

## A Dynamic Continuous Signature Monitoring Technique for Reliable Microprocessors

Sugihara, Makoto

Department of Information and Computer Sciences, Toyohashi University of Technology

<https://hdl.handle.net/2324/6794518>

---

出版情報 : IEICE Transactions on Electronics. E94-C (4), pp.477-486, 2011-04. Institute of Electronics, Information and Communication Engineers

バージョン :

権利関係 :



# A Dynamic Continuous Signature Monitoring Technique for Reliable Microprocessors\*

Makoto SUGIHARA<sup>†,††</sup>, Member

**SUMMARY** Reliability issues such as a soft error and NBTI (negative bias temperature instability) have become a matter of concern as integrated circuits continue to shrink. It is getting more and more important to take reliability requirements into account even for consumer products. This paper presents a dynamic continuous signature monitoring (DCSM) technique for high reliable computer systems. The DCSM technique dynamically generates reference signatures as well as runtime ones during executing a program. The DCSM technique stores the generated signatures in a signature table, which is a small storage circuit in a microprocessor, unlike the conventional static continuous signature monitoring techniques and contributes to saving program or data memory space that stores the signatures. Our experiments showed that our DCSM technique protected 1.4–100.0% of executed instructions depending on the size of signature tables.

**key words:** Soft Error, NBTI, SEU, SET, Control Signal Error, Continuous Signature Monitoring, Reliability, Vulnerability, Microprocessor

## 1. Introduction

Reliability issues such as a soft error and NBTI (negative bias temperature instability) have become matters of concern as a transistor pattern continues to shrink [9], [11]. Design for reliability (DFR) will become more and more important for space integrated circuits (ICs) as well as consumer ICs [13]–[17]. DFR techniques which achieve high reliability, small chip area, and low performance overhead are required for designing computer systems as traditional DFR techniques such as a multiplying technique such as TMR (triple modular redundancy) are forbiddingly expensive.

A soft error and NBTI cause a control signal error which makes a computer jump to a wrong address, makes a multiplexer select a wrong input, or makes an ALU do a wrong operation. A continuous signature monitoring (CSM) technique is the one which detects a control signal error during executing a program [18]. A CSM technique exploits a signature value onto which an invariable sequence of control signals such as that of a program counter is mapped. A typical CSM technique embeds a *reference signature*, which is precomputed for such an invariable sequence of control signals on compiling a program, into the program. A signature

generator on a chip computes a *runtime signature*, which is a signature computed during executing the program. The comparison of a runtime signature with a reference signature exhibits whether or not a control signal error occurred. Figure 1 shows an example of simple continuous signature monitoring. In this example, a program is partitioned into basic blocks [2] each of which has a single entry point. A signature function  $V$  maps an invariable sequence of control signals, that is a binary encoding of a consecutive stream of control signals for a basic block, to a signature. A linear feed-back shift register is often used as a circuit implementing a signature function [1]. A reference signature is conventionally generated and embedded into a program on compiling it. A signature instruction, which consists of opcode and signature parts, is added to the end of each basic block. A runtime signature is generated by a hardware signature generator during executing the basic block. The reference and runtime signatures are compared in order to check whether or not a control signal error has occurred when a microprocessor encounters a signature instruction. A control signal error is detected if there exists inconsistency between the reference and runtime signatures.

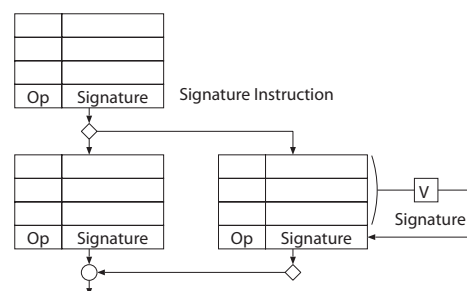


Fig. 1 An example of simple control flow checking.

CSM techniques have been studied since the 1980s [8], [10], [12], [18]. These techniques are to detect a *control flow error*, that is an error causing to jump from an address to a wrong address. The memory overhead and detection rate of control flow errors are major measures in evaluating the static CSM techniques. Namjoo's, Shen's and Wilken's approaches increase 12–21%, 6–15% and 4–11% of memory space respectively and also detect 99.5–99.9%, 85–93% and 99.9999% of control flow errors. Any of the approaches involves high memory overheads and performance overheads. Embedding signatures into a program sets a limit to

Manuscript received August 12, 2010.

Manuscript revised November 7, 2010.

<sup>†</sup>The author is with the Department of Computer Science and Engineering, Toyohashi University of Technology, Aichi 441-8580, Japan.

<sup>††</sup>This work is funded by the Japan Science and Technology Agency(JST), CREST.

