MARK E. SCHMID

FAULT TOLERANT COMPUTING VIA MONITORS

The growing use of embedded computers for both military and commercial applications has increased the importance of reliability in computer design. A generic technique called computer monitoring is explored for its fault tolerant capabilities, with emphasis on transient fault detection. The success of a particular monitor (the program flow monitor) is established through experimentation, and issues related to its implementation are examined.

THE NEED FOR FAULT TOLERANCE

The components used in early computers had relatively short lifetimes. Room-size computers using vacuum tubes and relays reflected the low reliability of their fundamental components. Calculations performed by computers needed to be scrutinized carefully; in some cases, the calculations were used only after verification by manual techniques.

Computer designers, recognizing the need for greater reliability, began developing techniques for fault avoidance and fault tolerance. Fault avoidance stresses the prevention of faults through the use of reliable, tested parts and conservative design practices that will tend to extend a part's lifetime. Although fault avoidance contributed to greater reliability, it alone was not enough. Fault tolerance, the ability to withstand an error and continue functioning correctly, became a common design practice primarily because it was needed to provide a reasonable availability of a properly functioning computer.

When the transistor and, later, the integrated circuit were introduced, the impact on the reliability of computers was so dramatic that designers found fault-avoidance techniques to be sufficient for most computer applications. The extra expense incurred in fault tolerant designs was no longer cost effective. But the advent of this new generation of reliable computers inspired a new range of application in which people would depend on the proper operation of the computer. Such dependency began in small experimental applications but has since spread throughout the Department of Defense and industry. Examples in industry that affect a very broad population are telephone systems, banking operations, and even new cars: all are highly dependent on computers. Certain applications, particularly life-critical ones, have always demanded fault tolerance, not just fault avoidance. But the current pervasive use of computers has made computer failure intolerable in a much wider range of applications.

BACKGROUND – FAULT TOLERANT TECHNIQUES

Fault tolerant computer systems have traditionally relied on various forms of redundancy to enable correct operation despite the occurrence of a fault. A technique known as triple modular redundancy, shown in Fig. 1, uses three separate modules to perform a desired function independently. Each module reports its results to a voter, which produces a majority result. This protects against errors in a single module and in most cases will allow the detection of multiple errors (because faulted modules will be unlikely to fault in exactly the same way). However, the effects of multiple module errors are implementation-dependent, and, if total protection against multiple failures is desired, higher degrees of redundancy (more modules) are needed.

The voter in Fig. 1 is a potential source of failure because it is not implemented redundantly. In any modular redundant system, a point will exist where a "decision" must be made to create a single output from the redundant inputs. Normally, the voter will be much simpler than the modules it votes on, giving it an inherently greater reliability. Its simplicity and the use of fault avoidance are the keys to its reliability.

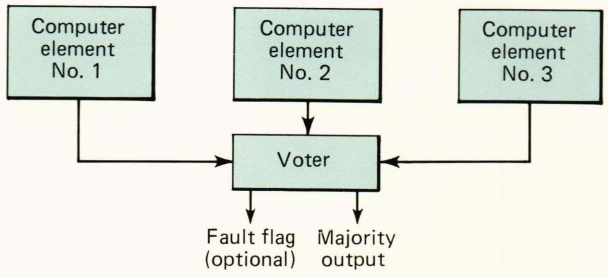

Figure 1—Triple modular redundancy uses a majority result that is produced by "counting the votes" from three independent computers.

Redundancy is still the best technique available for providing nonstop, error-free computing with reasonable certainty. Its only drawback is the cost of duplicating so much hardware. The cost extends beyond the obvious monetary one. Many systems that require fault tolerance, particularly in the military, also have severe restrictions on their physical size and power. Redundant designs may be impractical in such situations. At various times, the gains in Very Large Scale Integration technology promised to allow "on-chip redundancy," reducing the penalties associated with redundant designs. But with few exceptions, higher gate densities have been used to produce more complex computers, not redundant implementations of simple ones. Thus alternatives to redundancy-based fault tolerance have become of great interest, both in commercial and military markets.

Recently, the study of alternatives to redundancy has become a very active research area, with a technique called monitoring being one of the primary interests.[1] This is also the focus of a three-year research and development project at APL.

