Concurrent Error Detection Using Watchdog Processors—A Survey

AAMER MAHMOOD, MEMBER, IEEE, AND E. J. McCLUSKEY, FELLOW, IEEE

Abstract—**This is a survey of concurrent system-level error detection techniques using a watchdog processor. A watchdog processor is a small and simple coprocessor that detects errors by monitoring the behavior of a system. Like replication it does not depend on any fault model for error detection. However, it requires less hardware as compared to replication. It is shown that a large number of errors can be detected by monitoring the control flow and memory access behavior. Two techniques of control flow checking are discussed and compared to the current error detection techniques. A scheme for memory access checking based on capability-based addressing is described. The design of a watchdog for performing reasonableness checks on the output of a main processor, by executing assertions, is also discussed.**

Index Terms—**Capability-based addressing, concurrent checking, control flow checking, coprocessor, distributed computing, executable assertions, microprogramming, parallel computing, signature analysis, system-level error detection, watchdog processor.**

I. Introduction

CONCURRENT (on-line or implicit) error detection techniques used in digital systems can be divided into two classes: circuit-level techniques and system-level techniques. The use of single error correcting and double error detecting codes for memories, parity bits for data buses, residue codes for ALU's, and self-checking circuits are all examples of circuit-level techniques [58]. Capability-based addressing [17], watchdog timers [47], fault-tolerant data structures [63], and use of replication (ESS-1A [60], FTMP [23], SIFT [66], C.vmp [57], and N-Version programming [9]) are some of the examples of the techniques used to detect errors at the system level.

A. Watchdog Processors

A watchdog processor [35], [42] is a small and simple coprocessor used to perform concurrent system-level error detection by monitoring the behavior of a main processor. The watchdog is an extension of the idea of a watchdog timer [10], [47], [48]. The organization of a system using a watchdog processor is shown in Fig. 1. Error detection by means of a watchdog is a two phase process. In the first phase (setup phase) the watchdog is provided with some information about the processor or process to be checked. During the second (checking) phase, it monitors the processor and collects the relevant information concurrently. Error detection is done by comparing the information collected concurrently with the information provided during the setup phase. The information provided to the watchdog to detect errors can be about the memory access behavior [43], the control flow [15], [26], [27], [36], [44], [46], [55], [59], [65], [67], the control signals [14], or the reasonableness of results [28], [37], [41], [49].

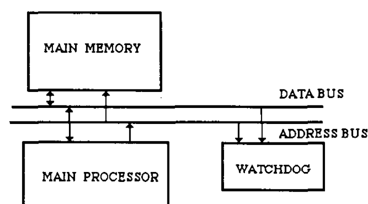


Fig. 1. Error detection using a watchdog.

In terms of complexity, the watchdog lies between the current circuit-level and system-level techniques. Like other system-level techniques it does not require a simplistic fault model (single stuck-at, unidirectional, etc.), but it is cheaper than replication. Moreover, as the checking is done concurrently, the performance of the system does not suffer significantly. The use of watchdogs for concurrent (on-line) testing can be compared to functional (off-line) testing of microprocessors [64]. In both cases, the checking is done at a level higher than the circuit level. The watchdog can be added to any system without major changes to the system. If the system being monitored uses circuit-level error detection techniques, then the use of a watchdog can increase the reliability of the system by detecting errors which escape detection at the lower level. Another advantage of using a watchdog processor is that the checking circuitry is totally independent of the checked circuitry. This provides protection against common or related errors because of design diversity [6]. Other schemes use complementary logic [54] or processors from different manufacturers, as in Boeing 737-300 airplane [68], to overcome this problem. The very use of a watchdog processor provides protection against such errors. The use of a watchdog not only detects hardware errors but also software and design errors if reasonableness checks are performed on the output of the checked processor. Recently there has been a trend towards distributed computing using dedicated processors to perform specialized functions like floating-point calculations and input/output processing. [13] is

Manuscript received December 12, 1985; revised July 31, 1986 and January 26, 1987. This work was supported in part by the National Science Foundation under Grant DCR-8200129 and by ROLM ML-SPEC Computer, San Jose, CA.

A. Mahmood was with the Center for Reliable Computing, Computing Systems Laboratory, Stanford University, Stanford, CA 94305. He is now with ROLM MIL-SPEC Computers, San Jose, CA 95134.

E. J. McCluskey is with the Center for Reliable Computing, Computing Systems Laboratory, Stanford University, Stanford, CA 94305.

IEEE Log Number 8716240.

0018-9340/88/0200-0160$01.00 © 1988 IEEE

TABLE I
ERROR DETECTION MECHANISMS [52]

IPF:	Invalid Program Flow	Improper sequence of instructions
IOA:	Incorrect Opcode Address	Fetching instruction from non-instruction address
UNM:	Unused Memory	Memory access to existent but unused memory
IRA:	Invalid Read Address	Read access (for data) to instruction area, or non-existent memory
IOC:	Invalid Opcode	Illegal instruction
IWA:	Invalid Write Address	Attempt to write into non-alterable memory
NEM:	Non-Existent Memory	Access to a location with no memory

an example of design where a specialized processor is used for both I/O processing and error detection. Also, many systems are using special console processors for off-line diagnostics [5], [34]. The popularity of these techniques further supports the idea of using a watchdog coprocessor for concurrent error detection.

The paper is organized as follows: 1) Section II describes the experimental results about the error coverage of different system-level error detection techniques, 2) two kinds of control flow checking techniques are discussed in Section III, which also deals with detecting errors in hardwired and microprogrammed control units, 3) Section IV describes memory access checking, and 4) Section V discusses the use of a watchdog processor for performing reasonableness checks on the output of a main processor by executing assertions.

II. Experimental Evaluation of Different Error Detection Mechanisms

A major decision in the design of a watchdog processor is the choice of a system characteristic to monitor. The chosen characteristic must satisfy the following requirements: 1) it should not make the watchdog complex, 2) it should provide good error coverage, 3) it should not require major changes to be made in the design of the checked processor, and 4) it should not result in high overhead to the monitored system. Many different mechanisms that can be used to detect errors at the system level were studied experimentally in [33] and [52].

In [52] the seven mechanisms shown in Table I were studied by means of simulation. Faults (temporary and permanent) were injected on the external pins of the Z-80 microprocessor and the subsequent system response was recorded. For each mechanism, the relevant features of the response were extracted and compared to the response of the system with no faults. Any difference in the faulty and fault-free run was viewed as a successful detection of the fault by that particular mechanism. A total of 539 faults were injected. Of these 73 percent were detected by at least one of the mechanisms within 250 μs (250 μs being the maximum time for which the state of the system was observed). Fig. 2 describes the performance of

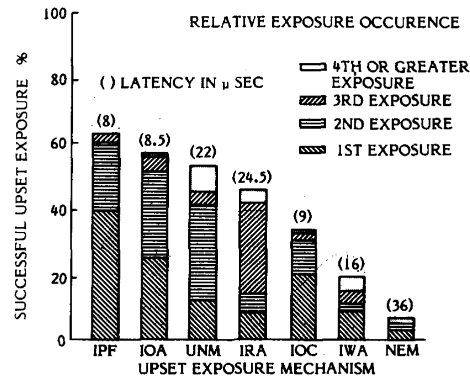


Fig. 2. Comparison of different mechanisms [52].

each individual mechanism. The highest point of the bar for each mechanism gives the total coverage, with different shaded portions indicating the time of detection relative to the other mechanisms. The first area indicates the percentage of time that the particular mechanism was the first to expose the fault, the second area indicates the percentage it was second, and so forth. The average latency from fault to detection is listed on the top of each bar. Fig. 3 shows the performance of each mechanism depending on where the fault was injected. It is clear from Figs. 2 and 3 that invalid program flow (IPF) was the best in detecting faults, both in terms of coverage (63 percent) and latency (8 μs). Moreover, most of the mechanisms did a good job of detection if the fault was on the address lines, but performed poorly if the injected fault was on the data line.

Similar results were obtained in an independent experimental study [33]. In this particular case, a software model of the processor (Texas Instrument SBR 9900) was used. This provided the opportunity to insert faults inside the processor and not just on the external pins.

III. Control Flow Checking

It is obvious from the results presented in Section II that control flow and memory access checking can be used very effectively for detecting errors at the system level. Control flow checking is discussed in this section and memory access checking is described in Section IV.

Any program can be represented graphically, with nodes representing some program unit and arcs representing the flow of control. A *node* can be a single statement, a block of statements with no jumps allowed from or into the block (branch-free interval), a loop-free interval, or a single procedure. All schemes of control flow checking are based on associating a signature or token with a node (called *node signature*). The watchdog is provided with the signatures and the relationship among signatures. This becomes the *watchdog program*. During execution of a program, the watchdog monitors the control flow of the program, computes the node signature concurrently (or accepts the signature explicitly transmitted by the main processor), and compares it to the signature provided earlier. Any discrepancy in the two signatures is taken as an indication of error. Many schemes for

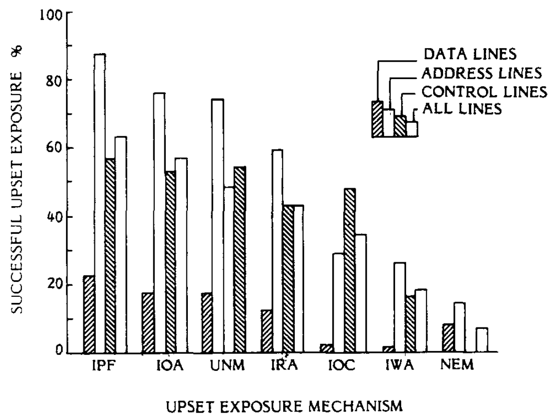


Fig. 3. Error coverage depending on fault type [52].

checking control flow have been proposed [14], [27], [36], [44], [59], [67]. These schemes differ in their definition of a node and the node signature and in their derivation and representation of the watchdog program. In some schemes, the node is a high-level language construct [36] and in others it is a branch-free interval consisting of Assembly language instructions [44]. The signatures associated with the nodes can be either assigned arbitrarily (for example, using prime numbers) or they can be derived from the instructions in the node. Checking techniques in which the signatures are associated arbitrarily with the nodes will be called *assigned-signature control flow checking* and the techniques in which the signatures are derived from the nodes will be called *derived-signature control flow checking*. In both cases, the watchdog program is homomorphic to the control flow structure of the main program. However, the method of computing the node signature has an important bearing on the error detection capability of different schemes.

