troller), it has the advantage of maintaining sequential consistency, thus allowing parallel programs to work as expected.

VII. Summary and Conclusions

The delays due to error checking being performed in series with intermodule communication are one of the primary causes of performance degradation associated with implementing concurrent error detection and correction in VLSI systems. This performance penalty may be caused by dedicated checkers in the communication paths between each module and the rest of the system or the need to wait for the hardware to perform redundant operations that verify the validity of the "recent" results. Checks that are area efficient are usually serial and slow, and faster checks are often based on checkers which take up large area, thus slowing down surrounding circuits. Checking delays are compounded when data are transferred between several modules, and checked at several places.

This fundamental problem in achieving fault tolerance in high-performance VLSI systems can be overcome by performing checks on the data in parallel with intermodule communication. Data to be checked can be latched, and error checking can take place in one or more subsequent cycles (as a pipeline). As a consequence, error signals can arrive one or more cycles after error-damaged data are received for processing. In this paper, we have described a mechanism, called *micro rollback,* which allows checking to proceed in parallel with communication by supporting fast rollback of a few cycles when a delayed error signal arrives. Micro rollback is a powerful technique that facilitates the implementation of high-performance VLSI systems which are also highly fault tolerant. It allows a variety of concurrent error detection and correction techniques to be used with minimal performance penalty. With micro rollback, it is feasible to operate systems in hostile environments, where there is a high rate of transient faults, due to the ability of individual modules to initiate rollback and retry which can complete in a few cycles without resorting to expensive system-wide rollbacks.

We have presented a systematic way to design VLSI computer modules that can roll back and restore the state which existed when the error occurred. Specifically, the implementation of micro rollback in simple synchronous systems involves replication of small isolated registers and the use of full *delayed-write buffers* (DWB's) for storing recent state changes to large register files. When applied to a VLSI RISC processor, the micro rollback technique is characterized by extremely low performance overhead and a modest area overhead compared to the area of the entire processor. We have shown how micro rollback can be used in a complete system with a memory hierarchy and multiple processors. The various subsystems presented are representative of many common modules in a wide variety of VLSI systems and are thus new critical building blocks which are likely to have wide application in future systems that will combine fault tolerance and high performance.

References

[1] L. Censier and P. Feautrier, "A new solution to coherence problems in multicache systems," *IEEE Trans. Comput.*, vol. C-27, no. 12, pp. 1112–1118, Dec. 1978.

[2] M. L. Ciacelli, "Fault handling on the IBM 4341 processor," in *Proc. 11th Fault-Tolerant Comput. Symp.*, Portland, ME, June 1981, pp. 9–12.

[3] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Trans. Comput.*, vol. C-28, no. 9, pp. 690–691, Sept. 1979.

[4] W. W. Hwu and Y. N. Patt, "Checkpoint repair for out-of-order execution machines," in *Proc. 14th Annu. Symp. Comput. Architecture,* Pittsburgh, PA, June 1987, pp. 18–26.

[5] J. H. Patel and L. Y. Fung, "Concurrent error detection in ALU's by recomputing with shifted operands," *IEEE Trans. Comput.*, vol. C-31, no. 7, pp. 589–595, July 1982.

[6] B. Randell, P. A. Lee, and P. C. Treleaven, "Reliability issues in computing system design," *Comput. Surveys,* vol. 10, no. 2, pp. 123–165, June 1978.

[7] R. W. Sherburne, M. G. H. Katevenis, D. A. Patterson, and C. H. Séquin, "A 32-Bit NMOS microprocessor with a large register file," *IEEE J. Solid-State Circuits,* vol. SC-19, no. 5, pp. 682–689, Oct. 1984.

[8] J. E. Smith and A. R. Pleszkun, "Implementing precise interrupts in pipelined processors," *IEEE Trans. Comput.*, vol. C-37, no. 5, pp. 562–573, May 1988.

[9] Y. Tamir, M. Tremblay, and D. A. Rennels, "The implementation and application of micro rollback in fault-tolerant VLSI systems," in *Proc. 18th Fault-Tolerant Comput. Symp.*, Tokyo, Japan, June 1988, pp. 234–239.