Many emerging computer applications (especially those using microprocessors for real-time control) have reliability requirements that are less stringent than the ultrareliable, nonstop nature of redundant systems with voters. Often brief periods of errant operation can be tolerated, as long as correct operation is eventually resumed or a shutdown is completed.

Two practices are currently used to achieve this level of reliable operation: the watchdog timer and self-test. The watchdog timer[2] is a device that provides a "sanity check" on a processing element. The watchdog requires a periodic update to be delivered by the computer. If the update is not made within a certain length of time, the watchdog asserts that the computer is faulty. This may result in an attempt to restart the program, a shutdown of the computer, an attempt to start a standby processor, or other alternatives.

The self-test function is a program that is designed to test a large percentage of the capabilities of a processor or a system. These tests often have diagnostic capabilities that also help speed maintenance. Many commercial products use self-test capabilities to indicate the product's status before use. This is referred to as a power-up test. Self-tests may also be executed periodically, allowing the integrity of the system to be verified during use. This use of the self-test is called confidence testing.

There is an interesting distinction between these two techniques. The self-test generally is a thorough test of the functionality of the hardware. It can determine when a part of a processor or system has a permanent defect. The watchdog timer, on the other hand, continuously requires the computer to perform a very simple task and is considered a "monitor" of the computer. Because that task (periodically reporting to the watchdog) is written as part of the application program, most events that severely disrupt execution of the program will cause the watchdog timer to report the system as faulty. The event causing the disruption may be a permanent fault of the type that the self-test would catch, or it might be a transient fault – one in which no permanent damage is done but that causes the normal operation to be upset. In cases where there is no permanently failed part, even a periodic application of a built-in test would not detect the improper operation.

Built-in tests usually do a very thorough job of checking functionality, something that watchdog timers do only indirectly and incompletely. The capability of the watchdog timer to respond to transients would not be significant if the prevalence of transient faults were not so great. But data accumulated from a number of experimental systems indicate that the occurrence of transient faults is 20 to 50 times greater than permanent faults.[3]

THE GENERAL MONITORING CONCEPT

Because of the relatively high frequency of transient faults, a generalized concept of computer monitoring has been developed; it is shown in Fig. 2. A monitor uses information about the program being executed to check constantly for correct operation. In the case of a watchdog timer, the information used is knowledge of the periodic update to the watchdog. Much more complex monitors have been devised, with the limiting case being a duplicate of the processor checking every aspect of operation.

A comprehensive overview of the monitoring approach to fault tolerance is presented in Ref. 1. Highly ranked among the relevant issues are characterizations of performance deviations to which monitors can be sensitized and assessments of monitor effectiveness. The APL effort has focused on observation of performance deviations and experimental evaluation of monitors.

Faults can be viewed at different levels. At the lowest level, noise on a transmission line, a marginally operating transistor, or the impact of an alpha particle on a memory cell may be the underlying physical cause for faulty operation. The physical fault may propagate upward, perhaps causing execution of an errant instruction that in turn might cause a system action that could be externally discernible. The various lev-

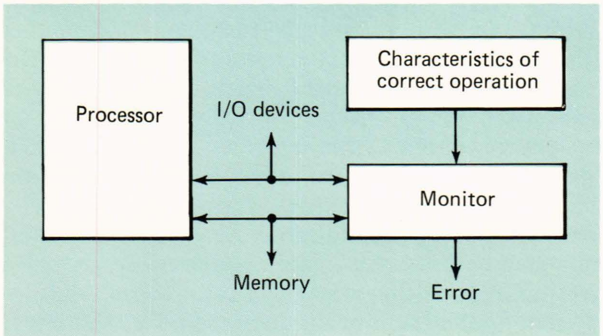

Figure 2—General monitor concept. The monitor continuously compares processor activity with stored characterizations of the correct operation.

els of fault manifestation have been characterized in Ref. 4, which presents a taxonomy of fault tolerance. The monitoring devices considered here are sensitized at the informational level, where the computer program may provide an extensive description of normal operation. A disruption at this level will be referred to as an upset.

EXPERIMENTATION

There has been little previous experimental work regarding characterizations of upsets in computer systems. This problem was addressed with two series of fault-injection experiments on a representative microprocessor-based system. In the first series, the system executes a small machine language program; in the second series, a much larger software environment developed with a high-level language is used. In each series, faults from a broad spectrum of fault conditions are applied in individual experiments to an otherwise properly functioning computer. An extensive instrumentation complex records data associated with the upsets that result from the injections. These data are used to characterize the upsets and to serve as a benchmark for the performance of candidate upset monitors.