On the basis of their error detection capability, the schemes for checking control flow can be divided into three classes: 1) the schemes that check that the nodes are executed in an allowed sequence, 2) the schemes that verify the sequencing of the contents of a node, and 3) the schemes that do both. The schemes that use assigned-signature checking fall into the first category and the schemes that use derived-signature checking fall into the last two categories. As an example, consider the program and its graph shown in Fig. 4(a). Fig. 4(b) shows the watchdog program that only checks that the nodes are executed in an allowed sequence. (The allowed sequences are [$V1$ $V2$ $V4$] and [$V1$ $V3$ $V4$].) The node signatures are assigned arbitrarily in this case and explicitly transmitted to the watchdog. Most errors in the execution of the node itself, for example, change of an add instruction into a subtract instruction, are not detected. One advantage of such a scheme is that the watchdog program can be generated directly from a program written in a high-level language. Another advantage is that the two processors can operate asynchronously without making the watchdog complex. [36] is an example of such a technique.

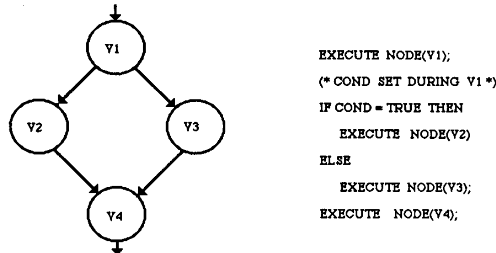
Fig. 4(c) shows the watchdog program that only checks the sequencing of the contents of a node (with the node signature computed from the contents of the node). The watchdog has to be informed which node is being executed by the main processor. An example of such a scheme is [14]. Fig. 4(d) shows an example of a watchdog program whose execution by the watchdog results not only in checking the node transitions but it also detects errors in sequencing of the contents of the node itself. Signatures in both these cases [Fig. 4(c) and (d)] are not transmitted explicitly by the main processor but are computed concurrently by the watchdog. [15], [44], and [59] are examples of the technique shown in Fig. 4(d).

It should be pointed out that these schemes for control flow checking only verify that the nodes are executed in an allowed sequence and not necessarily the correct sequence (a sequence may be allowed but incorrect). Table II illustrates the main properties of different control flow checking schemes. These schemes are discussed later in Sections III-A, III-B, and III-C.

A. Assigned-Signature Control Flow Checking

Examples of such techniques are [27], [36], and [67]. In [36] a technique called *structural integrity checking* (SIC), which is based on recognizing high-level control flow structures in computer programs, labeling these structures with signatures or labels, and checking the integrity of these structures at run time using a watchdog processor, is presented. The concept of SIC is based on the theory of formal languages and automata. It uses syntax-driven methods for encoding program structures and automatically generating two programs, one for the main processor and the other for the watchdog processor, from the original program. The schemes described in [27] and [67] also assign labels to program constructs. However, the method used for arriving at the labels in SIC results in much simpler implementation as compared to these two techniques.

A typical software configuration is shown in Fig. 5, using the language Pascal as an example. The SIC preprocessor reads in the Pascal source program and analyzes the program for four constructs: concatenation, selection, repetition, and abstraction (procedures). Table III shows examples of some of these constructs. Labels (signatures) are attached to these constructs and a program known as a Labeled Structured Program is generated for the main processor. The second output of the preprocessor is the Structural Reference Program for the watchdog. The structure of this program mimics the structure of the source program. In place of computations in the source program, the structural reference program contains statements to receive and check labels from the main processor. As the structural reference program does not contain code to execute the actions of the program being checked, the computational requirements for the watchdog are less than those for the main processor. As the main processor executes the labeled structured program, it transmits each label it encounters. The watchdog receives these labels and compares them to the labels generated within the watchdog by the structural reference program. In case of a mismatch, an error is signaled to the system component that manages error detection and recovery. The operations of SIC can be abstractly modeled as the activities of two automata. One automaton is the execution of the labeled structured program




```
EXECUTE NODE(V1);
(* COND SET DURING V1 *)
IF COND = TRUE THEN
   EXECUTE NODE(V2)
ELSE
   EXECUTE NODE(V3);
EXECUTE NODE(V4);
```

(a)

```
ACCEPT SIG(V1);
EITHER
   ACCEPT SIG(V2)
OR
   ACCEPT SIG(V3);
ACCEPT SIG(V4);
```

(b)

```
CASE NODE OF
   V1: CHECK SIG(V1);
   V2: CHECK SIG(V2);
   V3: CHECK SIG(V3);
   V4: CHECK SIG(V4);
END;
```

(c)

```
ACCEPT SIG(V1), CHECK SIG(V1);
EITHER
   ACCEPT SIG(V2), CHECK SIG(V2)
OR
   ACCEPT SIG(V3), CHECK SIG(V3);
ACCEPT SIG(V4), CHECK SIG(V4);
```

(d)

Fig. 4. Different schemes for control flow checking. (a) Main program. Watchdog programs for (b) checking node transitions, (c) checking the node, and (d) checking both.

TABLE II
PROPERTIES OF DIFFERENT CONTROL FLOW CHECKING SCHEMES

SCHEME	NODE	SIGNATURE	CHECKING RESOLUTION	INTRA-NODE CHECKING	INTER-NODE CHECKING	SYNCH-RONIZATION	MEM. OVERHEAD	TIME OVERHEAD
SIC	HIGH LEVEL LANG. CONSTRUCT.	ASSIGNED	VARIABLE	NO	YES	ASYNC	H	H
BASIC PSA	BRANCH-FREE INTERVAL	DERIVED	ONE NODE	YES	YES	SYNC	H	H
GEN PSA	BRANCH-FREE INTERVAL	DERIVED	SEQUENCE OF NODES	YES	YES	SYNC	L	H
BAH	BRANCH-FREE INTERVAL	DERIVED	SEQUENCE OF NODES	YES	YES	SYNC	L	H
8085 CHECKER	MACRO-INSTRUCTION	DERIVED	ONE NODE	YES	NO	SYNC	H	L
CERBERUS-16	BRANCH-FREE INTERVAL	DERIVED	SEQUENCE OF NODES	YES	YES	SYNC	L	L
RMP	BRANCH-FREE INTERVAL	DERIVED	SEQUENCE OF NODES	YES	YES	ASYNC	L	L

SIC: STRUCTURAL INTEGRITY CHECKING
PSA: PATH SIGNATURE ANALYSIS
BAH: BRANCH ADDRESS HASHING
RMP: ROVING MONITORING PROCESSOR
ASYNC: ASYNCHRONOUS
SYNC: SYNCHRONOUS
H: HIGH
L: LOW

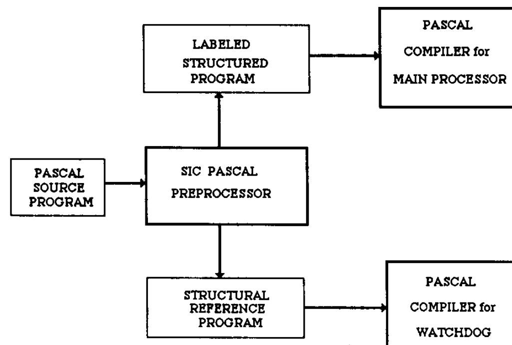


Fig. 5. Typical software configuration (SIC).

TABLE III
SOME CONTROL FLOW CONSTRUCTS (SIC)

SOURCE	MAIN	WATCHDOG
	CONCATENATION	
begin	begin	begin
	send (x);	if token < > x then error;
a;	a;	< check a >
	send (y);	if token < > y then error;
b;	b;	< check b >
end;	end;	end;
	SELECTION	
if a then	if a then	
	begin	
	send (-x);	if token = -x then
b	b	< check b >
	end	
else	else	else
	begin	
	send (-y);	
c;	c;	< check c >
	end;	
Token :	Same as Label or Signature.	
< Check Declaration >:	Apply SIC recursively to abstractions in declarations.	

TABLE IV
STRUCTURAL INTEGRITY CHECKING. (a) SOURCE PROGRAM. (b) MAIN PROCESSOR PROGRAM. (c) WATCHDOG PROGRAM.

```
.
.
read(input, number);
repeat
   if number < 0 then begin
      negsum  := negsum + number;
      number  := number * -1;
   end;
   sum := sum + number;
   read(input, number);
until  (number = 0) or (eoln(input));
.
.
```

(a)

```
send(50);
read(input, number);
send(187);
begin

  repeat
    send(-82);
    (* -82 means loop executed *)
    if number < 0 then begin
       send (-12);
       negsum := negsum + number;
       send(28);
       number := number * -1;
    end else send(-13);
    send(155);
    sum := sum + number;
    send(48);
    read(input, number);
  until (number = 0) or (eoln(input));
  send (-83);  (* terminate loop *)
end;
```

(b)

```
if token <> 50 then error;

if token <> 187 then error;
begin (* check repeat loop *)
  if token <> -82 then error;
  repeat
  (* loop terminated when
     token not equal to -82 *)
    if token = -12 then begin
    (* -12 means 'if' executed *)

       if token <> 28 then error;

    end;
    if token <> 155 then error;

    if token <> 48 then error;

  until token <> -82;

end;
```

(c)

in the main processor and the second automaton is the execution of the structural reference program in the watchdog processor. The strings of labels produced by the execution of a labeled structured program can be described by a context-free grammar. The structural reference program is so constructed that the watchdog acting as a push-down automaton accepts exactly those strings described by the context-free grammar and rejects any other strings. As an example consider a part of a program shown in Table IV(a). From this program two programs shown in Table IV(b) and (c) are generated. The first one is executed by the main processor and the second one is executed by the watchdog processor.

