written programs may serve as measurement routines. Measurement routines may be inserted dynamically to be called when control reaches arbitrary locations within the measured program. An important quality of this measurement facility is that no error in its use can cause a system failure. Also, system degradation due to measurement programs can be automatically controlled. The Informer is a software system - no hardware probes are employed.

1. Introduction

The Informer is meant to attack three problems that arise continually in large programming systems: debugging, performance anlaysis, and environment analysis. We first consider these problems in more detail before describing the workings of the Informer itself.

Large programs always have errors in them. These errors fall into two categories:

A) Logic bugs - these are errors that cause the program to operate incorrectly. That is, the program gives wrong answers or has improper side effects or "blows up" in some manner. Many program bugs are timing dependent and occur unpredictably, when the environment of the program achieves some exceptional state. Timing dependent bugs are rarely reproducible, and represent the thorniest of problems in debugging.

B) Performance bugs - a program may operate correctly but manage resources inefficiently in doing so. In the case of a time-sharing operating system, response time may be much longer or CPU utilization poorer than analysis indicates the hardware configuration could provide. The reasons for this lie in inefficient or improper algorithms for utilization of the hardware.

We are particularly interested here in tools for dealing with programs where the flow of control is dependent on many external conditions which change dynamically at a very high frequency. The flow of control is highly unpredictable for any real run of such a program and generally impossible to recreate. In this paper we will use time-sharing operating systems as an example. Of course, the best path to reliability in such complex systems is careful design rather than persistent or even inspired debugging. However, the tools required for rigorous elimination of logical errors are still in an embryonic state and the statistical methods for a priori performance analysis tend to be unmanageable in real-life cases. In terms of effort required to reach an operational system, flexible debugging tools seem to be the best investment at present.

A global (user's-eye) evaluation of the performance of a time-sharing operating system might be based on a wide range of characteristics including convenience of use, reliability, and response times. However, the system implementers need quite different data to determine how best to improve system performance: for example, frequency of use of the various software components, load on the hardware components, and quality of service being delivered in response to particular user requests. Since these quantities depend on the behavior of unpredictable real users, they can only be obtained while the system is in operation, and must be obtained without affecting that operation.

Information about the environment of a program is also useful in improving ("tuning") an existing program or influencing the design of a future program. For example, in a time-sharing operating system one might want to know the frequency of user requests from consoles, the distribution of memory requirements, or the commands or sequences of commands that are used most often. This information can only be obtained by studying the system while in use by a real user community.

Our implementation of the Informer has been confined to one fairly simple machine, an SDS-940 supporting the Berkeley time-sharing operating system[1]. This machine has only three central registers, and a 14-bit address field which is also the size of the address space. It allows multi-level indirect addressing with the option of indexing at each level. It normally operates in a time-shared mode in which both the user address space and part of the separate operating system space are paged. Any page may be protected from writing or may be "missing": an attempt to write on a read-only page or access a missing page results in a trap, whether the at-

tempt is being made by the user or by the operating system. Programs running in the operating system address space may access the user space with no software intervention and with all the user's protection in force. The 940 has an extensive priority interrupt system (the operating system uses at least seven different levels) and an independent trap mechanism. The simplicity of the machine design simplified our work, but we believe the Informer framework is largely machine-independent and will indicate solutions to a few of the problems which arise from different processor designs at appropriate points in the body of the paper.

2. Goals

The primary goal of the Informer is to provide a facility in which a user of an operating system can gather any specific measurement of that system he desires. The Informer allows users of the facility to name an arbitrary point in the operating system (called a checkpoint) and submit a program which is to be executed in the environment of the operating system each time the flow of control reaches the checkpoint.

The second goal is that no error on the part of a user or his measurement programs can cause any interference with the continued use of the operating system. This implies that any program which is submitted by a user as a measurement routine must be rejected if there is any way it could adversely affect the system. In particular, it must be checked that the measurement routine will never store into the code or data of the operating system or in any other way modify the environment of the operating system. Also, it must be checked that the measurement routine will not exceed what is considered to be the maximum amount of execution time which can be spent at the particular checkpoint from which it is called. This includes overhead due to the Informer itself.

An important goal is to minimize the time and effort required by a user to compose, submit, debug and execute his measurement programs. The approach is to make every step of the measurement process as flexible and interactive as possible. To facilitate composition the Informer will accept measurement programs in any language which can be assembled or compiled into machine languate. There are very few restrictions on the form that measurement programs may take. By automatic (program) verification of the integrity of a measurement program, the time lag between submission of a routine and its acceptance or rejection is on the order of milliseconds. In order to allow measurement programs to be debugged while actually in the environment of the operating system (and for other reasons), a general communication facility exists between measurement programs and user programs.