Control-Flow Checking Using Watchdog Assists and Extended-Precision Checksums

NIRMAL R. SAXENA AND EDWARD J. McCLUSKEY

Abstract—**A control-flow checking method using extended-precision checksums and watchdog assists is proposed. Control-flow checking based on extended-precision checksums is shown to have low error detection latency compared to previously proposed methods. Analytical measures are derived to demonstrate the effectiveness of using extended-precision checksums for control flow checking. It is shown that the error detection latency in the extended-precision checksum-based control-flow checking remains relatively constant both for single and multiple sequence errors. In the case of signature-based methods, error detection latency increases linearly with the number of sequence errors.**

Previous work did not address several architectural issues that relate to the support of control-flow checking. A watchdog assist architecture for control-flow checking in programs is proposed which addresses these issues. This watchdog assist architecture can support control-flow checking for multiprocessor, multiprogramming, and cache-based environments. The Hewlett-Packard Precision Architecture is used as an example architecture to demonstrate the feasibility of watchdog assists.

Index Terms—**Checksums, concurrent error detection, coprocessors, signatures, watchdog assists.**

I. Introduction

This paper has two parts. In the first part, a *watchdog assist* architecture that supports *control-flow checking* [1], [2] at the assembly instruction level is proposed. In the second part, some metrics are developed that quantify the effectiveness of using *extended-precision checksums* [3] as a control-flow checking method. The watchdog assist is scalable and extensible to multiprocessor, multiprogramming, and cache-based environments. The proposed architecture does not significantly affect the system architecture. For example, the introduction of watchdog assist in the system does not require any changes in the bus design, the memory system design, the cache architecture, or the CPU design. The use of extended-precision checksums in the context of control-flow checking has several demonstrable advantages. These advantages are the following.

1) Extended-precision checksums provide the capability of a watch-

Manuscript received July 6, 1989; revised November 23, 1989. This work was supported in part by the Hewlett-Packard Resident Fellowship, in part by the National Science Foundation under Grant MIP-8709128, and in part by the Innovative Science and Technology Office of the Strategic Defense Initiative Organization administered through the Office of Naval Research under Contract N000014-85-K-0600.

N. R. Saxena is with Hewlett-Packard, Cupertino, CA 95014.

E. J. McCluskey is with the Center for Reliable Computing, Computer Systems Laboratory, Departments of Electrical Engineering and Computer Science, Stanford University, Stanford, CA 94305.

IEEE Log Number 8933875.

dog timer without actually requiring a separate watchdog timer. This helps in the detection of infinite loops due to control-flow errors.

2) Extended-precision checksums most effectively exploit natural redundancy occurring in program codes. This redundancy helps both in enhancing error detection coverage and reducing error detection latency.

The control-flow checking methods have been classified [1] into two categories: *assigned-signature checking* and *derived-signature checking*. Derived-signature checking is the control-flow checking method considered in this paper. In this paper, *bit errors* and *sequence errors* [4] are assumed in the control-flow model. Results reported in [12] further validate this error model. Bit errors alter one or more instruction bits. Sequence errors correspond to those failures that result in incorrect program flow. The effectiveness metrics studied in this paper are error latency and error coverage. The issues about memory overhead and performance loss due to control-flow checking are not considered because of the restrictions on the length of this paper.

II. Architectural Visibility of Control Flow Checking

In [1], the proposed watchdog processor architecture was at the system bus level. This architecture is not useful for control-flow checking at the machine instruction level for a multiprocessor, multiprogramming, or cache-based environment. Also, previous papers [2], [4]–[6] have not examined several architectural aspects of control-flow checking in detail. The architectural visibility of the apparatus involved in control-flow checking is an important factor to consider. The control-flow checking apparatus concurrently monitors and computes some attribute of the program and therefore carries some state of program. The following are some of the reasons why control-flow checking should be architecturally visible in a given computer system.

1) In a multiprogramming environment, the state of the control-flow checker needs to be saved and restored due to context switches.

2) In a multiprocessor environment, process migration may require moving the state of the control-flow checker from one processor to another.

3) The state save/restore sequence may be necessary due to power fail/error recovery.

4) In an asymmetric multiprocessor environment, some processors may not have control-flow checkers. The operating system may need to identify the presence of control-flow checkers in processors through some architectural mechanisms. This may be necessary because the operating system has to schedule processes that use control-flow checking to processors that have control-flow checkers.

5) The state save/restore sequence has to be carried through load/store instructions or some such mechanisms defined by the architecture.

In addition to the above requirements, there needs to be a mechanism that differentiates control-flow checker instructions from the rest of the instructions in the program. Two approaches have been suggested. 1) control-flow checker instructions can be identified by the use of additional bits in the instruction word [5]. 2) Special opcodes [2] in the instruction set can define instructions for control-flow checkers. The latter approach is used in this paper. The former approach is not attractive because:

1) This approach increases memory overhead and affects the memory system design. The memory system has to support new data alignment requirements due to the addition of extra bits to the instruction word. Typically, instructions and data are stored in the same memory. The data portion of the program need not carry additional bits because the control-flow checker does not monitor the data part of the program. To minimize memory overhead, the memory system may have to distinguish instruction storage from data storage. This increases the complexity of memory system design.