B. Derived-Signature Control Flow Checking

An example of derived-signature control flow checking is *path signature analysis* (PSA) [44] (only programs that do not modify themselves are considered). First, the basic scheme (called **Basic PSA**) will be discussed and then the modified scheme (called **Generalized PSA**) will be described.

In the *Basic PSA Scheme*, a deterministic signature is derived for each node of the graph (where each node is a branch-free interval); the *signature of a node* represents some characteristic of that node. For example, the result of mod-2 addition, a checksum, or linear feedback shift register (LFSR) signature of the instruction words in a node can be defined as the signature of that node. The signature is inserted at the beginning of each node. Two tag bits are used to differentiate signatures from the rest of instructions in the node. The watchdog monitors the data bus and captures the signatures as they appear, using tag bits to differentiate them from the rest of instructions. The main processor executes a NOP instruction whenever a signature is fetched. For the rest of the instructions in the node, the watchdog computes the signature concurrently. At the end of the node, the watchdog compares the computed signature to the actual signature. Fig. 6 shows a typical structure of a node. Fig. 7(a) shows a sample program for the 68000 microprocessor with its corresponding graph. Fig. 7(b) shows the same program with signatures added. The signatures were computed using the Exclusive-OR operation. The scheme described in [59] is very similar to the above mentioned technique. The main difference is that in [59] the signatures are added to the nodes such that the signature computed on-line at the end of node is all-one. This simplifies the checking circuitry because instead of a comparator, a simple gate is enough to generate an error signal.

1) Improvements: The insertion of explicit signatures in the instruction stream increases the memory overhead and reduces the performance. The schemes to be described next compute signature in such a way that each signature checks a greater number of instructions (compared to the Basic PSA), thereby reducing the total number of explicit signatures that need to be stored.

The first such scheme *Generalized PSA*, is described in [44]. In this scheme, the signatures are computed for sequences of nodes, that is, paths rather than single nodes. The program graph is broken into path sets and one signature is derived for each path set. Each path set contains one or more paths, with each path in the path set starting from the same node. As there is only one signature for each path set, each path in the path set must result in the same signature. In order to make this possible, sometimes pseudosignatures, called

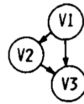
01	SIGNATURE
00	FIRST INSTRUCTION
00	SECOND INSTRUCTION
00	
00	
11	LAST INSTRUCTION

Fig. 6. Structure of a typical node (PSA).

```
001000         207C00006002   L1:  MOVEA.L    #DATA1,AO
001006         4201                CLR.B      D1
001008         10386000            MOVE.B     DATA2,DO
00100C   V1    0C000009            CMP.B      #9,DO
001010         6206                BHI.S      L2

001012   V2    4880                EXT.W      DO
001014         12300000            MOVE.B     0 (AO,DO),D1

001018   V3    11C16001       L2:  MOVE.B     D1,DATA3
00101C         4E75                RTS
```



(a)

```
001000   01    1C46           L1:  (NOP)
001002   00    207C00006002        MOVEA.L    #DATA1,AO
001008   00    4201                CLR.B      D1
00100A   00    10386000            MOVE.B     DATA2,DO
00100E   00    0C000009            CMP.B      #9,DO
001012   11    6208                BHI.S      L2

001014   01    5AB0                (NOP)
001016   00    4880                EXT.W      DO
001018   00    1230                MOVE.B     0 (AO,DO),01
00101A   11    0000

00101C   01    3FB5           L2:  (NOP)
00101E   00    11C16001            MOVE.B     D1,DATA3
001022   11    4E75                RTS
```

(b)

Fig. 7. Basic path signature analysis. (a) A program for the 68000 processor. (b) The same program with signatures added.

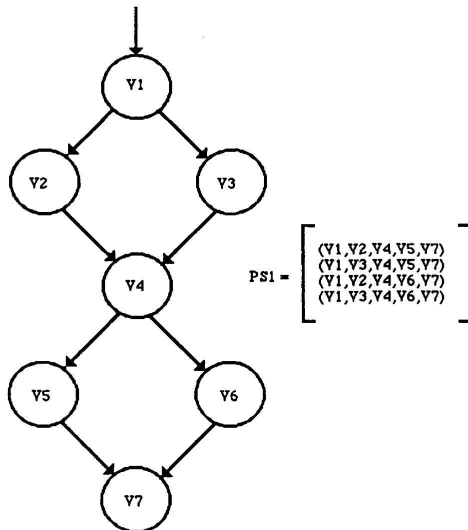


Fig. 8. A program graph to illustrate Generalized PSA.

justifying signatures, are added to the paths in the path set (assuming that the inverse of the signature exists), so that no matter what path is executed in the path set, the same signature results.

As an example, consider the program graph shown in Fig. 8. It has one path set that consists of four different paths. In order to make all these paths equisignature, the signatures of the nodes $v3$ and $v6$ are justified by assigning two justifying signatures to these nodes. (It is assumed that the Exclusive-OR operation is being used for computing signatures.)

$$h'(v3) = h(v3) \oplus h(v2) \quad \text{(justifying signature at node } v3\text{)}$$

$$h'(v6) = h(v6) \oplus h(v5) \quad \text{(justifying signature at node } v6\text{)}.$$

After this modification, the signatures of the paths would be

$$P1\colon H1 = h(v1) \oplus h(v2) \oplus h(v4) \oplus h(v5) \oplus h(v7)$$

$$P2\colon H2 = h(v1) \oplus h(v3) \oplus h'(v3) \oplus h(v4) \oplus h(v5) \oplus h(v7) = H1$$

$$P3\colon H3 = h(v1) \oplus h(v2) \oplus h(v4) \oplus h(v6) \oplus h'(v6) \oplus h(v7) = H1$$

$$P4\colon H4 = h(v1) \oplus h(v3) \oplus h'(v3) \oplus h(v4) \oplus h(v6) \oplus h'(v6) \oplus h(v7) = H1$$

where $h(vi)$ is the signature of a single node.

The common signature of these paths ($H1$) is stored at the node $v1$. The total number of signatures for this example is three which is less than that in the Basic Scheme (seven). Fig. 9 shows the same program as in Fig. 7(a) with signatures added based on the Generalized method. The total number of signatures in this case is two, which is less than that in the basic method.

The second scheme, *Branch Address Hashing (BAH)* described in [55] improves the Basic PSA by combining some of the signatures with branch target addresses, thereby reducing the overhead for storing signatures by 50 percent. There are some notable differences between this scheme and the Basic PSA. The program graph is partitioned differently and the signatures are inserted at the end of nodes rather than at the beginning of nodes. Instead of embedding one signature immediately preceding every branch instruction, the branch address of the branch instruction is modified at assembly time such that during execution, the changed branch address combined with the concurrently computed signature gives the correct branch address. An explicit signature still needs to be embedded preceding every branch-in point (labeled instruction). In the case of an error, the concurrently computed signature and, hence, the computed branch address will be incorrect, resulting in the execution of a branch to an erroneous destination. The error will be detected when the next embedded signature is encountered. This is shown in Fig. 10. For the programs written in the Assembly language of the MC 68000 microprocessor, the signatures are embedded by using a pseudo-branch instruction (psbr). A psbr instruction is an unconditional branch to the location PC + 2. A 16-bit embedded signature as shown in Fig. 11, is stored immediately following the psbr instruction. The psbr instruction signals the monitoring hardware of the embedded signature. Use is made of the one-word prefetch of the MC 68000 microprocessor during the decode of psbr instruction. As the psbr instruction is being decoded, the embedded signature is

```
001000  01  23F3           L1:  (NOP)      ;Path signature
001002  00  107C00006002        MOVEA.L    #DATA1,AO
001008  00  4201                CLR.B      D1
00100A  00  10386000            MOVE.B     DATA2,DO
00100E  00  0C000009            CMP.B      #9,DO
001012  00  6208                BHI.S      L2

001014  10  5AB0                (NOP)      ;Justifying
                                           signature
001016  00  4880                EXT.W      DO
001018  00  1230                MOVE.B     0 (AO,DO),D1
00101A  00  0000

00101C  00  11C16001       L2:  MOVE.B     D1,DATA3
001020  11  4E76                RTS        ;Terminal node
```

Fig. 9. The same program as in Fig. 7(a) with signatures added using the generalized method.

	CONCURRENTLY COMPUTED SIGNATURE		SOURCE CODE	
	0000		jsbr	input
	3D08		movl	d0, tie
	F7A5		cmpl	#1, d0
a:	B863		jlt	.L10000
	B0C5		cmpl	#384, d0
	51B5		jle	.L81
b:	0000		psbr	.L10000
	7D22	.L10000:	movl	#13, sp@-

a: Concurrently computed signature is combined with the address given in the source code to compute the actual address.

b: Pseudo-Branch for inserting an explicit signature.

Fig. 10. Branch address hashing [55].

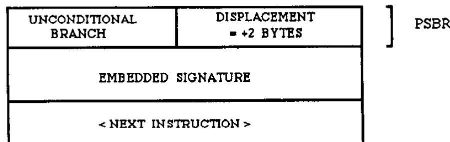


Fig. 11. An embedded signature (BAH).

prefetched, detected, and compared to the running (concurrently computed) signature. In actual implementation, the inverse of the signature is stored so that the running signature is all-zeros, thereby making the comparison easier.