Finally, it is a goal of the Informer that a measurement once initiated will continue as long as desired. This implies that there is some control over who can modify or delete a measurement routine, and also, that part of the standard crash recovery procedure of the operating system includes code which attempts to permit continuation of those measurements which were in progress before the crash.

3. Program/Informer Linkage

The Informer operates by arranging for specific pieces of code submitted by a user, called measurement routines, to be executed whenever control reaches given instructions, called checkpoints in the program being investigated. (The genesis of the measurement routines is described in the next section of this paper). The part of the Informer which performs this task is called the Loader. The Loader first verifies the validity of the submitted measurement routine, then relocates it into the address space of the operating system, and finally attaches the measurement routine to the operating system by the simple and time-honored technique of patching: refer to figure 1:

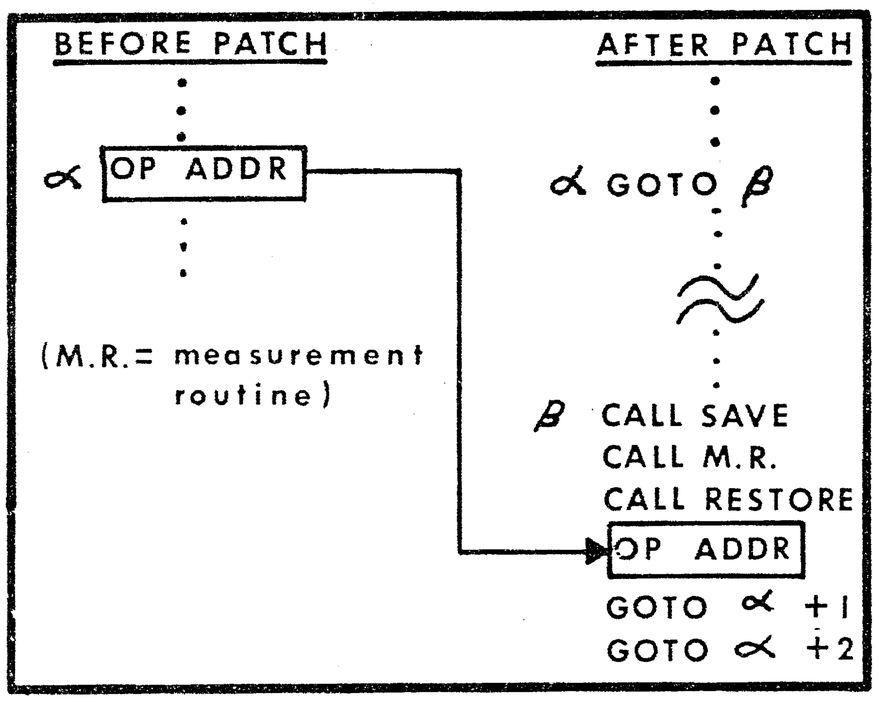

Figure 1. Connecting a checkpoint

In the figure, "save" and "restore" refer to routines to store or load the complete machine state (registers, machine conditions, etc.). However, there are two factors that complicate this scheme. One is the requirement that the operating system be unaware of the presence of the Informer. This means, in particular, that the Loader must be aware of some unused area of the operating system address space in which to place measurement code and its associated data. This information cannot be supplied by the user of the Informer, but must be designated by the operating system author.

A complication to this procedure may occur on machines with variable instruction sizes or with instructions which for some reason are not trivially displaceable to different locations. Patching is only slightly more expensive on such machines.

The other complication to the simple patching method arises from the desire to measure routines at different priority levels: for example, to trace both a routine making entries on a disk queue and an interrupt routine taking them off. There seem to be only two solutions to this problem:

1) Disable interrupts for the duration of the measurement process, including saving and restoring the state. This is highly undesirable.

2) Provide separate saved-state areas for each priority level which is actually being measured. (This can be done with a stack on sufficiently talented machines). These areas must include the return link for the save and restore routines. This approach is clearly preferable.

We have assumed that the operations of saving and restoring the machine state are relatively cheap. On multiple-register machines such as the IBM 360 this is unfortunately not the case. Therefore, at least as long as this style of machine architecture remains in vogue, it is better to require the measurement routines themselves to save and restore the registers they use. The mechanism for enforcing this will be discussed later.