2) The bus architecture is affected because it has to support the transfer of additional bits appended to the instructions. In order to optimize performance and increase the bus bandwidth, the bus system may have to distinguish data transfers from instruction transfers.

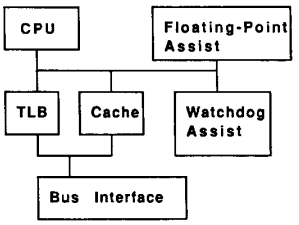

Fig. 1. Watchdog coprocessor.

This may involve creating new read/write transactions in the bus architecture.

3) Like the memory system, the cache architecture and the processor module architecture are also affected.

4) This approach penalizes user processes that do not use control-flow checking. If this is to be avoided then the entire system has to distinguish between processes that use control-flow checking and processes that do not use control-flow checking.

In short, the support of the former approach causes major perturbations in the system architecture. The latter approach is more feasible but is instruction set architecture dependent. There has been a recent trend [1], [7] to use dedicated processors to perform specialized functions. The popularity of these techniques supports the idea of using a dedicated watchdog coprocessor for concurrent error detection. For example, *Hewlett-Packard Precision Architecture* (HPPA) [7] can support up to eight coprocessors. HPPA allows for flexible instruction set extensions by means of assist instructions. *Assist instructions* are instructions in which the data movement functions are defined between the CPU or the memory and the assist hardware, but the data transformations are left unspecified. Assist instructions that specify data movement functions are essentially coprocessor load/store instructions. These instructions provide a generic interface for moving data to and from the coprocessor. Data transformation assist instructions provide the flexibility of defining data manipulation operations for each coprocessor depending upon the application. Examples exist among other architectures [8] that provide similar functionality. Only HPPA examples are used in this paper. Fig. 1 illustrates the watchdog coprocessor configuration. Communication with the watchdog is established via coprocessor instructions. This approach allows the watchdog coprocessor to inherit all the data movement functions (loads and stores) and to specify its own data transformation operations. For control-flow checking, the data transformation could be linear feedback shift register based signature computation or adder-based checksum computation.

The watchdog coprocessor approach to control-flow checking is attractive because of the following attributes.

1) A watchdog coprocessor can be added to any system that supports future definition of coprocessors [7], [8]. The addition of watchdog does not cause any major perturbation in the system architecture. For systems that do not support future definition of coprocessors or do not have extensible instruction sets, control-flow checking support is difficult unless some design changes are made. The notion of coprocessors gives a generic interface to specialized function units and simplifies the design of software.

2) In previous approaches [1], [9], the presence of cache in processors and bus arbitration protocols cause latency in error detection. Watchdog coprocessors do not experience any error detection latency due to cache or bus arbitration protocols.

3) This approach is easily extensible to multiprogramming and multiprocessor environments. During context switches, process migration, or power fail/error recovery, generic state save/restore routines can be used by the software for the watchdog coprocessor. Watchdog coprocessors inherit all the architecture defined interruption/trap and coprocessor identification mechanisms (to indicate the presence or absence of a particular coprocessor). This simplifies the design of system software.

4) User processes that do not use control-flow checking do not incur any performance overhead. The lazy state save/restore ap-

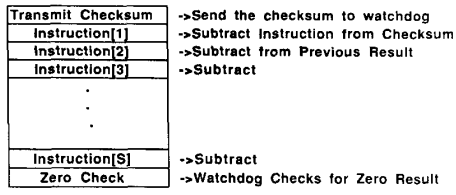

Fig. 2. Extended-precision-based control-flow checking.

proach can be used to minimize the context switch overhead. This overhead exists because the operating system has to save/restore the state of watchdog coprocessor during context switches. In lazy state save/restore approach, the state of the watchdog coprocessor is saved, during a context switch, only if the process which is brought in as a result of the context switch uses the watchdog coprocessor.

III. Extended-Precision Checksum-Based Control-Flow Checking

Fig. 2 represents a block of branch-free instructions and illustrates the use of extended-precision checksums in control-flow checking. Extended-precision checksum of the instruction block is the sum-total of instruction_ 1 through instruction_S or the sum-total of some transformation over these instructions. Explicit watchdog coprocessor instructions first transmit the extended-precision checksum of the instruction block *instruction_*1 through *instruction_S*.