\*An abridged version of this paper was presented at the EUROMICRO Conference on Digital System Design, Lille, France, September 2010.

the width of the signatures because of memory overheads. This means that the number of control signals for which a signature is generated must be low in order to avoid increasing memory space. The conventional techniques also make a program to be incompatible between microprocessors because the approaches entail an incompatible hardware mechanism and a modified program. The program compatibility problem becomes critical in reusing software as intellectual property.

This paper proposes a dynamic continuous signature monitoring (DCSM) technique for detecting control signal errors in a microprocessor. Our DCSM technique computes both reference and runtime signatures during executing a program, stores the reference signatures in a signature table (signature tables), and compares reference and runtime signatures to find whether or not a control signal error occurred. It is necessary that an instruction is executed twice or more to generate both reference and runtime signatures and compare them for detecting a control signal error. The principle of locality [4] suggests that a part of a program is executed frequently and that a signature table need store a reference signature of the part of the program temporarily. The principle helps the size of a signature table to be relatively smaller than the size of all reference signatures which the conventional techniques store in instruction and data memory. Unlike the conventional techniques, our technique hardly requires memory space for storing reference signatures. A small signature table is provided to store reference signatures instead. Our technique requires a constant size of a signature table while the conventional approaches require the memory overhead whose size is linear to that of a program. This paper proposes a DCSM technique as well as investigates how many instructions are protected from control signal errors by our DCSM technique.

The remainder of this paper is organized as follows: Section 2 reviews conventional continuous signature monitoring techniques. Section 3 proposes a dynamic continuous signature monitoring technique which achieves high reliability with a small chip area. Section 4 quantitatively evaluates how many instructions our DCSM technique covers and protects from control signal errors. Section 5 summarizes this paper and provides concluding remarks.

## 2. Control Flow Error and Continuous Signature Monitoring

This section reviews a control flow error and static continuous signature monitoring techniques [8], [10], [18].

### 2.1 Control Flow Error

A soft error or NBTI unexpectedly and momentarily destabilizes the value of a signal line and possibly causes a faulty behavior of a computer system. Such an undesirable event is modeled to cause a computer system to jump to a wrong address. We call this fault behavior a *control flow error*.

A program can be represented by a *program graph*, a

directed graph that represents each block of a program by a node and all legal transitions between blocks, which are specified in a program, by an arc. Figure 2 shows an example program graph. A control flow error is modeled as an error which causes a transition between the nodes between which there are no legal arc.

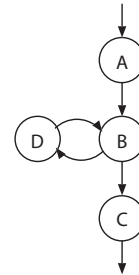


Fig. 2 An example program graph.

### 2.2 Static Continuous Signature Monitoring

The comparison between expected and occurred values at all control signal lines is ideal for assuring correct behavior of a microprocessor. Such a comparison, however, is unrealistic because it is impractical to observe billions of control signals and prepare expect values for them.

A microprocessor generates a constant sequence of control signals such as a program counter and control signals for a datapath as far as the microprocessor runs correctly. A constant sequence of control signals can be mapped to a constant value with a mapping function which can be implemented with a data compression circuit such as a linear feed-back register (LFSR). We define a *signature* as a value to which a sequence of signals is mapped by any mapping function. A signature is expected to be correct unless the corresponding signal lines are destabilized. The correct execution of a microprocessor can be verified by comparing a signature generated during executing a program with the expected value for the signature. We call the expected value for a signature a *reference signature* and a signature computed during program execution a *runtime signature*.

The conventional CSM techniques compute a reference signature for a sequence of control signals on compiling a program. Reference signatures are often embedded into a program code [8], [10], [18]. A *signature instruction*, which compares reference and runtime signatures, is also embedded into a program code. A signature instruction often consists of an opcode and a reference signature. A runtime signature is generated during executing a program. Once the microprocessor encounters a signature instruction, it compares reference and runtime signatures to detect a control flow error. The conventional approaches that embed signature instructions into a program code entail memory and performance overheads because of embedded signature instructions and signatures.

We overview the continuous signature monitoring theory generalized by Wilken [18]. We define a sequence of executed instructions as a *path*. A path consists of  $N$  instructions. Each instruction on the path is correlated with an *intermediate signature*, which is intermediately obtained to compute a signature for the path. Signature  $S_i$  which is correlated with the  $i$ -th ( $1 \leq i \leq N$ ) instruction is calculated as follows:

$$S_i = V(S_{i-1}, W_{i-1}), 1 \leq i \leq N, \quad (1)$$

where  $V$  is a signature function,  $S_0$  is an initial signature, and  $W_{i-1}$  is a set of signal values correlated with  $(i-1)$ -th instruction. Instruction  $i$  is correlated with Signature  $S_i$ . Signature  $S_i$  is the signature which correlates with the subsequence  $[0, i-1]$ .