There may be specific reasons to forbid insertion of a measurement routine at a particular location:

1) The location may be in a data area rather than code, consequently patching the location would invalidate the data;

2) The location may be referenced in some non-standard way ("execute" instruction, indirect addressing, modified or used as data);

3) There may be unusually severe timing constraints in that piece of system code, such as a series of instructions which must be performed uninterrupted to set up an I/O operation.

These conditions cannot be detected by the Loader on its own: the author of the operating system must supply the Loader with a table of areas where patches are prohibited. If this table is given in symbolic form, then it is only necessary to re-link the Loader to the operating system when the latter changes, using a standard linking loader.

4. Measurement routines

In contrast to most other system measurement facilities[2], the Informer allows nearly arbitrary pieces of user-written code as measurement routines. In fact, the principal task for the Informer is to guarantee that user measurement routines cannot violate the integrity of the system. This means, in particular, that such routines may not: store or branch into the operating system; perform illegal or privileged instructions; execute for an uncontrolled length of time; modify themselves or other measurement routines; have the possibility of being re-entered before being exited; or read data or code of the operating system which are considered private even to some classes of system measurers. If the Informer finds that the restrictions have all been adhered to, it copies the routine into the area where it will run.

To simplify the checking of user-supplied routines, the Informer forces them to structure their addressing environment into a number of distinct, internally homogenous regions: program, literals, temporary cells, switches, and temporary address cells. The significance of the first three is obvious. Temporary cells used for addresses are distinguished from ordinary temporaries, because the set of legal values for the former is restricted to pointers to ordinary temporaries. Switches provide for changing the connections between checkpoints and measurement routines: the user of the Informer may set a switch to point to a measurement routine or a no-op. Measurement routines are also only allowed to call other routines by executing switches. Roughly speaking, program may be branched to or read; literals may only be read; temporary cells may be read or written; switches may only be executed; and temporary address cells may be stored into with pointers to temporary cells and used as indirect address pointers. When the Informer copies the routine into the measurement code area, it properly relocates references to each region.

The check against illegal types of references to storage and illegal opcodes is made by the Loader. The heart of the procedure is a vector giving type information about each opcode, together with a matrix which specifies which regions may be referenced by which kind of opcode. This table is presented as figure 2. Notice that indexing is permitted only for load and non-memory instructions.

OPCODE TYPE	LEGAL REGIONS			
	PC	TC	TAC	SW
LOAD	↘	↘	↘	↘
STORE		↘	(2)	
STORE INDIRECT			↘	
BRANCH	(1)		↘	
EXECUTE				(1)
NON—MEMORY	(IRRELEVANT)			

PC: Program Cell
TC: Temporary Cell
TAC: Temporary Address Cell
SW: Switch

↘ = allowed
(1) = allowed subject to flow restrictions
(2) = interpreted

Figure 2. Address checking

The contents of temporary address cells can only be checked when the measurement program is running. The Loader transforms every instruction which stores into a TAC in a submitted measurement routine into a subroutine call to a runtime checking routine followed by the actual store. The checking routine ensures that the quantity to be stored lies within the range of TC addresses allowable for this measurement program, and then permits the store to be done.

System calls and nonexistent instructions are illegal. However, certain I/O instructions or instruction sequences may be permitted if the Informer is being used in conjunction with hardware measurement tools: in this case it is clearly desirable to give measurement routines control over this hardware.

In a machine with a large number of state registers, the Loader may identify which registers are actually modified by the measurement routine, and then surround the routine with code to save and restore those registers, as an alternative to always saving all the registers.

Flow tracing within a routine is confined to a simple check that there are no backward branch instructions. An exception is made for a few loop-closing sequences, and then only if the upper bound on the number of repetitions is a constant.

Flow checking is complicated by the fact that measurement routines are allowed to call other measurement routines, through switches as mentioned above. This requires checking that a routine cannot call itself, by scanning the routine (and the routines it calls, recursively) for "EXECUTE switch" instructions, whenever a switch is changed.

To guarantee that no errors can result because of priority interrupts, the Informer requires that each instruction in the operating system have an associated fixed priority level. A routine at a given priority level can only be interrupted to start execution at a higher priority and otherwise runs to completion. There is a table, made available to the Informer by the operating system, which identifies the priority level with each area of measureable instructions in the operating system. This table can be maintained similarly to the table of unmeasureable instruction regions. In fact, these two tables can be merged in a natural way. Thus a priority level can be assigned to each measurement routine as well, which is the priority of the instruction where the patch was made. However, the measurement routine may call other routines by executing a switch. Since measurement routines are assumed not to be reentrant, the Informer must ensure that no routine can be called from routines or checkpoints of differing priorities. The check must be made again whenever the setting of any switch is changed. The check essentially consists of tracing through the code looking for switch executions, as for flow checking: a called routine or switch with no assigned priority receives the priority of the calling routine, and an already assigned priority different from that of the caller causes an error.