In both experiment series, one particular class of upsets is seen to be dominant. However, full, rigorous designs of monitoring devices sensitized to such upsets are, in general, prohibitively complex and/or memory intensive. Hence, there is motivation to consider more practical upset monitor designs. This is done for a particular class of upsets in the form of "compressed monitors." Here, the database that characterizes correct operation is reduced to make implementation more reasonable. Interestingly, rather extensive compressions yield upset monitors with upset detection capabilities that approach those of a full, rigorous implementation.

A testbed containing a small computer system based on a Z80 microprocessor was developed for experimentation. Single faults are injected into the system under a wide range of conditions. Since so little is known about the details of faults that actually occur, a set of approximately 700 unique fault-injection conditions is used for each of the two experiment series. The faults are generally transient and are injected in a single microprocessor line primarily because they may cause more subtle perturbations of computer operation than permanent and multiple line faults. Detection of an upset caused by such a fault is likely to be a lower bound on upset monitor sensitivity.

Data are collected from separate "gold" and "faulted" test runs as illustrated in Fig. 3. A gold run is initiated by starting execution from a known state. Microevents (e.g., central processing unit cycle states) and macroevents (e.g., input/output) are recorded over an interval that starts just prior to the fault injection. Faulted runs occur on the same testbed under identical conditions but with a fault injected during the run in a precise, reproducible manner. Computer events are recorded as in the gold run.

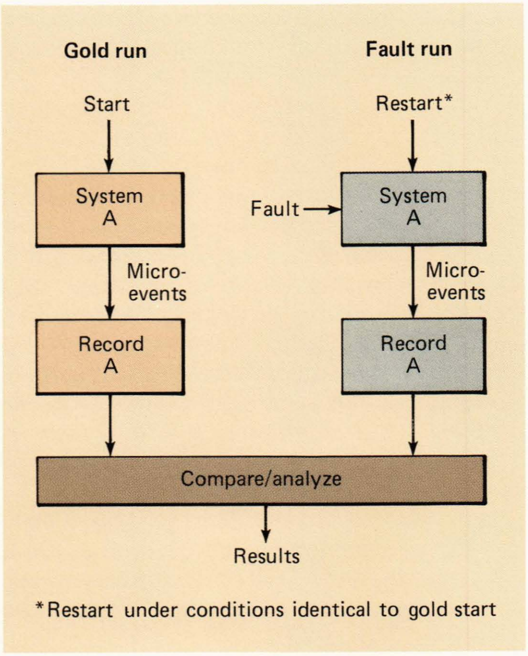

Figure 3—Experimental procedure. Data are collected from an unfaulted period of execution (gold run). Further data are collected over the same period, but with a precisely injected fault (fault run). The data from both runs are analyzed to characterize manifestations of the fault.

UPSET CLASSIFICATIONS

For each experiment, the induced upset is characterized by means of a cycle-by-cycle comparison of the recorded data for gold and faulted operations. Where differences exist, the primary features are examined and an attempt is made to associate them with one or more upset classes. Naturally, the classes of most interest are those that can be described in terms of highly observable features of operation, since these correspond to the most promising upset monitors. Others, such as data upsets, appear to be unmonitorable without redundant hardware because of the difficulty in characterizing correct operation. Certain upset classes tend to cover a significant majority of the upsets observed. However, some recorded upsets cannot be classified because they correspond to differences between gold and faulted runs that are very complex, and a general classification is not realistic from the perspective of external monitoring. Fortunately, such cases are the exception; only 20% of the upsets were unclassifiable.

Seven monitorable upset classes were identified. These classes are listed and explained in Table 1. Figure 4 shows the extent to which each upset class covered the upsets produced by an injection. Also reported is the average time (in microseconds) between fault injection and the observance of a classifiable upset, called the time of emergence.

Time of emergence is obviously related to latency of detection by associated upset monitors. In Fig. 4, the first bar, labeled "All," indicates the percent of upsets that could be classified in any of the monitor-

Table 1 – Upset classes.