Typically, the block length is small and therefore fewer bits are required to encode the checksum value. In general, an *n-bit* signature or an *n-bit* checksum cannot be transmitted using a single instruction. However, a *load single word* instruction can be used but it has a performance penalty. If the instructions are space compacted from *n-bits* to *r-bits* ($r < n$) and the extended-precision checksum is computed on the compacted instructions then it is possible to transmit the checksum value in a single instruction. After the watchdog receives the extended-precision checksum value, it starts subtracting instructions as illustrated in Fig. 2. At the end of the block, the watchdog checks for an all-zero result. This zero check can be triggered by the beginning of a branch instruction. Clearly, error detection by this approach is threefold.

1) An error is detected if the watchdog result becomes negative before it receives a zero check signal.

2) An error is detected if the result in the watchdog is nonzero when it receives a zero check signal.

3) An error is detected if the watchdog receives a zero check signal before the checksum is transmitted.

Since the error can be detected even before the zero check signal is transmitted, the error detection latency can be reduced. This kind of early detection is not possible if signature analysis, two's complement checksums, one's complement checksums, or honeywell checksums [11] are used. Early detection is possible because of the monotonic nature of extended-precision checksums. This monotonic nature also helps in detecting infinite loops, reducing error-detection latency in multiple sequence-errors, and enhancing error detection coverage. The following section presents a notation and definitions used in obtaining effectiveness metrics for extended-precision checksums.

A. Notation and Definitions

Definition 1: An n-bit word A_i is defined as a string of binary symbols $a_{j,i}$ and is denoted as $a_{n,i}a_{n-1,i}\cdots a_{1,i}$. The magnitude $|A_i|$ of A_i is given by

$$|A_i| = \sum_{j=1}^{n} 2^{j-1} a_{j,i}.$$

Definition 2: A block of S n-bit words $[A_1, A_2, \cdots, A_S]^T$ will be denoted by an $S \times n$ matrix.

Definition 3: The checksum K of an $S \times n$ matrix is given by

$$K = \sum_{i=1}^{S} |A_i|.$$

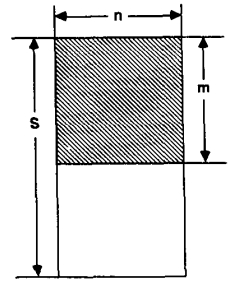

Fig. 3. Bit errors in m instructions.

If the addition is performed without any loss of precision then K is the extended-precision checksum. $K \bmod 2^n$ and $K \bmod (2^n - 1)$ are two's complement and one's complement checksums, respectively.

Definition 4: $C(S, K, n)$ is defined as the number of possible distinct $S \times n$ matrices that have the same extended-precision checksum K.

Definition 5: $G_i(Z) = \sum_{q=0}^{2^n-1} i_q Z^q$ is the probability generating function of the static distribution of *n-bit* wide instructions. The coefficient i_q is the probability of finding an instruction with encoding corresponding to value q in the program code.

Definition 6: $G_b(Z) = \sum_{q=0}^{T} b_q Z^q$ is the probability generating function of the static distribution of branch-instruction-free block lengths. The coefficient b_q is the probability of finding a branch-free instruction block length of q in the program code.

B. Bit-Error Detection Latency and Coverage

The bit-error detection latency is the average number of instruction cycles to detect bit errors. Suppose a block of m instructions has bit errors as illustrated in Fig. 3.

An upper bound on the bit-error detection latency $\mathrm{Lat}_b(m, K_S)$ is estimated. K_S is the extended-precision checksum of S instructions shown in Fig. 3. The probability that the error is not detected in the $(h-1)$th instruction cycle is the probability of having a checksum value J less than or equal to K_S. The probability of detecting the error in the hth cycle (given that the sum of $(h-1)$ instructions is J) is

$$\frac{1}{2^n} \sum_{\alpha > K_S - J} C(1, \alpha, n).$$

Summing over all J, the probability of detecting the bit errors exactly in the hth cycle is given by

$$\frac{1}{2^{hn}} \sum_{J \le K_S} C(h-1, J, n) \sum_{\alpha > K_S - J} C(1, \alpha, n).$$

Assume that if the bit errors are not detected in the first m cycles then they are detected only after an additional $S - m$ cycles. This gives an upper bound on latency which is

$$\mathrm{Lat}_b(m, K_S) = \sum_{h=1}^{m-1} \frac{h-1}{2^{hn}} \sum_{J \le K_S} C(h-1, J, n) \cdot \sum_{\alpha > K_S - J} C(1, \alpha, n) + \frac{(S-m)}{2^{mn}} \sum_{J \le K_S} C(m, J, n).$$

Fig. 4 illustrates the dependence of $\mathrm{Lat}_b(m, K_S)$ on m and K_S. The plots are for $S = 10$ and $n = 3$. It is quite apparent from the characteristics illustrated by Fig. 4 that low values of K_S reduce latency substantially. In signature-based methods, the latency will always be $S - 1$.