In order to understand how branch address hashing works, consider Fig. 10. Starting from the first instruction, the signature is computed concurrently as each instruction is fetched. At point marked "*a*" in the figure, in the Basic PSA, concurrently computed signature would have been compared to the signature inserted explicitly in the beginning of the node. However, in this case, comparison is only made when a pseudobranch instruction is encountered, which is before a branch-in point (labeled instruction). In case the jump is not taken at point "*a*," the signature is continued to be computed as instructions are fetched. If the jump is taken, then the jump address is computed by combining the signature computed up to that point with the address given in the jump instruction. This scheme has been actually implemented and experiments are being conducted to determine the error coverage.

2) Independent Watchdog Processors: All the techniques for derived-signature control flow checking presented so far require that the signatures be inserted in the instruction stream. This increases the execution time of a program. However, by storing signatures in the local memory of a watchdog, this overhead can be removed. Two such schemes will be discussed. The first is a synchronous scheme for checking uniprocessor systems [46] and the second is an asynchronous scheme for monitoring multiprocessor systems [15], [65].

The watchdog processor which uses the first scheme is called **Cerberus-16**. The information about the program graph and path signatures is moved into (the environment of) the watchdog processor. The program executed by the watchdog has the same control flow structure as the program being executed in the main processor; that is, the graphs of both programs are identical. A node in the graph (G) of the program (P) being executed on the main processor represents a branch-free interval, whereas the node in the program (P') being executed in the watchdog consists of a single instruction that contains some information (such as signature or the size of node) about the corresponding node in graph G.

The architecture and organization of Cerberus-16 differs from most existing processors in two respects: 1) most instructions of this processor are control transfer instructions and 2) there is no ALU in this processor. The instructions for Cerberus-16 can be divided into two categories. The instructions in the first category are used for representing control flow in the program graphs. The general format of such instructions is

$$\text{OP } Z, [L], [D]$$

where Z, L, and D represent the node size, the next node address (label), and the node signature, respectively. L and D fields are optional. The instructions in the second category are used for initialization and communication with the main processor. They have the following format.

$$\text{OP or OP } D.$$

Any program graph can be represented by a set of watchdog instructions. Depending on the type of a node, a suitable instruction is used to represent that node. As that node is executed by the main processor, the instruction corresponding to that node is executed by the watchdog. After z instructions have been executed by the main processor, the watchdog then expects a signal from the branch detecting circuitry to indicate whether a branch was taken by the main processor or not. If it detects a branch, then the watchdog also executes a branch using the L field to calculate the target address of the next instruction. The next watchdog instruction corresponds to the node which should be executed by the main processor after the branch. In case no branch is detected, the watchdog executes the next instruction in sequence. As each instruction is executed by the watchdog, the value in the D field (signature) is fed to the data compression unit whose output is saved in a register. As an example of how this architecture can be used for control flow checking, consider a program, with graph as shown in Fig. 12(a), being executed on the main processor. The corresponding program executed by the watchdog is

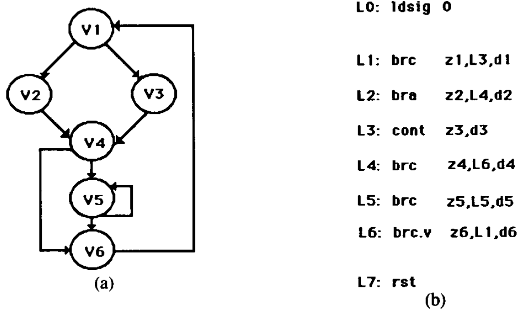


Fig. 12. Control flow checking with Cerberus-16. (a) The graph of the program executed by the main processor. (b) The corresponding program executed by Cerberus-16.

shown in Fig. 12(b). There are exactly six instructions corresponding to the six nodes of the main program. The Zi and di $(i = 1 \cdots 6)$ parameters in each instruction represent the size and the signature of the corresponding node in the main program, respectively. The instruction LDSIG initializes the signature register (S) to zero. At any given time, the register S contains the accumulated signature of the executed path. For example, if the main processor executes the node sequence $(v1, v2, v4, v5, v5, v5, v6)$ and the compression function selected is XOR, then the watchdog executes the instructions at addresses $(L1, L2, L4, L5, L5, L5, L6)$ and the accumulated signature after the execution of this path would be

$$h = d1 \oplus d2 \oplus d4 \oplus d5 \oplus d5 \oplus d5 \oplus d6.$$

The instruction corresponding to the node $v6$ is BRC.V (branch conditional and verify) instruction. This instruction signals the end of the path. The signature computed by the watchdog at this point is compared to the actual signature of the path.

The same watchdog can be used for collecting statistical data about a system by changing the data [D] field of the watchdog instruction. An example of such data is the frequency of different instructions. It is interesting to note that the earliest design of a processor for monitoring another processor was for the purpose of collecting performance data [16]. However, the watchdog is much simpler compared to the earlier design.

The above mentioned design of Cerberus-16 requires the watchdog to be tightly synchronized with the main processor. An asynchronous version of the above scheme is discussed in [15]. It is based on storing the control flow information of the program being executed on the monitored processor as the signature graph in the local memory of the monitoring processor. The graph is stored in the form of a linked list data structure. As each node is executed in the main processor, its signature is transmitted to the monitoring processor through a signature queue. The monitoring processor uses the signature graph to check that the next signature it receives belongs to a set of nodes which can be reached from the current node. This scheme can also be used for monitoring more than one processor if a different signature queue is used for each processor and signature graphs of all the programs being executed on different processors are stored in the local memory of the monitoring processor.

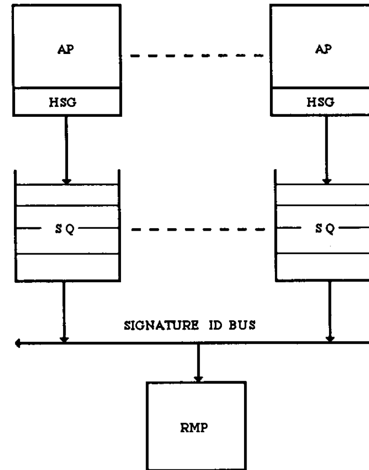


Fig. 13. Roving monitoring system [65].

[65] discusses the actual implementation of the processor (called **Roving Monitoring Processor** or **RMP**) described in [15]. Fig. 13 shows the general configuration of Roving Monitoring System. There is one hardware signature generator (HSG) for each application processor (AP). Each HSG generates a stream of signatures for its corresponding AP and stores it in the signature queue (SQ). The RMP samples the SQ's according to some policy and checks that the signature stream generated by the AP corresponds with its signature graph, already stored in the local memory of RMP. Only eight instruction types, shown in Table V, are sufficient to represent the program graph. Signature words containing signature and processor ID arrive at the RMP from SQ via the common signature bus. The processor ID field is used by the RMP to access the appropriate program counter from the program counter register file. The prototype for RMP has been designed and implemented. It uses 90 MSI/SSI packages (excluding memory) and can monitor 16 processors. A series of fault insertion experiments, designed to determine the error coverage and error latency are being conducted.

3) Overhead: The memory overhead for storing signatures can be as high as 20 percent if the Basic PSA scheme described in Section III-B is used. However, if the improvements discussed in Section III-B-1 (Generalized PSA and branch address hashing) are used, then the overhead can be reduced to 10 percent. The hardware required to implement the watchdog itself can be as little as 20 percent of the complexity of the MC 68000 microprocessor [40].

4) Error Coverage: Derived-signature control flow checking techniques discussed in Section III-B can detect both memory and control flow errors. Examples of control flow errors are incorrect sequence of instructions, branch to wrong address, branch from a wrong address, etc. These control flow errors can be the result of failures in the instruction register, the program counter, the address register, decoding circuitry, memory addressing circuitry, etc. It was shown in [40] that control flow errors can be modeled as memory errors for the

TABLE V
INSTRUCTIONS FOR RMP [65]

CMPS S, SRP	Compare signature with S, branch relative to SRP if match
CMPSE S,SRP	Compare signature with S, branch relative to SRP if match. Error if no match.
CMPIE S	Compare signature with S, go to the next instruction if match. Error if no match.
HASH	Short relative branch, based on lower signature bits.
PUSH LPTR	Push current address to stack, jump long to location LPTR
POP	Pop current address from stack.
POPSE	Compare signature and pop the next address from stack. Error if no match.
LJMP LPTR	Unconditional jump to location LPTR

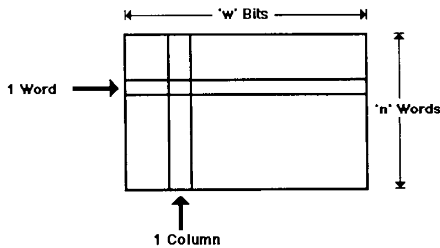


Fig. 14. A single node.

purpose of calculating error coverage. The problem is thus transformed to that of calculating coverage for errors in a block of data. The error coverage when parallel linear feedback shift register (PLFSR or MISR) is used to compute the signature is discussed in [40]. The results described in [40] are based on the properties of linear feedback shift registers [8], [20], [21] and on the observation that the number of instructions in a node is small [24], [56]. Suppose that the node has "n" words, each "w" bits wide (that is, each memory word is "w" bits wide), as shown in Fig. 14. Using a PLFSR to compute the signature of the node, it was shown in [40] that all single word errors are detected and most single column errors are detected. If $n = 10$ and $w = 16$, then the percentage of errors not detected due to aliasing (when the signature is same even in the presence of error), assuming all errors are independent and equally likely, is 0.0015 percent. Although the above results are for Basic PSA, they apply equally well to the other schemes (Generalized PSA, BAH). These results are true not only for memory errors but also for control flow errors. This is because control flow errors can be modeled as memory errors as described earlier.