A related problem arises in connection with two types of Informer subroutines described later: they are executed as part of the measurement routines, so they must exist in multiple copies, or at least some part of them which locates temporary storage for the subroutine must have several copies. However, it is very wasteful to provide a copy for each different priority level that exists in the machine. To avoid this, the Informer maintains a list of active priority levels, i.e. those priority levels such that some piece of code at that level has been patched. The number of copies of these routines is just the maximum number of simultaneously measurable priority levels, an assembly parameter of the Informer.

The timing check consists of making a worst-case estimate of the time required to execute the measurement routine. If loops are not involved, this is just the sum of the individual instruction times (with upper bounds for instructions with data-dependent timing). Since the starting and ending values of loops are required to be constants, the longest possible time spent in a loop is just the maximum number of iterations times the maximum time per iteration; of course, this calculation may have to be nested for nested loops. If the routine executes a switch, the Informer must recalculate all the maximum times every time the user changes the setting of a switch.

The Informer incorporates a map of the operating system similar to the one which associates priority levels with locations in the code, which gives maximum allowable times for measurement routines inserted at various places in the system. For example, it might be allowable to interrupt the processing of a system call (SVC, SYSPOP, ...) for several milliseconds, whereas the time limit in a drum interrupt routine might only be 50 microseconds. Once the maximum time for a new measurement routine has been computed, the Informer will reject the routine if this time exceeds the maximum allowable time for the location to be patched minus the sum of the execution times of the routines already patched into the region.

For the purpose of allowing a measurement routine to alert a user program about a critical condition, one switch is specifically defined such that if a call to it is included in a measurement routine,

when executed will send a wake-up to the user program. The implementation of the subroutine will vary from system to system, but the intent is to allow a user process to wait in a "blocked" condition, which does not consume any CPU or core resources, until the wake-up arrives.

Two kinds of runtime errors cannot be anticipated at load time. One is an addressing violation which may be caused by an indexed instruction. The other is an attempt to store an inva-

lid pointer into a TAC. In keeping with the Informer philosophy of allowing maximal flexibility, each of these operations will store an error message in a fixed TC if the operation fails. In this way the measurement routine itself can decide what to do about the error.

5. User interface

When attempting to design and construct a complex facility, a reasonable approach is to decide on a set of primitive operations, combinations of which will provide the full power desired. This philosophy has the advantage of making implementation of the facility simpler, while allowing for flexible and perhaps unanticipated use of the facility. Accordingly, the interface between the user and the Informer consists of the following primitive functions:

NEW MEASUREMENT PROGRAM (<user-space address>)

Calling this function requests that the program which exists in the user address space at the specified address be copied into measurement program space in the environment of the operating system. If the program is found to be unacceptable, then the function returns a value which explains the reason for rejection. If the program is valid, the function returns a value which may later be used as an identifier of the submitted routine.

ATTACH(<identifier>,<operating system address>)

This function creates a checkpoint at the specified address, and connects it to the specified routine if the request is valid.

DELETE MEASUREMENT ROUTINE(<identifier>)

All checkpoints attached to the specified routine are disconnected and the routine is removed from measurement program space.

DETACH CHECKPOINT(<operating system address>)

If the specified address is a checkpoint, the checkpoint is disconnected from its associated switch.

SET SWITCH(<switch number>,<identifier>)

The identified routine is associated with the specified switch.

CLEAR SWITCH(<switch number>)

If there is a routine associated with the specified switch, it is detached. The switch is changed to a NOP, therefore any routine which executes that switch will be affected.

BLOCK()

This function causes a pause in the user program until one of his measurement routines causes a wake-up.

EXECUTE IN CONTEXT(<instruction>)

This function executes the specified instruction as if it were part of a measurement routine. The central registers are modified only if the execution of the instruction does it. The instruction may be any instruction which can occur in a measurement routine, including a SWITCH execution. The primary purpose of this function is to permit a user program to have access to the intermediate results of measurement routines.

ACCESS MEASUREMENT SPACE(<user space address>)

This function causes the measurement space to be placed in the virtual address space of the user, with read-only status.

Also to be included in this set of primitives are functions which when called will return with descriptions of the state of the user's measurement space. For example:

IS CHECKPOINT?(<operating system address>)

This function will return either yes or no depending on whether the specified address is currently serving as a checkpoint.