IPF (invalid program flow)	Improper sequence of instructions
IOA (invalid opcode address)	Fetch of an instruction from a noninstruction address
UNM (unused memory)	Memory access to an existent but unused memory area
IRA (invalid read address)	Read access (for data) to an instruction area, or unused or nonexistent memory
IOC (invalid opcode)	Fetch of an illegal instruction, or an instruction not part of the subset used in the specific task software
IWA (invalid write address)	Attempt to write into memory not designated as alterable
NEM (nonexistent memory)	Access to a location with no memory

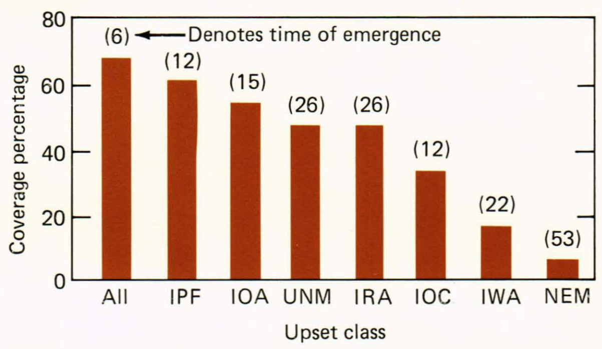

Figure 4—Coverage by monitorable upset class. Seventy percent of the upsets fell into at least one monitorable upset class. Bar heights indicate the percentage of upsets that were covered by a class. Numbers in parentheses report the average time, in microseconds, between the injected fault and the behavior indicative of the upset class. (See Table 1 for full monitor names.)

able categories. The most predominant single category is invalid program flow. Note that a given upset could often be characterized as fitting more than one classification, resulting in significant overlap between classes.

FULL PROGRAM FLOW MONITORS

In both series of experiments, invalid program flow characterized the greatest number of observed upsets and also had the minimum time of emergence. This provided strong motivation for an investigation of practical and efficient implementations of program flow monitors.

Program flow is defined as the sequence of instructions that are executed by the computer. It is characterized by the addresses associated with the instructions and may be described by source-destination address pairs. The source and destination addresses are defined, respectively, as the first and second addresses of a pair of sequentially executed instructions. A given destination address may have more than one possible source address (e.g., the first instruction of a subroutine) or a given source address may be paired with multiple destinations (e.g., a return instruction). During execution, the program flow monitor compares known valid source-destination address pairs with pairs that are observed.

Figure 5 shows realizations of a full program flow monitor. The term "full" is used to describe these realizations because they completely identify all valid and invalid address combinations. That is, no significant attempt has been made to condense the source-destination pair representation. In the figure, the number of valid address pairs is represented by v, and the number of bits in an address specification is w.

Content Addressable Program Flow Monitor

Figure 5a illustrates what is perhaps the simplest full program flow monitor in concept, but one that is difficult to implement. The valid source-destination pairs are concatenated and entered into a memory that is content addressable. An observed source-destination pair addresses the memory to determine whether the pair is valid. Although this method requires the least memory of the depicted realizations ($v2w$ bits), it imposes the greatest complexity on the mechanism that validates an observed address pair.

Address Universe Program Flow Monitor

Figure 5b displays a memory intensive realization of a full program flow monitor. A 1-bit-wide memory table of all possible combinations of source and destination addresses (2^{2w} bit memory) is used to represent the valid address pairs. Observed source and destination addresses are concatenated to form an index into the table, and the memory output directly indicates the validity of the address pair.

Indexed Program Flow Monitor

Figure 5c is a compromise that uses indexing to reduce the memory required for the address universe method. An observed address pair is tested by using the source address to find the start of a set of valid destinations. The valid destinations are compared to the observed destination to determine the observed pair's validity. Such a realization of a full program flow monitor requires $2^w(\log_2 v) + wv$ bits.

COMPRESSED PROGRAM FLOW MONITORS

The full program flow monitor realizations described in the previous section require either extensive memory or device complexity. In general, this represents a major drawback to their utilization. Hence, an investigation of more practical program flow monitors was warranted. The monitors discussed here are referred to as compressed program flow monitors. The term compressed is used because the source-address