For extended-precision checksums to be effective, it is desirable that the static distribution of instructions be nonuniform. Extensive study of the distribution of instructions in program codes of the various computer architectures has been done in [10]. Results in [10]

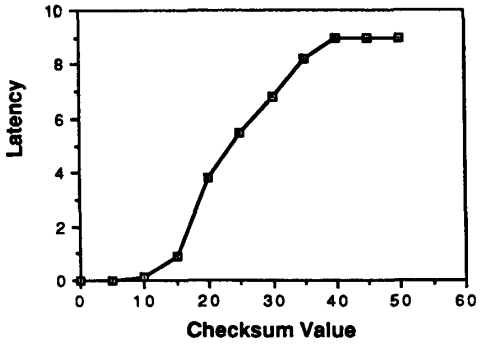

Fig. 4. Bit error latency.

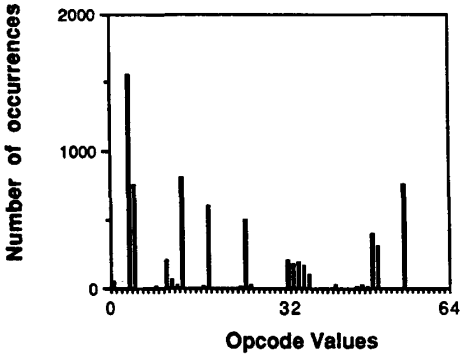

Fig. 5. Static distribution of HPPA instructions.

show a nonuniform static distribution of high-level language statements. It is quite likely that instructions in the machine code of these high-level programs would also have nonuniform distribution characteristics. Experimental results presented in [3] demonstrate this. For example, Fig. 5 illustrates a static distribution of HPPA instructions based on the major 6-bit opcode field. This distribution is of a typical application code. Nonuniform distribution of instructions is quite apparent from Fig. 5.

From an information theoretic point of view, most of the program codes (at the machine instruction level) have inherent redundancy. This is so because not all instruction encodings are meaningful; for example, a 32-bit instruction may not have meaning for all 2^{32} encodings of the instruction word. Also, in most of the program codes some instructions occur more often than the others. An analogy can be drawn with the encoding of decimal digits. Again from an information theoretic standpoint only 3.3219 ($\log_2 10$) bits are required to represent decimal digits. However, the number of bits must be a whole number; therefore, 4 bits are chosen to represent decimal digits. This inherent or natural redundancy cannot be avoided if a regular representation of decimal numbers is desired. Sometimes this natural redundancy is well suited for error detection. In checksum-protected program codes, it is highly desirable to move the checksum value far away from the mean value [3]; this will decrease the error masking in extended-precision checksums significantly. Next, it will be shown how this natural redundancy in program codes can be exploited to reduce error detection latency.

A typical instruction word is a structured field having an *opcode, register*, and other *opcode-extension* fields. Let us assume that the opcode field is the most significant field in the instruction word. If opcodes are encoded such that

- an all zero code is assigned to that instruction opcode which has, on the average, a high frequency of occurrence in program codes, and
- increasing binary values of codes are assigned to instruction opcodes that, on the average, occur in decreasing order of the frequency of occurrence

then this will accomplish the task to minimizing the checksum value. Fig. 6 illustrates the instrumentation of this approach.

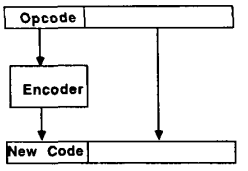

Fig. 6. Opcode reassignment.

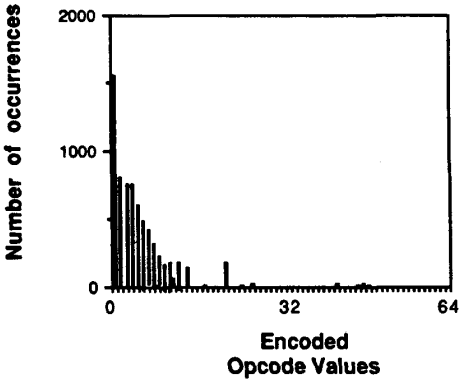

Fig. 7. Static distribution after encoding instructions.