In order to study the error coverage in more detail suppose that "w," the width of a node, is 32 bits. Further assume that the memory consists of 32 chips each with 1 bit/chip. If at most one memory module fails at one time, then at most one column (of the node) can be in error. If the size of a node, "n" (number of memory words in a node), is less than the width of the node, then all the errors will be detected [40]. If the size is greater than "w," then the ratio of undetected errors to the total possible errors is $(2^{n-w} - 1)/(2^n - 1)$. For large "n," this can be approximated by 2^{-w}. Since at most one column can be in error, fault masking cannot occur. As an example, assume that the distribution of node sizes is as shown in Table VI. This distribution is based on the number of instructions executed between successful branches and is described in [24]. For each node size, the percentage of errors detected is shown in column "e" in Table VI. It can be seen that more than 99.9 percent of the errors are detected. If there is more than 1 bit/chip, then single chip failure can result in multiple column error. However, even then the error coverage is greater than 90 percent as shown in [40].

Most of the memory errors will appear as column errors. Stuck faults on the data lines can also be modeled as column errors. Control flow errors can be represented as word errors. Such errors can be divided into two classes: one in which a node is mapped into an erroneous node of the same size and the other in which the size of erroneous node is different. If the checking scheme used by Cerberus-16 (Section III-B-2) is used, then any change in the node size is immediately detected. This is because the node size of each node is stored with the signature of that node. So only the errors which change a node into a different node of the same size can go undetected. For detection purposes, such errors can be modeled as some or all bits in a data block have been changed. As described earlier, for a node size of 10 words each 16 bits wide, the percentage of errors not detected due to aliasing is only 0.0015 percent. As an example of an error that changes the node size, consider a node of size "n." Suppose that as a result of a control flow error, words 0 to $b - 1$ of the node are not executed and the execution starts from the bth word. Then the signature computed after executing the words b to $n - 1$ in the node will be the same as if all the words in the node (0 to $n - 1$) were executed with the words 0 to $b - 1$ replaced by all zero words. This means that jumping in the middle of node and starting the execution from the bth word is equivalent to the first b words being in error. This is shown in Fig. 15. If only the first word is missed, then this error (equivalent to a single word error) is guaranteed to be detected (assuming that the missed word was not all zero itself) [40]. Many other errors, like illegal opcode, illegal memory read/write, addition of a branch to a node, omission of a branch from a node, etc., can also be modeled as word errors. Hence, the results derived for memory errors also hold in the case of control flow errors.

The results described above are based on the assumption that all errors are equally likely. Although this assumption has been challenged for off-line (explicit) testing, there is as yet no reason to doubt its validity for concurrent testing. However, the best way to confirm the above results is by means of experimental studies. Such studies are being conducted at Carnegie-Mellon University [53] in which the schemes described in [55] and [65] are being evaluated experimentally.

TABLE VI
ERROR COVERAGE FOR 32-BIT WIDE MEMORY WITH 1 BIT/CIP

Node Size n	Freq. f	% Error Detected e	f*e
1	.0192	100	1.921%
2-3	.0864	100	8.645%
4-7	.2381	100	23.81%
8-15	.3522	100	35.22%
16-31	.1499	100	14.99%
>31	.1539	99.9	15.38%
OVERALL COVERAGE		∑ f*e	99.9%

n = node size

f= frequency

e = % error detected

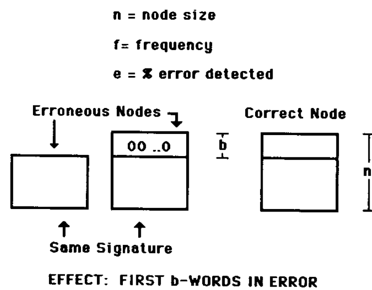


Fig. 15. An example of a control flow error.

C. Checking the Control Section of a Processor

Some of the techniques used for control flow checking can also be used for checking the control section of a processor. A design for checking the sequence of control signals corresponding to each opcode, for hardwired control units, is discussed in [14]. It is based on assigning a unique signature to each opcode (some opcodes like branches require more than one signature). The signature is computed by compacting the sequence of control signals using a PLFSR. These signatures are stored in a ROM. When a particular opcode is executed by the processor, a signature is computed concurrently by using a PLFSR. The same opcode is used to fetch the stored signature from the ROM. A checker for the 8085 microprocessor was implemented using SSI/MSI chips. Seven control signals (visible at the pins) were monitored: ALE, S0, S1, IO/M, RD, WR, and INTA. The implementation required approximately 535 gates which is about 11 percent of the complexity of 8085.

The techniques used for checking the control flow of programs written in Assembly language, and discussed in Section III-B, can also be used for checking microprogrammed control units. However, this can result in high memory overhead and performance degradation. This is because the node size can be very small, especially if horizontal microinstructions are used. To overcome this difficulty, the schemes have been modified for checking microprogrammed control units. All the modifications are aimed at extending the microinstructions and inserting the signatures in parallel to microinstructions, thereby improving the performance. More details can be found in [40], [45], [59], and [26]. The schemes described in [40], [45], and [59] use derived signature checking, whereas [26] is based on assigned signature checking.

1) Comparison to the ESS-3A Processor: The ESS-3A (Electronic Switching Systems) is a microprogrammed processor [11], [61] and is designed to control the No. 3 ESS [25], a small electronic switching system, which can handle from 500 to 5000 telephone lines. ESS-3A was chosen for comparison for two reasons. First it has a very high availability requirement (down time of a few minutes per year), and second, it does not use duplication at the system level for error detection. Instead, it relies on the error detection circuitry built into the processor to detect faults. The techniques used in the control section for fault detection are as follows: 1) each of the two 8-bit fields of a microinstruction is encoded in the 4-out-8 code, 2) two parity bits are used to check the sequencing of microinstructions, and 3) the instruction decoder is checked with a self-checking checker. The overhead of checking circuitry in the control section is about 30 percent. This does not include the memory overhead due to the 4-out-8 encoding and the two extra parity bits. The microprogram ROM contains 1024 words (extendable to 4096), each 32 bits wide. The memory overhead is about 14.28 percent. The redundancy in the control section provides protection against the following errors: 1) all unidirectional errors in the microprogram ROM, 2) some sequencing and addressing errors, and 3) decoding errors.

It was shown in [40], using the same results as described in Section III-B-4, that the watchdog processor provides better error coverage for comparable overhead cost. More than 90 percent of the memory errors are detected as was shown previously. All single word errors in the node are detected and very high error coverage is obtained for column errors (because of the small node size). This technique also provides much better protection against sequencing and addressing errors. If the signature is formed at the output of the decoder, instead of at the output of the microprogram ROM, then the decoding errors can also be detected.

The memory overhead can be as low as 7 percent (as compared to 14 percent in ESS-3A). The nonmemory overhead for the watchdog is comparable to the hardware dedicated to checking in ESS-3A.

IV. Memory Access Checking

The watchdog can also detect errors in the main processor by monitoring the memory access behavior [43]. In fact, the watchdog discussed in Section III-B-2 (Cerberus-16) can be modified easily to check the memory access along with the control flow [46], thereby improving the error coverage. The design of a watchdog for memory access checking as described in [43] is based on the capability-based addressing [17] scheme in which each accessing object has to provide a capability (a protection key) for accessing another object. Each capability uniquely specifies an object. In the scheme described in [43], and shown in Fig. 16, explicit access to the physical memory segments is checked by an external watchdog processor. Basically, the set of all active objects (code or data) at any given time is represented by a directed graph (object graph). The object graph is stored in the watchdog processor in a table called segment access table (SAT). The SAT has one row for each code object; the entries in each row

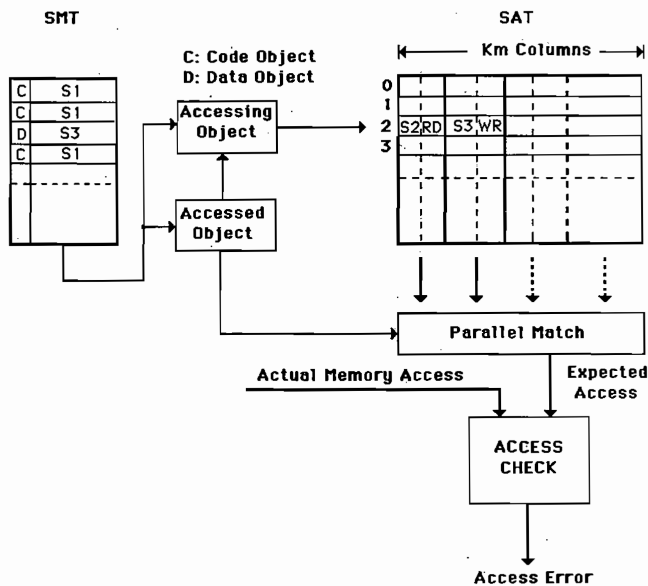


Fig. 16. Memory access checking using a watchdog processor.

are the capabilities (access rights) given to that object. The watchdog processor has another table called segment map table (SMT) that is used for mapping the physical page frame numbers (derived from the memory address bus) onto segment numbers representing objects. The SMT has one entry corresponding to each page frame in the processor's main memory. For each memory access, two segment numbers, *Si* and *Sj*, are determined from the SMT, one for the accessing object and the other for the accessed object. These segment numbers are used to read the expected access right, SAT(*Si*, *Sj*), from the SAT. The size of the SAT can be reduced significantly if it is assumed that at most *Km* objects can be accessed by any object. This makes it possible to organize the SAT as a parallel hash table that contains *Km* entries in each row. The segment number of the accessing object selects one row of this table. A parallel search is done to match the segment number of the accessed object with a segment number in the selected row and to read the corresponding access right (if a match occurs). This access right is compared to the actual CPU access. An access error is signaled if there is no match in the SAT or the access right is not correct.