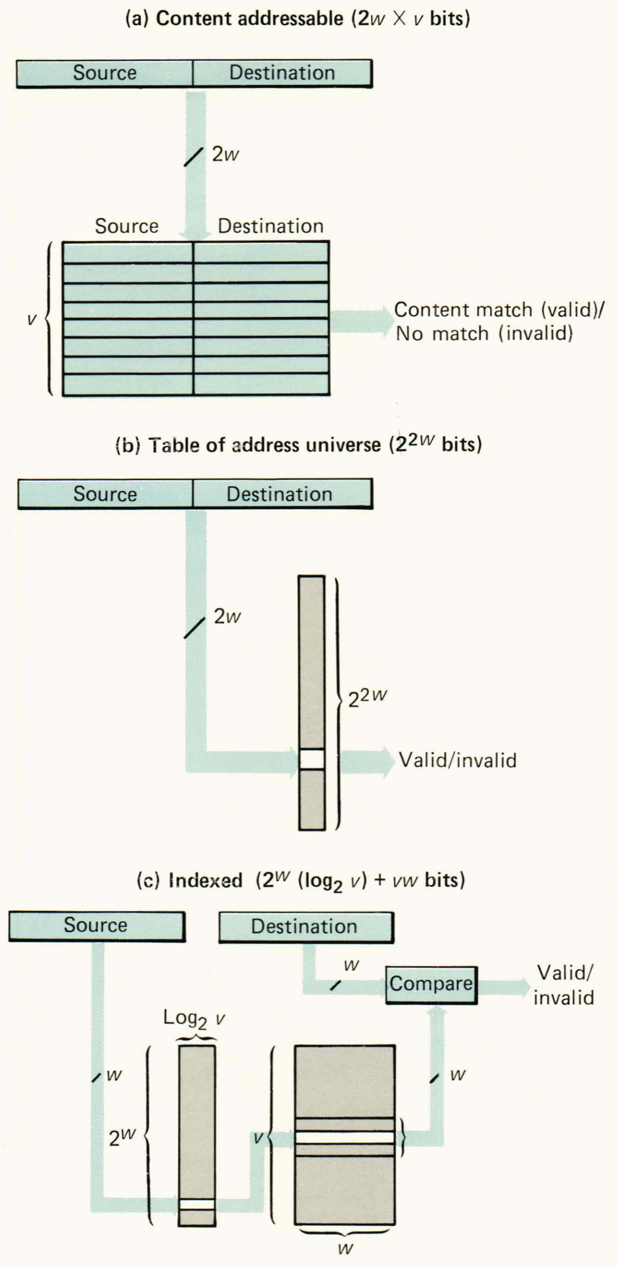

Figure 5—Three full program flow monitors. (a) A table of valid address combinations is searched for a match to the current combination. (b) The current address combination is used to index a table of all possible address combinations. The indexed value indicates the validity of the combination. (c) The current source address indirectly indexes a table to determine a set of valid destinations. A search of that set is made for the current destination address.

pair representations that the designs utilize are condensed relative to those of full program flow monitors.

A compressed program flow monitor is illustrated in Fig. 6. Note the similarity to the address universe full program flow monitor (Fig. 5b) in the use of a "bit map" to indicate the validity of source-destination address combinations. However, instead of having a valid/invalid entry in the map for all possible source-destination address combinations, the compressed program flow monitor uses a compression or coding scheme to reduce the domain of combinations over which valid/invalid entries must be provided. For evaluation purposes, each individual source-destination address combination was compressed to a representation of length m (where m is no greater than w, the number of bits in one address specification), resulting in a bit map of 2^m bits. For each of the compression schemes, the bit map size may be varied so that ranging degrees of compression could be evaluated. An analogous technique, called "hashing," is used extensively in software design to achieve a fast search capability with a low storage requirement.

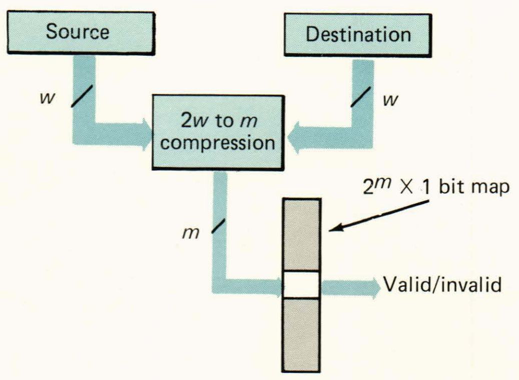

Figure 6—Compressed program flow monitor. Source and destination addresses are combined and compressed to form an index into a "bit map" of all possible compressed combinations. The indexed value indicates the validity of the combination.