TABLE I
BIT ERROR DETECTION LATENCY

m	without opcode reassignment	with opcode reassignment
1	9.0	5.61
2	9.0	2.89
5	8.97	1.84
9	8.81	1.79

The compiler computes the checksum based on the encoded instructions. The watchdog monitors the original instructions but encodes them and then subtracts from the previous result. Notice that this does not require any CPU design change as the original instruction set is intact. This only requires a collaboration between the watchdog and the compiler, which was the case even when no encoding was involved. Fig. 7 is a new distribution of the code represented in Fig. 5 based on the encoded instructions.

It is clear how low checksum values can be achieved based on this encoding approach. Table I illustrates the effect of reassigning opcodes on bit-error detection latency. Monte Carlo experiments were performed on an HPPA branch-free instruction block of length 10. Here the parameters were $S = 10$ and $n = 32$.

The second column in Table I is the average bit-error latency without any opcode reassignment for various values of m. The third column in Table I is the bit-error latency with opcode reassignment. It is clear from this table that latency is significantly reduced after opcode reassignment. The opcode reassignment was based on a study of static distribution of HPPA instructions as illustrated by Fig. 6. For a discussion on bit-error coverage see [3].

C. Sequence Error Detection Latency and Coverage

The model of sequence errors is best described by Fig. 8. The line segments represent the branch-instruction-free blocks. N sequence

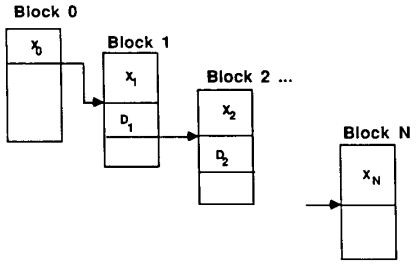

Fig. 8. Sequence error model.

errors are assumed. X_0 is a random variable which represents the number of instruction cycles from the beginning of the block where the first sequence error occurs. Subsequent X_j's represent the random landing of the sequence error into a random block such that the point of landing is X_j instructions away from the beginning of the jth block. These random blocks are in the program space and the block lengths have the probability generating function $G_b(Z)$. D_j is a random variable which represents the random number of instruction cycles from the point of landing to the point of the next sequence-error in block j. D_j is defined for $0 < j < N$. Let V_j denote the random variable that defines the length of block j.

Let us denote $X_0, \cdots, X_N$ and $D_1, \cdots, D_{N-1}$ by random variables X and D, respectively. The probability generating function for random variable X for a given block length q is given by

$$G_X(Z; q) = \frac{1}{q}(1 + Z + \cdots Z^{q-1}) = \frac{1}{q}\frac{1 - Z^q}{1 - Z}.$$

Removing the conditional probability, the probability generating function for X is

$$G_X(Z) = \sum_{q=1}^{T} \frac{b_q}{q}\frac{1 - Z^q}{1 - Z} = \sum_X P_X Z^X.$$

The probability generating function for D given that the block length is q and the point of landing is X instruction cycles away is

$$G_D(Z; q, X) = \frac{1}{q - X}\frac{1 - Z^{q-X}}{1 - Z}.$$

Removing the conditional probability, the probability generating function for D is

$$G_D(Z) = \sum_X P_X \sum_q b_q G_D(Z; q, X).$$

The sum of random variables $D_1, \cdots, D_{N-1}$, and $(V_N - X_N)$ will be the number of instruction cycles before the zero check signal. Let $V_N - X_N$ be denoted by X'_N. The probability generating function for the random variable X'_N given that the block length is V_N is

$$G_{X'}(Z; q) = \frac{1}{q}(Z + Z^2 + \cdots Z^q).$$

Removing the conditional probability, the generating function for X' is

$$G_{X'}(Z) = \sum_q b_q G_{X'}(Z; q).$$

The probability generating function for number of instruction cycles, $I(N)$, before the zero check signal can be obtained by using convolution theorem and the generating function is

$$G_{I(N)}(Z) = G_D(Z)^{N-1} G_{X'}(Z).$$

From this generating function the average instruction cycles, $I_{av}(N)$, can be found for N sequence errors. $I_{av}(N)$ is given by

$$I_{av}(N) = \left.\frac{dG_{I(N)}(Z)}{dz}\right|_{Z=1}.$$

This evaluates to

$$I_{av}(N) = \frac{(N+1)}{4} b_{av} - \frac{(N-3)}{4}.$$

Here b_{av} is the average block length and is given by

$$b_{av} = \left.\frac{dG_b(Z)}{dz}\right|_{Z=1}.$$

Clearly, $I_{av}(N)$ increases linearly with N. The sequence error coverage is given by