V. Reasonableness Checks

All the schemes of system-level error detection, described so far, are aimed at detecting memory access and control flow errors. These schemes do not provide adequate coverage in case of semantic or data manipulation errors. This is also evident from Fig. 3 in Section II. One way to detect data manipulation or semantic errors is to have the watchdog execute assertions about the program being executed in the main processor. Semantic checking requires a notion of correctness. This notion can be provided by assertions. An assertion is an invariant relationship between the variables of a program, written as a logical statement, and inserted at different points in the program. It signifies what the designers believe to be true at the point the assertion is inserted. An example of a program segment with an assertion inserted as a comment is shown in Table VII. This program segment was

TABLE VII
A Program Segment with an Assertion

```
Define Procedure LAT.INNER to be

    begin
    .
    .
    if R.TEST.COMPL
    then begin
        RL8 = RL8.D = DLIMIT (RL8.D + RL11.D + RL11.D.S, 0.258);
        RL11.D.S = RL11.D;
        RL13 = LIMIT (RL7 + RL8, 0.171429) / 0.203333;
        end
    else RL13 = TEST.CMD (RAM.PTR (R.TEST.PTR));
    .

    .
    DELA.CMD = RL13;

COMMENT ASSERT ABS(DELA.CMD) < 0.13;
    .

    .
    end;
```

taken from the flight software package of a commercial wide-bodied aircraft. In case the assertion evaluates to false, an error is assumed to have occurred. Assertions can be written by making use of specifications or some property of the problem or algorithm. They are usually based on the inverse of the problem, the range of variables, or the relationship between variables [37].

Assertions have been used in program verification [18], [22], in program testing [4], [38], [62], for exception handling [12], and for reasonableness checks [29], [49]. The use of executable assertions for detecting hardware and software faults has also been suggested in [2], [3], and [50]. [32] discusses the use of assertions for increasing the safety of systems. Assertions not only serve as a good medium for documentation, but they are also useful for detecting any kind of errors (hardware, software, design) throughout the lifecycle of the system. The effectiveness of assertions in detecting errors has been demonstrated experimentally in [4] and [39]. Many designs have been proposed for executing assertions efficiently. In [30] a design that uses a recursive cache was presented. However, that scheme cannot be used for concurrent execution of assertions. The design presented in [28] requires a complex monitoring system with a sophisticated memory management scheme. The references in [31] provide a more detailed list of different techniques used in executing assertions efficiently. However, all these techniques require complex implementation. The reason is that the objective in these schemes is not only to detect errors by executing assertions but also to recover to the last verified state. Since the emphasis in the watchdog is on error detection, it can be implemented with much less complexity. The main objective in the design of a monitoring system that uses a watchdog processor is to have a simple design for a watchdog processor and also to be able to transfer data from the main processor to the watchdog without excessive overhead. The design of the watchdog processor to execute assertions, discussed in [41], is described below.

A. Special Purpose Watchdog

The time overhead for transferring data to the watchdog can be reduced and the watchdog design simplified by designing the watchdog to suit a particular application. There are many applications in which the flow of data (the sequence in which the data appear on the data bus) is known and invariant. For such applications, the data need not be transferred explicitly to the watchdog [37]. Instead, the code for assertions is stored in the local memory of the watchdog and the instructions which assign values to the variables needed by the watchdog are tagged. The watchdog monitors the data bus and captures the tagged data. It uses the data flow information to recognize the data. Examples of problems that can be solved using this technique are solution of system of equations using Gaussian elimination [19], discrete Fourier transform, eigenvalues, etc.

Special purpose watchdog processors can also be used for some real-time systems. Telephone switching systems [1] and digital flight control systems [7] are examples of two such systems. The software used in these two systems has some special characteristics which can be used to transfer data to the watchdog with very little time overhead. The programs used in these systems are cyclic in nature and they use a large number of global variables. Executable assertions which use these variables can be stored in the local memory of the watchdog and the data transferred by simultaneously writing to both the main memory and the local memory of the watchdog (making use of global variables). Since the programs are cyclic in nature, the watchdog processor uses the classic dual buffer scheme to execute assertions. One buffer is used to capture the data and the second is used to execute assertions on the data captured during the previous cycle.

B. General Purpose Watchdog

The design of a general purpose watchdog to execute assertions concurrently still remains an area of active research. The major problem is in transferring the data from the main processor to the watchdog without complicating the design. There are two models of parallel execution: shared memory (multiprocessor) and message passing (distributed computing). The use of shared memory can result in complex software because of timing and synchronization problems. The other option is to explicitly send data to the watchdog in the form of messages as suggested in [51]. The hardware organization (based on message passing) is as shown in Fig. 17. The main processor writes to the shared buffer and the watchdog reads from it. Both the processors also have their local memories besides the shared buffer. The software structure is as follows.

1) The underlying hardware is transparent to the programmer who writes assertions as they would be written for the program to be executed on a uniprocessor. The code inserted, for checking purposes only, can be separated from the rest of the program by using a special syntax. Table VII shows an example of such a syntax. Another example is that of ANNA (Annotated ADA) [29] which is an extension of language ADA.

2) Once the code inserted for checking is identified, the preprocessor replaces all the code for one assertion with a single write statement

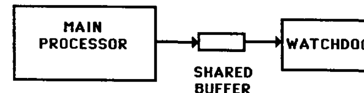


Fig. 17. Hardware organization for assertion checking.

write_buffer(assertion_number, space_needed, data)

where assertion_number is the assertion identifier, space_needed is memory space needed by the data, and data are the values of all the variables which are used in executing the assertion. The code for the assertions is transferred to the environment of the watchdog. All variables are renamed and referenced with respect to the start of the data packet for that particular assertion. The transformed programs for the main processor and the watchdog are shown in Table VIII.

3) During execution, the main processor writes to the shared buffer and the watchdog reads from the shared buffer.

The data must be transferred from the main processor to the watchdog as fast as possible to reduce the time overhead to the main processor. The transfer time includes time to write data to the shared buffer, time spent in handshaking protocol, and time spent waiting to get access to the buffer because of memory conflicts. The time to write data can be reduced by using small and fast buffers and handshaking time can be reduced by implementing most of the functions in hardware. The time wasted due to memory conflicts can be minimized by using clever buffering techniques. The use of dual buffers, queues, or dual-ported memories are examples of some of these techniques.

The watchdog can be specially designed to execute assertions very fast. One way to implement the watchdog would be to use RISC-type architecture. The code for assertions is stored in the local memory of the watchdog. The code is organized as shown in Fig. 18. The table-driven execution of assertions is needed because the sequence in which the assertions will be executed is data dependent (assertions can be in a multiple branching statement or in a loop). Each data pack sent by the main processor has an assertion number in it. The assertion number is used for table lookup to find the pointer to the executable code for that particular assertion (as done in microprogramming). All the data are referenced from the start of packet. As most of the time the watchdog will be doing comparisons, special hardware support is provided for this purpose.

The same watchdog can also be used to check for control flow errors explicitly, as described in [41]. This is done by representing the sequence in which assertions can be executed as a control flow graph, and then checking concurrently that in fact the assertions are executed in the order as specified by the control flow graph.

VI. Summary and Conclusions

A watchdog processor is a small and simple coprocessor used to perform concurrent system-level error detection by monitoring the behavior of a system. It detects errors in a main processor by comparing the relevant information, collected

TABLE VIII
THE TRANSFORMED PROGRAMS FOR ASSERTION CHECKING

Main Processor Program	Watchdog Program
begin	begin
.	read_next (assertion_number);
.	case assertion_number of
write_buffer (1, space_needed, data);	1: get (data); execute ASSERTION_1;
.	2: get (data); execute ASSERTION_2;
.	.
.	.
write_buffer (n, space_needed, data);	n: get (data); execute ASSERTION_n;
.	end;
.	
end;	end;

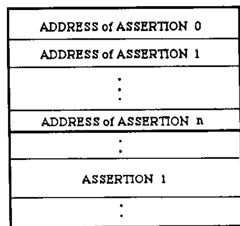


Fig. 18. Memory organization for assertion checking.

concurrently, to the information provided earlier. The information provided to the watchdog to detect errors can be about the memory access behavior, the control flow, the control signals, or the reasonableness of the results.

Two experimental studies were described that show that control flow and memory access checking can be used very effectively to detect errors. In schemes in which the watchdog monitors control flow, it detects errors by checking that the main processor traverses the control flow graph correctly. This is done by associating a signature (token or label) with each node (where a node represents some program unit). The same signatures are provided to the watchdog. During the execution of a program, the watchdog monitors the control flow, computes the node signatures concurrently (or accepts them from the main processor), and compares them to the signatures provided earlier. Two schemes were discussed; assigned-signature control flow checking in which the signatures are associated with the nodes arbitrarily and derived-signature control flow checking in which the signatures are derived from the nodes. It was shown that assigned-signature control flow checking only verifies that the nodes are executed in the allowed sequence, whereas derived-signature control flow checking can be used either to check the sequencing of the contents of the node or to do both, that is, verify the sequencing of the contents of a node and also check transitions among nodes. Structural integrity checking (SIC) is an example of the former technique and Basic Path Signature Analysis (PSA) is an example of the latter technique. The main advantages of assigned-signature checking are 1) depth and resolution of checking can be controlled by the programmer, and 2) asynchronous checking is easier. The disadvantage is that the signatures have to be explicitly transferred to the watchdog, which is not the case with derived-signature checking. Moreover, derived-signature checking provides much higher coverage of memory errors. Many schemes that reduce the time and memory overhead in Basic PSA, were discussed. One such scheme is branch address hashing in which some of the signatures can be combined with the branch addresses. Further improvement can be obtained by moving the signatures to the local memory of a watchdog. Both synchronous (Cerberus-16) and asynchronous (RMP) versions were discussed. It was shown that more than 90 percent of the memory and control flow errors can be detected with such schemes with hardware overhead (not including memory) of 10–20 percent and memory overhead of 10 percent. The techniques used for checking control flow can also be used for monitoring control units. Schemes for checking both hardwired and microprogrammed control units were described. The use of a watchdog processor for concurrent control flow checking was compared to the techniques used in the ESS-3A processor. It was shown that a watchdog provides better error coverage, especially for control flow errors, at comparable overhead cost.