As a benchmark, it is interesting to compare the memory requirements of compressed program flow monitors with those of the indexed full program flow monitor of Fig. 5c, which seems to be the most practical in terms of memory intensity and complexity of all the full program flow monitors. Assuming that there are w bits in an address, if the source address is used to index a memory table and if the number of valid and specifiable address combinations is v, then the memory required for an indexed full program flow implementation is $vw + 2^w(\log_2 v)$ bits. Since the number of valid address combinations may indeed approach the size of the address space, the memory required can be on the order of $w2^w$ bits, which is often the size of the monitored system's memory. Hence, a compressed program flow monitor requiring only 2^m bits (with m less than w) compares quite favorably, assuming, of course, that it can be shown to have similar coverage and latency figures. As will now be discussed, this is indeed the case.

The four compression methods listed in Table 2 have been evaluated for effectiveness. This evaluation used the experimental data described previously, which enabled the wide variety of fault conditions given in Table 1 to be used in assessing the performance of the compressed program flow monitors. Because the cost of this monitor is governed primarily by the size of the bit map, the degree to which this additional storage could be reduced was of great interest. For descrip-

Table 2—Program flow compressions.

Type	*Method*
Difference	Difference between source and destination addresses
Concatenate	Concatenation of lower half of source and destination addresses
Swap exclusive-or	Exclusive-or of byte swapped source with destination
Parity coding	Parity bits generated from concatenated source and destination

tive purposes, we will define the storage ratio as the ratio of the memory used by the bit map to the memory required for program storage (this does not include data storage). For each compression method, six storage ratios were examined, ranging from 1 to 1/128.

All compression monitors are based on combining source-destination address pairs in various ways. A very straightforward method is the "difference monitor," which uses the difference between source address and destination address to specify valid combinations. For example, if a combination of source address equaling 000100_2 and destination address equaling 000110_2 occurred, then 111110_2 (2's complement form) would represent a valid source-destination pair.

The next two compression methods involved swapping and exclusive-or operations on portions of the valid source and valid destination addresses. The "concatenated monitor" uses a compression that is a concatenation of the lower $w/2$ bits of the source address with the lower $w/2$ bits of the destination address. The "swap exclusive-or monitor" is based on a compression wherein the lower $w/2$ bits of the source address are first swapped with the upper $w/2$ bits, and an exclusive-or of the swapped source and destination addresses is then performed. In the above compression methods, representations of less than w bits are obtained by masking off the high-order bits as necessary.

The last compression method is a parity coding. To achieve the m bits to be used by the parity encoding monitor, groups of $2w/m$ bits are combined to create each of the m parity bits. Alternating bits are taken from source and destination addresses, starting with the high source bit and low destination bit, to get an interleaving effect. For example, again suppose that the source address is 000100_2 and the destination address is 000110_2. If $m = 4$, then three bits at a time are combined, resulting in 0010_2 as the representation.

Figure 7 shows the experimental results for the four different compressed program flow monitors for various storage ratios (i.e., the bit map was compressed to various sizes). It appears that, although the absolute performance for the different compressions varies, the performance of each approaches the coverage of the full program flow monitor as the degree of compression is reduced. The parity coding and swap exclusive-or compressions demonstrate moderately higher coverage and, although not indicated in Fig. 7, also exhibit the minimum latency. As points of reference, the memory requirements for the content addressable and address universe full program flow monitors are also shown on the horizontal axis. With the possible exception of the content addressable full monitor, these memory requirements are excessive. However, as mentioned earlier, the primary disadvantage of a content addressable program flow monitor is its significant hardware complexity.