The value of an intermediate signature determines whether or not a control flow error is detectable [18]. Assume that the correct behavior of a program causes an execution of Instruction D after Instruction C. A control flow error, the wrong behavior of a program, causes an execution of Instruction D\* after Instruction C. A control flow error is detectable by comparing runtime and reference signatures if  $S_D \neq S_{D^*}$ . A control flow error is not detectable if  $S_D = S_{D^*}$ . The coincidence of the two signatures causes the signature calculations done after Instruction D\* to coincide with those for the correct path.

The number of undetected control flow errors can be estimated by intermediate signatures. Assume that an intermediate signature is given to every instruction. Let a set of instructions whose intermediate signature is  $S$  be  $D_S$ . Let a set of instructions whose next instruction belongs to  $D_S$  be  $C_S$ . We assume that a control flow error occurs at any instruction at the same probability and causes an incorrect transition to any instruction. The sizes of  $D_S$  and  $C_S$  are given by  $d_s$  and  $c_s$  respectively. The number of all instructions is  $m$ . The fraction of undetected control flow errors which start at any instruction in  $C_S$  is shown as

$$(d_S - 1)/(m - 1) \approx (d_S - 1)/m. \quad (2)$$

If any instruction is executed at the same frequency, the fraction that a control flow error occurs in an instruction of  $C_S$  is shown as  $c_S/m$ . The fraction of undetected control flow errors is shown as follows:

$$e = \sum_i (d_S - 1) c_S / m^2 \quad (3)$$

Note that the above calculation is an approximate calculation. The 90/10 rule suggests that the 10% of a program code occupies 90% of its execution time [4]. This means that some instructions are executed frequently and the others not. We assume that any instruction is executed at the same frequency for a simple explanation.

The number of undetected control flow errors becomes minimal if the values of intermediate signatures distribute uniformly. Correlation between intermediate signatures enlarges the size of a set of the instructions which have the

same intermediate signature, and increases the number of undetected control flow errors. A naive CSM technique adopts the same initial signature for all paths. The adoption of the same initial signatures for all paths causes the first instructions of all the paths to belong to  $D_{S_0}$  and the last instruction of all the paths to belong to  $C_{S_0}$ . Transition from an instruction in  $C_{S_0}$  to any but a correct instruction in  $D_{S_0}$  is not recognized as a control flow error even if a signature is examined.

The lower bound of the fraction of undetected control flow errors is given by the number of undetected control flow errors attributed to the initial signature. Let the average path length be  $p$ . The fraction of the instructions which belong to  $D_{S_0}$  is given by  $1/p$  and the fraction of the instructions which belong to  $C_{S_0}$  is also given by  $1/p$ . As the number of paths is enough high, the fraction of the undetected control flow errors attributed to  $S_0$  is given by

$$e_{S_0} \approx p^{-2}. \quad (4)$$

Several researches reported that the average path length is from 4 to 10 [3], [6]–[8], [10]. Assuming that a signature instruction is added to the end of each and every path, the average path length becomes 5 to 11. From Equation (4), the fraction of undetected control flow errors becomes 1-4%. This control flow errors at the end of all paths are detected by 96-99%. Assume that uniformly random values are utilized for  $v$ -bit intermediate signatures. A control flow errors causes the next signature to be one of  $2^v$  signatures including a correct one. The utilization of  $v$ -bit intermediate signatures causes the fraction  $1 - 2^{-v}$  of possibly detectable control flow errors to be undetected.

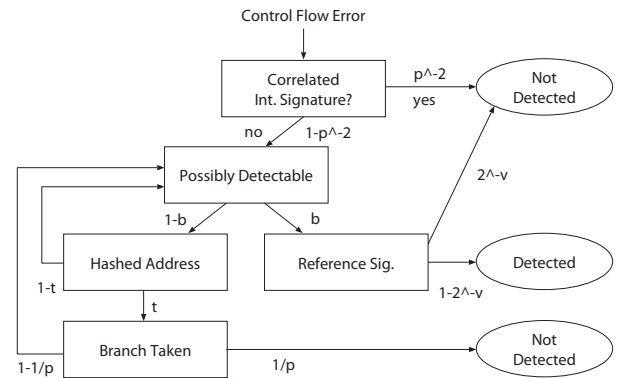


Fig. 3 The Markov model for control flow errors.