$$\text{sequence error coverage} = 1 - c(I_{av}(N), K_{V_0}, n)$$

where $c(I_{av}(N), K_{V_0}, n)$ is the coefficient of $Z^{K_{V_0}}$ in the probability generating function $G_i(Z)^{I_{av}(N)}$. K_{V_0} is the extended-precision checksum of block 0 in Fig. 7. As the number of sequence errors N increases the coefficient $c(I_{av}(N), K_{V_0}, n)$ falls off exponentially and therefore the sequence error coverage approaches unity. In a signature-based method, the sequence error coverage remains relatively constant at $\approx 2^{-L}$, where L is the length of the signature register, and does not improve with increasing N.

Based on the foregoing analysis, it can be shown that for signature-based control-flow checking methods, the average sequence error detection latency is the same as $I_{av}(N)$. Therefore, for signature-based methods, the latency increases linearly with N. For extended-precision checksum-based methods, the average latency $\text{Lat}_S(N, K_{V_0})$ is given by

$$\text{Lat}_S(N, K_{V_0}) = \sum_{h=1}^{I_{av}(N)} (h-1) \sum_{J \le K_{V_0}} c(h-1, J, n) \sum_{\alpha > K_{V_0} - J} c(1, \alpha, n).$$

$\text{Lat}_S(N, K_{V_0})$ is approximately $O(b_{av})$ and is relatively insensitive to N. Table II illustrates some experimental results on sequence error latency. The block length b of the first block that had sequence error was 10.

TABLE II
SEQUENCE ERROR DETECTION LATENCY

N	1	2	3	4	5	6	7	8
Latency	4.81	6.32	10.3	10.7	10.87	10.87	10.87	10.87

The second column in Table II lists the average sequence error latency for various values of N. It can be noticed that the average sequence error latency approaches $O(b)$ and becomes relatively independent of N, as N gets large.

IV. Summary

A control-flow checking architecture has been proposed in this paper. The architectural visibility of control-flow checking is an important factor in the overall system architecture. A watchdog coprocessor approach provides an architectural platform for control-flow checking without causing major perturbations in the system design.

The use of extended-precision checksums has several advantages. These advantages are in the areas of error-detection latency and error-detection coverage. The sequence error detection latency remains bounded by the average block length, and the sequence error coverage approaches unity as the number of sequence errors increase. Natural redundancy in program codes can be exploited to enhance the effectiveness of extended-precision checksums. A natural watchdog timer mechanism is possible by using extended-precision checksums.

References

[1] A. Mahmood and E. J. McCluskey, "Concurrent error detection using watchdog processor—A survey," *IEEE Trans. Comput.*, vol. C-37, no. 2, pp. 160–174, Feb. 1988.

[2] M. Schuette and J. Shen, "Processor control flow monitoring using signatured instruction streams," *IEEE Trans. Comput.*, vol. C-36, no. 3, pp. 264–276, Mar. 1987.

[3] N. R. Saxena and E. J. McCluskey, "Extended precision checksums," in *Dig. Papers 17th Annu. Int. Symp. Fault-Tolerant Comput. (FTCS)*, July 1987, pp. 142–147.

[4] T. Sridhar and S. Thatte, "Concurrent checking of program flow in VLSI processors," in *Dig. Papers 1982 IEEE Test Conf.*, Nov. 1982, pp. 191–199.

[5] M. Namjoo, "Techniques for concurrent test of VLSI processor operation," in *Dig. Papers 1982 IEEE Test Conf.*, June 1982, pp. 461–468.

[6] K. Wilken and J. Shen, "Continuous signature monitoring: Efficient concurrent-detection of processor control errors," in *Dig. Papers 1988 IEEE Test Conf.*, Sept. 1988, pp. 914–925.

[7] HP 9000/930 and HP 9000/840 Computers, *Precision Architecture and Instruction Reference Manual*, HP, Nov. 1986.

[8] G. Kane, *R2000 RISC Architecture*. Englewood Cliffs, NJ: Prentice-Hall, 1987.

[9] J. Shen and S. Tomas, "A roving monitoring processor for detection of control flow errors in multiple processor systems," *Microprocessing and Microprogramming*, vol. 20, no. 4&5, pp. 249–269, May 1987.

[10] J. C. Huck, "Comparative analysis of computer architectures," Ph.D. dissertation, Stanford Univ., Mar. 1983.

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

[12] U. Gunneflo, J. Karlsson, and J. Torin, "Evaluation of error detection schemes using fault injection by heavy-ion radiation," in *Proc. Nineteenth Int. Symp. Fault-Tolerant Comput.*, June 1989, pp. 340–347.