IS CONNECTED?(<identifier>)

This function returns either no or a list of checkpoints to which the specified measurement routine is connected.

6. Summary

Users may construct a measurement routine in any language, and then requestthat the routine be relocated into the address space of the operating system (observed program) to be executed whenever control reaches a specified operating system location. The Informer, before granting a request, automatically does a complete check of the submitted routine that there is no way that the routine could cause an error in the operating system. Capabilities exist to allow communication between measurement programs and programs of the user which submitted them. These facilities may be used for debugging, notification of events, or coordination of data buffering.

The Informer is two separate modules: the Loader which is invoked only during the insertion or deletion of measurement routines, and the Runtime which is a small package of routines which must be resident in the address space of the observed program (save and restore state, TAC verification, etc.). The Loader, which does the necessary verification of measurement requests is not resident during measurement and therefore cannot be considered as perturbing the measured system.

7. Example

Consider the following experiment to learn the average length of disk transfer requests. A measurement routine is constructed which is called each time the disk driver is entered. The measurement routine uses two temporary cells labelled COUNT and SUMSIZE. For each request, the routine increments COUNT and adds the size of the transfer to SUMSIZE. When the user feels a sufficient sample time has passed, he may execute code to read the values of COUNT and SUMSIZE and compute the average. This measurement routine

would be very simple and would hardly perturb the running system.

Other measurement schemes might require that each disk transaction be recorded for later analysis. Avoiding the collection and analysis of large volumes of data is clearly desirable.

8. Conclusion

An implementation of the Informer[3] has been in use on the Berkeley 940 time-sharing system since 1969. Experience has shown that the facility is an embodiment of the design goals. Flexible measurement programs have been written with a great deal of convenience. Experiments completed or in progress include: frequency and duration measurement of system calls, response times to interactive commands, analysis of operating system overhead, and an analysis of user characteristics analagous to the study done in[4]. The overhead due to the Informer is negligible. (Specifically, the overhead introduced by saving and restoring of the state of the machine at checkpoints is roughly 140 microseconds; this is quite reasonable if the events being measured only occur every few milliseconds.) All other overhead is due to the measurement routines themselves. Measurement routines may do analysis themselves, or by using the communication capabilities, pass data to user programs by double buffering techniques.

Implementation of the Informer required about four man-months of programming effort. Maintenance of the facility has been minimal, and only entails updating the Informer tables when modifications are made to the operating system. The Informer, once debugged, has never caused a system crash as far as we know.

Measurement routines are often constructed to be invoked each time the operating system makes access to a given datum or data structure. With the Informer, as described, that would require a checkpoint at each separate instruction location from which access to the data structure is possible. A hardware capability to trap on references to specified data words would allow Informer routines to be invoked only when the data was actually accessed. The Burroughs B5000 meta-bit facility is an example of an existing computer which could allow tracing of data structure accesses as well as the flow of control.

In the case of an environment where there is system information which is considered privileged even to system measurers, then the freedom of measurement routines to make arbitrary read accesses could be restricted. Any privileged regions could be made inaccessible in the map of the address space. All indirect reads would be required to be made through temporary address cells. (There would now be two classes of such cells, read TACs and read-write TACs.) Also, checkpoint locations might be restricted when the mere information that control has reached a specific location is considered privileged.

The primary lesson learned from the implementation, aside from certification of the design, has been that good documentation is essential to convenient measurement. While the Informer requires users to understand only those aspects of the measured program that they wish to measure, documentation must exist from which they may learn.

The techniques of program verification described here may well have implications beyond system measurement.

The concept of the Informer, as a facility to allow submission of user written code for measurement of the operating system with full protection against system disturbance was introduced by Remi F. Despres in 1968. The authors wish to acknowledge him as the originator of the Informer project.

References

1) Reference Manual - Time-Sharing System
Deutsch, P., Durham, L., and Lampson, B.
Project Genie Doc. R-21
Univ. of Calif., Berkeley, July, 1967.

2) A Bibliography on Performance Evaluation
Ferrari, D.
Computer Systems Research Doc. P-1.0/CSR
Univ. of Calif., Berkeley, December, 1970.

3) The INFORMER Users Manual
Grant, C. A.
Computer Systems Research Doc. R-1.0/CSR
Univ. of Calif., Berkeley, April, 1971.

4) JOSS: 20,000 Hours at a Console - A Statistical Summary
Bryan, G. E.
Fall Joint Computer Conference, 1967.