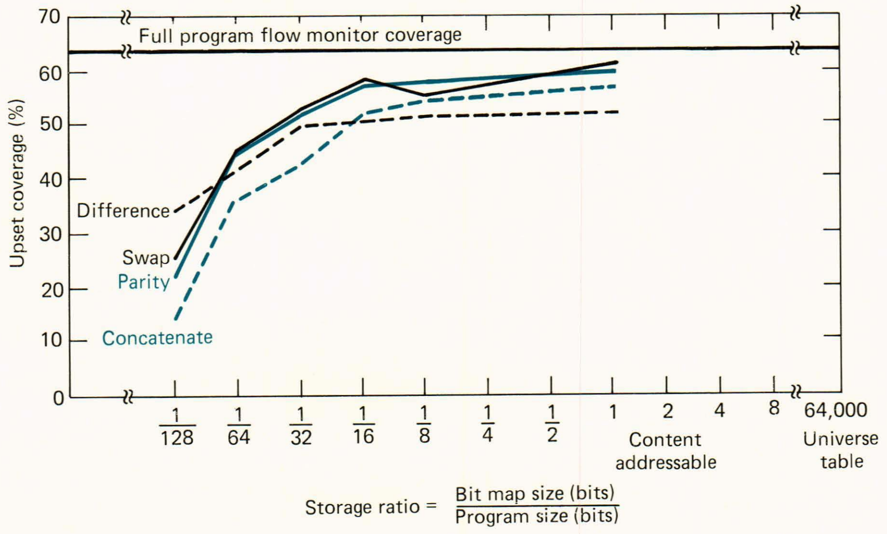

Figure 7—Performance of various compressed program flow monitors as a function of storage ratio. Compressions that reduce the bit map to a size greater than 1/16 the program storage size capture much of the capability of a full program flow monitor. Storage ratios for content addressable and address universe full program flow monitors are shown for comparison.

CONCLUSIONS

A large set of experiments, where single-line transient and permanent faults were injected into a microprocessor, was conducted to characterize the upsets induced by injected faults. It was found that the most prevalent monitorable behavior during the upsets was invalid program flow. This conclusion led to an investigation of program flow monitors.

Straightforward program flow monitor implementations were excessively large and/or complex because of the difficulty in representing valid program flow. As a result, four methods for reducing the valid program flow representation were developed and evaluated using the experimental data from the upset characterization study. The results were quite favorable. Figure 7 indicates that compressed program flow monitors may provide coverage of upsets nearly as well as a full program flow monitor but with a fraction of the full monitor's memory (and cost).

Monitoring techniques, and specifically program flow monitoring, can provide a satisfying degree of fault tolerance to systems that otherwise might employ no fault tolerance technique at all. Preliminary results from experiments conducted on a recently developed program flow monitor demonstration system indicate the monitor's performance to be better than the data presented here. Such results, if confirmed, would give even more impetus to the development of monitoring techniques.

REFERENCES

[1] A. Avizienis, "Fault Tolerance by Means of External Monitoring," in *Proc., AFIPS Conf.* **50**, pp. 30-40 (1981).

[2] W. H. Toy, "Error Switch of Duplicated Processor in the No. 2 Electronic Switch System," in *Proc. 1971 International Symp. on Fault Tolerant Computing,* pp. 108-109 (1971).

[3] D. P. Siewiorek and S. Swarz, *The Theory and Practice of Reliable System Design,* Digital Press, p. 18 (1982).

[4] A. Avizienis, "Framework for a Taxonomy of Fault Tolerance Attributes in Computer Systems," in *Proc. 1983 Computer Architecture Symp.*, pp. 16-21 (1983).

RELATED PUBLICATIONS

B. Courtois, "Some Results About the Efficiency of Simple Mechanisms for the Detection of Microcomputer Malfunctions," in *Proc. 1979 International Symp. on Fault Tolerant Computing,* pp. 71-74 (1979).

S. Z. Hassane, *Signature Testing of Sequential Machines,* Report 82-18, Stanford University Center for Reliable Computing (1982).

D. J. Lu, "Watchdog Processors and VLSI," in *Proc. National Electronic Conf.* **34**, pp. 240-245 (1980).

M. Namjoo, *CERBERUS-16: An Architecture for a General Purpose Watchdog Processor,* Report 82-19, Stanford University Center for Reliable Computing (Dec 1982).

M. Namjoo and E. J. McCluskey, "Watchdog Processors and Capability Checking," in *12th International Symp. on Fault Tolerant Computing,* pp. 245-248 (1982).

M. E. Schmid, R. L. Trapp, A. E. Davidoff, and G. M. Masson, "Upset Exposure by Means of Abstraction Verification," in *12th International Symp. on Fault Tolerant Computing,* pp. 237-244 (1982).

ACKNOWLEDGMENTS—Much credit for the accomplishments of this project must go to the other primary participants, R. L. Trapp and A. E. Davidoff of APL, and G. M. Masson of The Johns Hopkins University G. W. C. Whiting School of Engineering. This work was supported as an Independent Research and Development Project under general contract N00024-83-C-5301 with the Department of the Navy.