A scheme for detecting errors by monitoring the memory access behavior was also discussed. The scheme is based on capability-based addressing. The watchdog which monitors the control flow can be extended easily to check the memory access, thereby providing higher error coverage.

Control flow and memory access checking do not detect semantic or data manipulation errors. Such errors can be detected by having a watchdog execute assertions concurrently about the program being executed on the main processor. The main problem is to be able to transfer data from the main processor to the watchdog without excessive time overhead. Design of special purpose watchdogs that make use of either the data flow information or the cyclic nature of some programs to transfer data were described. Also, a design of a general purpose watchdog based on message passing was discussed. It was shown how the same watchdog can be used both for control flow and data checking.

From the details presented in this paper, it is reasonable to conclude that watchdog processors provide a viable alternative to the current concurrent error detection schemes. They (watchdog processors) can be used independently or in addition to the existing circuit-level error detection techniques. Error detection by means of watchdogs does not rely on traditional fault models nor does it use massive replication. A great advantage of the watchdog processor is that it provides an independent circuitry for error detection, at a reasonable overhead cost. Moreover, the use of watchdog processors is more in the spirit of distributed computing, where dedicated processors are used to perform specialized tasks. There are many ways of increasing the reliability of the watchdog itself. The use of built-in self test is one way. The other is the duplication of the watchdog.

ACKNOWLEDGMENT

The authors would like to thank all the members of Center for Reliable Computing (especially H. Amer and L. T. Wang) for many useful suggestions and comments.

REFERENCES

[1] R. J. Andrews, J. J. Driscoll, J. A. Herndon, P. C. Richards, and L. R. Roberts, "Service features and call processing," *Bell Syst. Tech. J.*, vol. 48, pp. 2713-2764, Oct. 1969.

[2] D. M. Andrews, "Software fault tolerance through executable assertions," in *Conf. Rec. 12th Asilomar Conf. Circuits, Syst., Comput.*, Pacific Grove, CA, Nov. 6-8, 1978, pp. 641-645.

[3] ——, "Using excutable assertions for testing and fault tolerance," in *Dig. 9th Annu. Int. Symp. Fault Tolerant Comput., FTCS-9*, Madison, WI, June 20-22, 1979, pp. 102-105.

[4] D. M. Andrews and J. P. Benson, "An automated program testing methodology and its implementation," in *Proc. 5th Int. Conf. Software Eng.*, San Diego, CA, Mar. 9-12, 1981, pp. 254-261.

[5] A. Avizienis, "Fault tolerance by means of external monitoring of computer systems," in *Proc. AFIPS Conf.*, vol. 50, Chicago, IL, May 4-7, 1981, pp. 27-40.

[6] A. Avizienis and J. P. J. Kelly, "Fault tolerance by design diversity: Concepts and experiments," *Computer*, vol. 17, pp. 67-80, Aug. 1984.

[7] G. E. Bendixen, "The digital flight control and active control systems on the L-1011," in *Proc., IEEE/AIAA, 5th Digital Avion. Syst. Conf.*, Seattle, WA, Oct. 31-Nov. 3, 1983, pp. 11.2.1-11.2.11.

[8] N. Benowitz, "An advanced fault isolation system for digital logic," *IEEE Trans. Comput.*, vol. C-24, pp. 489-497, May 1975.

[9] L. Chen and A. Avizienis, "N-Version programming: A fault-tolerance approach to reliability of software operation," in *Dig. Papers Eighth Annu. Int. Conf. Fault-Tolerant Comput., FTCS-8*, Toulouse, France, June 21-23, 1978, pp. 3-9.

[10] J. R. Connet, E. J. Pasternak, and B. D. Wagner, "Software defenses in real time control systems," in *Dig. Int. Symp. Fault Tolerant Comput., FTCS-2*, Newton, MA, June 19-21, 1972, pp. 94-99.

[11] R. W. Cook, W. H. Sisson, T. F. Storey, and W. N. Toy, "Design of self-checking microprogram control," *IEEE Trans. Comput.*, vol. C-22, pp. 255-262, Mar. 1973.

[12] F. Cristian, "Exception handling and software fault tolerance," *IEEE Trans. Comput.*, vol. C-31, pp. 531-540, June 1982.

[13] Y. Crouzet and J. Chavade, "A 6800 coprocessor for error detection in microcomputers: The PAD," *Proc. IEEE*, vol. 74, pp. 723-731, May 1986.

[14] S. F. Daniels, "A concurrent test technique for standard microprocessors," in *Dig. Papers Compcon Spring 83*, San Francisco, CA, Feb. 28-Mar. 3, 1983, pp. 389-394.

[15] J. B. Eifert and J. P. Shen, "Processor monitoring using asynchronous signatured instruction streams," in *Dig., 14th Int. Conf. Fault-Tolerant Comput., FTCS-14*, Kissimmee, FL, June 20-22, 1984, pp. 394-399.

[16] G. Estrin, "Snuper Computer—A computer in instrumentation automation," in *Proc. AFIPS Spring Joint Comput. Conf.*, vol. 30, Atlantic City, NJ, April 18-20, 1967, pp. 645-656.

[17] R. S. Fabry, "Capability-based addressing," *Commun. ACM*, vol. 17, pp. 403-412, July 1974.

[18] R. W. Floyd, "Assigning meaning to programs," in *Proc. Symp. Appl. Math.*, vol. 19, Amer. Math. Soc., Providence, RI, 1967, pp. 19-32.

[19] G. E. Forsythe, M. A. Malcolm, and C. B. Moler, *Computer Methods for Mathematical Computations.* Englewood Cliffs, NJ: Prentice-Hall, 1977, ch. 3.

[20] S. Z. Hassan, D. J. Lu, and E. J. McCluskey, "Parallel signature analyzers—Detection capability and extensions," in *Dig. Papers Compcon Spring 83*, San Francisco, CA, Feb. 28-Mar. 3, 1983, pp. 440-445.

[21] S. Z. Hassan and E. J. McCluskey, "Enhancing the effectiveness of parallel signature analyzers," in *Dig., IEEE ICCAD-84*, Santa Clara, CA, Nov. 12-15, 1984, pp. 102-104.

[22] C. A. R. Hoare, "An axiomatic basis of computer programming," *Commun. ACM.*, vol. 12, pp. 576-580, Oct. 1969.

[23] A. L. Hopkins, T. B. Smith, and J. H. Lala, "FTMP—A highly reliable fault-tolerant multiprocessor for aircraft," *Proc. IEEE*, vol. 66, pp. 1221-1239, Oct. 1978.

[24] J. C. Huck, "Comparative analysis of computer architectures," Tech. Rep. 83-243, Comput. Syst. Lab., Stanford University, Stanford, CA 94305, May 1983.

[25] E. A. Irland and U. K. Stagg, "New developments in suburban and rural ESS (No. 2 and No. 3 ESS)," *Rec., Int. Switching Symp.*, Munich, West Germany, Sept. 9-13, 1974, pp. 512/1-512/7.

[26] V. S. Iyengar and L. L. Kinney, "Concurrent fault detection in microprogrammed control units," *IEEE Trans. Comput.*, vol. C-34, pp. 810-821, Sept. 1985.

[27] J. R. Kane and S. S. Yau, "Concurrent software fault detection," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 87-99, Mar. 1975.

[28] K. H. Kim and C. V. Ramamoorthy, "Failure-tolerant parallel programming and its supporting system architecture," in *AFIPS Conf. Proc. (National Comput. Conf.)*, vol. 45, New York, NY, June 7-10, 1976, pp. 413-423.

[29] B. Krieg-Bruckner and D. C. Luckham, "ANNA: Towards a language for annotating Ada programs," *ACM SIGPLAN Notices*, vol. 15, pp. 128-138, Nov. 1980.

[30] P. A. Lee, N. Ghani, and K. Heron, "A recovery cache for PDP-11," in *Dig. Papers 9th Annu. Int. Symp. Fault Tolerant Comput., FTCS-9*, Madison, WI, June 20-22, 1979, pp. 3-7.

[31] Y. H. Lee and K. G. Shin, "Design and evaluation of a fault-tolerant multiprocessor using hardware recovery blocks," *IEEE Trans. Comput.*, vol. C-33, pp. 113-124, Feb. 1984.

[32] N. G. Leveson and P. R. Harvey, "Analyzing software safety," *IEEE Trans. Software Eng.*, vol. SE-9, pp. 569-579, Sept. 1983.

[33] K. W. Li, "Detection of transient faults in microprocessors by means of external hardware," M.Sc. Thesis, Dep. Elec. Eng., Virginia Polytechnic Institute and State Univ., Blacksburg, VA, Mar. 1984.

[34] T. S. Liu, "The role of a maintenance processor for a general-purpose computer system," *IEEE Trans. Comput.*, vol. C-33, pp. 507-517, June 1984.

[35] D. J. Lu, "Watchdog processors and VLSI," in *Proc. Nat. Electron. Conf.*, vol. 34, Chicago, IL, Oct. 27-28, 1980, pp. 240-245.

[36] ——, "Watchdog processor and structural integrity checking," *IEEE Trans. Comput.*, vol. C-31, pp. 681-685, July 1982.

[37] A. Mahmood, D. J. Lu, and E. J. McCluskey, "Concurrent fault detection using a watchdog processor and assertions," in *Proc. 1983 Int. Test Conf.*, Philadelphia, PA, Oct. 18-20, 1983, pp. 622-628.