Optimal Diagnosis Procedures for k-out-of-n Structures

MING-FENG CHANG, WEIPING SHI, AND W. KENT FUCHS

Abstract—**This paper investigates diagnosis strategies for repairable VLSI and WSI structures based on integrated diagnosis and repair. Knowledge of the repair strategy, the probability of each unit being good, and the expected test time of each unit is used by the diagnosis algorithm to select units for testing. The general problem is described followed by an examination of a specific case. For k-out-of-n structures, we give a complete proof for the optimal diagnosis procedure proposed by Ben-Dov. A compact representation of the optimal diagnosis procedure is described, which requires $O(n^2)$ space and can be generated in $O(n^2)$ time. Simulation results are provided to show the improvement in diagnosis time over on-line repair and off-line repair.**

Index Terms—**Diagnosis, k-out-of-n, repair, VLSI, wafer probe testing, WSI.**

I. Introduction

Repairable VLSI or WSI structures may contain spare modules (units) for purposes of tolerating manufacturing defects and enhancing yield. In order to tolerate faulty units, it may be necessary to diagnose the system and apply reconfiguration strategies for repair. In the traditional approach, diagnosis and repair have been treated as separate stages: first diagnosis is performed and then the locations of the faulty units are passed to the reconfiguration algorithm [Fig. 1(a)]. An example of this approach is the diagnosis and repair algorithms for repairable memory arrays by Chang, Fuchs, and Patel [1].

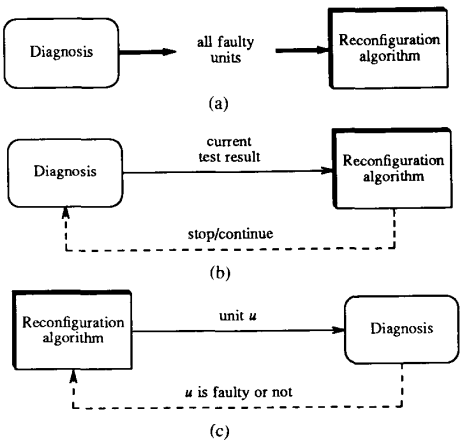

Fig. 1. Different diagnosis and reconfiguration methods. (a) Traditional off-line repair. (b) On-line repair. (c) Optimal diagnosis and repair.

However, in some applications the reconfiguration algorithm may not need complete information regarding the location of defective units to determine a repair solution. Haddad and Dahbura [2] proposed an on-line repair method which can detect unrepairable random-access memory chips during testing and hence eliminate unnecessary tests. A similar approach was proposed by Huang and Lombardi in their development of a memory repair algorithm which is executed in an on-line fashion with the diagnosis algorithm [3]. The on-line repair approach operates on partial diagnosis information, and therefore can potentially terminate the diagnosis procedure early and provide a repair solution earlier than an off-line repair [Fig. 1(b)].

In what we will call *optimal diagnosis and repair*, the reconfiguration algorithm not only works on-line with the diagnosis algorithm, but also tells the diagnosis algorithm which unit to test next [Fig. 1(c)]. The reconfiguration and diagnosis algorithms terminate when they determine a repair solution for the structure regardless of the status of the remaining units, or they determine the structure has to be discarded regardless of the status of the remaining units. In addition to exploiting the repair strategy, the diagnosis algorithm can also exploit knowledge about the expected yield of each unit and the expected test time of each unit in the structure.

Defect densities as well as distributions may vary across the wafer [4], and the expected access and test time of specific units may vary depending on the location as well as the type of unit under test. Even for identical units, the expected test time may differ because of the difference in yield. A fault-free unit has to be tested by all the test vectors, while the testing of a faulty unit is aborted when a fault is detected. Thus, the expected test time of a unit with higher yield is usually longer than that of a unit with lower yield.

In this paper, we consider structures whose units are tested individually and sequentially, such as in wafer probe testing [5]. An optimal diagnosis procedure gives the sequence in which units are tested, and minimizes the expected time for determining whether the structure is repairable or unrepairable. The specification of the structure includes the following information: 1) the structure of the system (location and type of each unit, the reconfigurable design),

Manuscript received June 21, 1989; revised November 30, 1989. This work was supported by the Semiconductor Research Corporation under Contract 88-DP-109. A brief version of this paper was presented at the 1989 IEEE International Conference on Computer Aided Design (ICCAD-89).

The authors are with the Center for Reliable and High-Performance Computing, Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, Urbana, IL 61801.

IEEE Log Number 8933876.