[38] A. Mahmood, D. M. Andrews, and E. J. McCluskey, "Writing executable assertions to test flight software," in *Conf. Rec. 18th Annu. Asilomar Conf. Circuits, Syst., Comput.*, Pacific Grove, CA, Nov. 5-7, 1984, pp. 262-266.

[39] ——, "Executable assertions and flight software," in *Proc. AIAA/IEEE 6th Digital Avion. Syst. Conf.*, Baltimore, MD, Dec. 3-6, 1984, pp. 346-351.

[40] A. Mahmood and E. J. McCluskey, "Watchdog processors: Error coverage and overhead," in *Dig. 15th Annu. Int. Symp. Fault-Tolerant Comput., FTCS-15*, Ann Arbor, MI, June 19-21, 1985, pp. 214-219.

[41] A. Mahmood, A. Ersoz, and E. J. McCluskey, "Concurrent system level error detection using a watchdog processor," in *Proc. 1985 Int. Test Conf.*, Philadelphia, PA, Nov. 19-21, 1985, pp. 145-152.

[42] A. Mahmood and E. J. McCluskey, "Concurrent error detection using watchdog processors—A survey," CRC Tech. Rep. 85-7, CSL TR. 85-266, Center Reliable Comput., Computer Systems Lab., Stanford Univ., Stanford, CA 94305, June 1985.

[43] M. Namjoo and E. J. McCluskey, "Watchdog processors and capability checking," in *Dig. Papers 12th Annu. Int. Symp. Fault Tolerant Comput., FTCS-12*, Santa Monica, CA, June 22-24, 1982, pp. 245-248.

[44] M. Namjoo, "Techniques for concurrent testing of VLSI processor operation," in *Dig. 1982 Int. Test Conf.*, Philadelphia, PA, Nov. 15-18, 1982, pp. 461-468.

[45] M. Namjoo, "Design of concurrently testable microprogrammed control units," in *Proc. 15th Annu. Workshop Microprogramming, MICRO-15*, Palo Alto, CA, Oct. 1982, pp. 173-180.

[46] M. Namjoo, "CERBERUS-16: An architecture for a general purpose watchdog processor," in *Dig. Papers 13th Annu. Int. Symp. Fault Tolerant Comput. FTCS-13*, Milano, Italy, June 28-30, 1983, pp. 216-219.

[47] J. S. Novak and L. S. Tuomenoksa, "Memory mutilation in stored program controlled telephone systems," in *Conf. Rec. 1970 Int. Conf. Commun.*, vol. 2, 1970, pp. 43-32 to 43-45.

[48] S. M. Ornstein, W. R. Crowther, M. F. Kraley, R. D. Bressler, A. Michel, and F. E. Heart, "Pluribus—A reliable multiprocessor," in *Proc. AFIPS Conf.*, vol. 44, Anaheim, CA, May 19-22, 1975, pp. 551-559.

[49] B. Randell, "System structure for software fault tolerance," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 220-232, June 1975.

[50] S. H. Saib, "Executable assertions—An aid to reliable software," in *Conf. Rec. 11th Asilomar Conf. Circuits, Syst., Comput.*, Pacific Grove, CA, Nov. 7-9, 1977, pp. 277-281.

[51] S. H. Saib, "Distributed architectures for reliability," in *Proc. AIAA*

Conput. Aerosp. Conf., Los Angeles, CA, Oct. 22–24, 1979, pp. 458–462.

[52] M. E. Schmid, R. L. Trapp, A. E. Davidoff, and G. M. Masson, "Upset exposure by means of abstract verification," in *Dig. Papers 12th Annu. Int. Symp. Fault Tolerant Comput. FTCS-12*, Santa Monica, CA, June 22–24, 1982, pp. 237–244.

[53] M. A. Schuette, J. P. Shen, D. P. Siewiorek, and Y. X. Zhu, "Experimental evaluation of two concurrent error detection schemes," in *Dig. 16th Annu. Int. Symp. Fault-Tolerant Comput., FTCS-16*, Vienna, Austria, July 1–4, 1986, pp. 138–143.

[54] R. M. Sedmak and H. L. Liebergot, "Fault-tolerance of a general purpose computer implemented by very large scale integrating," *IEEE Trans. Comput.*, vol. C-29, pp. 492–500, June 1980.

[55] J. P. Shen and M. A. Schuette, "On-line self-monitoring using signatured instruction streams," in *Proc. 1983 Int. Test Conf.*, Philadelphia, PA, Oct. 18–20, 1983, pp. 275–282.

[56] L. J. Shustek, "Analysis and performance of computer instruction sets," SLAC Rep. 205, STAN-CS-78-658, Stanford Univ., Stanford, CA 94305, May 1978.

[57] D. P. Siewiorek, V. Kini, H. Mashburn, S. R. McConnel, and M. M. Tsao, "A case study of C.mmp, Cm*, and C.vmp: Part 1—Experiences with fault tolerance in multiprocessor systems," *Proc. IEEE*, vol. 66, pp. 1178–1199, Oct. 1978.

[58] D. P. Siewiorek and R. S. Swarz, *The Theory and Practice of Reliable System Design.* Bedford, MA: Digital, 1982, ch. 3.

[59] T. Sridhar and S. M. Thatte, "Concurrent checking of program flow in VLSI processors," in *Dig. 1982 Int. Test Conf.*, Philadelphia, PA, Nov. 15–18, 1982, pp. 191–199.

[60] R. E. Staehler, "Organization and objectives," *Bell Syst. Tech. J.*, vol. 56, pp. 119–134, Feb. 1977.

[61] T. F. Storey, "Design of a microprogram control for a processor in an electronic switching systems," *Bell Syst. Tech. J.*, vol. 55, pp. 183–232, Feb. 1976.

[62] L. G. Stucki and G. L. Foshee, "New assertion concepts for self metric software validation," in *Proc. Int. Conf. Reliable Software*, Los Angeles, CA, Apr. 21–23, 1975, pp. 59–71.

[63] D. J. Taylor and J. P. Black, "Principles of data structure error correction," *IEEE Trans. Comput.*, vol. C-31, pp. 602–608, July 1982.

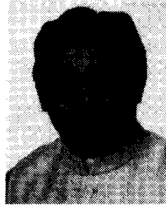
[64] S. M. Thatte and J. A. Abraham, "Test generation for microprocessors," *IEEE Trans. Comput.*, vol. C-29, pp. 429–441, June 1980.

[65] S. P. Tomas and J. P. Shen, "A roving monitoring processor for detection of control flow errors in multiple processor systems," in *Proc. IEEE Int. Conf. Comput. Design: VLSI Comput.*, Port Chester, NY, Oct. 7–10, 1985, pp. 531–539.

[66] J. H. Wensley, L. Lamport, J. Goldberg, M. W. Green, K. N. Levitt, P. M. Melliar-Smith, R. E. Shostak, and C. B. Weinstock, "SIFT; Design and analysis of a fault-tolerant computer for aircraft control," *Proc. IEEE*, vol. 66, pp. 1240–1255, Oct. 1978.

[67] S. S. Yau and Fu-Chung Chen, "An approach to concurrent control flow checking," *IEEE Trans. Software Eng.*, vol. SE-6, pp. 126–137, Mar. 1980.

[68] L. J. Yount, "Architectural solutions to safety problems of digital flight-critical systems for commercial transports," in *Proc. AIAA/IEEE 6th Digital Avion. Syst. Conf.*, Baltimore, MD, Dec. 3–6, 1984, pp. 28–35.



Aamer Mahmood (S'78–M'87) received the B.S.E.E. degree (with Honors) from University of Engineering and Technology, Lahore, Pakistan in 1979 and the M.S.E.E. and Ph.D. degrees in electrical engineering from Stanford University, Stanford, CA, in 1986.

Currently he is working as Member Technical Staff in CPU Development at ROLM MIL-SPEC Computers. His interests include concurrent checking, fault tolerance, design for testability, computer architecture, and parallel computing.

E. J. McCluskey (S'51–M'55–SM'59–F'65) received the A.B. degree (summa cum laude) in mathematics and physics from Bowdoin College, Brunswick, ME, in 1953, and the B.S., M.S., and Sc.D. degrees in electrical engineering from Massachusetts Institute of Technology, Cambridge in 1953, 1953, and 1956, respectively.

He worked on electronic switching systems at the Bell Telephone Laboratories from 1955 to 1959. In 1959, he moved to Princeton University, Princeton, NJ, where he was Professor of Electrical Engineering and Director of the University Computer Center. In 1966 he joined Stanford University, where he is Processor of Electrical Engineering and Computer Science, as well as Director of the Center for Reliable Computing. He has published several books and book chapters. His most recent book is *Logic Design Principles with Emphasis on Testable Semicustom Circuits*, (Englewood Cliffs, NJ: Prentice-Hall, 1986), book chapters include Design for Testability in *Fault-tolerant Computing*, edited by D. K. Pradhan, and chapters on Logic Design in the Van Nostrand Reinhold *Encyclopedia of Computer Science and Engineering* and in *Reference Data for Engineers*, edited by E. C. Jordan. He is President of Stanford Logical Systems Institute which provides consulting services on fault-tolerant computing, testing, and design for testability.

Dr. McCluskey served as the first President of the IEEE Computer Society and as a member of the AFIPS Executive Committee. He is a member of the Organizing Committees of the IEEE DFT and BIST Workshops and the Program Committees of ICCAD'87 and FTCS-17. He is Associate Editor of the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN, a Technical Advisor for *VLSI Systems Design*, and on the Editorial Board of the *TSI Journal*. He was a founding member of the Editorial Board of *IEEE Design and Test* Magazine. In 1984, he received the IEEE Centennial Medal and IEEE Computer Society Technical Achievement Award in Testing.