Figure 3 shows the Markov model for control flow errors. The adoption of the identical initial signature for all paths makes  $p^{-2}$  of all the control flow errors undetectable. The remaining control flow errors are probably detectable. Let the fraction of the paths which end with a signature instruction be  $b$ . When a path ends with a signature instruction, the signature instruction is executed to compare reference and runtime signatures. The probability that a control flow error happens to make the runtime signature identical to

the correct one is  $2^{-v}$ . The probability that a control flow error happens to make the runtime signature distinct from the correct one is  $1 - 2^{-v}$ . The fraction of the paths which end with a jump/branch instruction is  $1 - b$ . The jump/branch instruction is applied the BAH (branch address hashing) [12] to. The BAHed instruction retains the value of the destination address XORed with the intermediate signature. The hashed address is XORed with the current intermediate signature on executing the instruction. A correct intermediate signature correctly restores the destination address while an incorrect intermediate signature incorrectly restores the destination address. Let the fraction that the branch is taken be  $t$ . The fraction  $1 - t$  of the BAHed branches is not taken and a control flow error is expected to be probable detectable. A control flow error causes to jump to the beginning of a path at the probability of  $1/p$  and results in being undetectable. Otherwise, a control flow error is probably detectable with a signature instruction which probably appears later.

### 3. Dynamic Continuous Signature Monitoring Technique

This section presents a dynamic continuous signature monitoring (DCSM) technique which dynamically generates runtime signatures as well as reference ones and compares them in order to detect a control signal error during executing a program for detecting a control signal error.

#### 3.1 The Overview of Dynamic Continuous Signature Checking

The conventional static CSM techniques embed signatures and signature instructions into a program/data memory on compiling a program. Embedding signatures and signature instructions into program/data memory causes the size of program/data memory to increase. The execution of the embedded signature instructions also causes the execution time of a program to increase. The program code into which signature instructions are embedded loses program compatibility with static CSM-incompliant microprocessors.

In contrast, our DCSM technique which we propose in this paper computes a reference signature for a part of a program during executing a program and stores the reference signature into a signature table, which is a small storage circuit in a microprocessor. Our DCSM technique

stores reference signatures in a small signature table instead of program/data memory. The first execution of a part of a program generates a reference signature and stores it into a signature table. The next execution of the program generates a runtime signature to compare with the corresponding reference signature if the reference signature exists in the signature table. The runtime signature is registered into the signature table as a reference signature if the corresponding reference signature does not exist. A control flow error is not detected if a part of a program is executed only once. Our DCSM technique passes the instructions which are executed only once. The DCSM-compliant microprocessor automatically compares the reference and runtime signatures to determine whether or not a control signal error occurred when both reference and runtime signatures are available for the part of a program. No signature instructions are necessary for the DCSM-compliant microprocessors to detect control signal errors. The DCSM technique does not lose program compatibility at all between DCSM-compliant and DCSM-incompliant microprocessors.

Figure 4 gives an overview of a DCSM hardware mechanism. The DCSM hardware typically consists of three parts: a signature generator, path analyzer, and signature table. A path analyzer and signature table are peculiar to the DCSM technique. These hardware components cause chip area overhead and should be designed carefully. A path analyzer detects a path, an instruction stream for which a signature is generated, with the value of the program counter and the instruction pointed by the program counter. A path is defined as a unique instruction stream specified by a pair of start and end addresses. The signature table retains entries each of which consists of the beginning and end addresses, a reference signature and an initialization signature. The path analyzer looks the corresponding entry up in the signature table once the path analyzer detects the beginning of a path. All the paths which start at an identical beginning address have the same initial signature. The signature generator is initialized with the initial signature of the path if the corresponding entry is found. The signature generator is initialized with the current value of the signature generator otherwise. The path analyzer looks the corresponding entry up when reaching the end of a path. The runtime and reference signatures are compared to determine whether or not a control signal error occurred if the corresponding entry exists in the signature table.

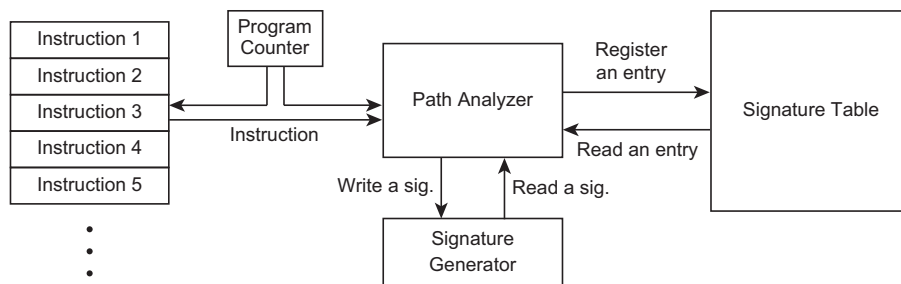


Fig. 4 An overview of dynamic control-flow checking hardware.

### 3.2 Path Detection for Dynamic Continuous Signature Monitoring

This subsection gives an example of path detection for dynamic continuous signature monitoring. The DCSM technique requires automatically detecting a path for which the DCSM technique generates reference and runtime signatures. The conventional static CSM techniques may spend a long computation time on compiling a program in finding the optimal paths for a program such that they minimize the memory size or the number of executions for signature instructions. In contrast, the DCSM technique cannot afford to spend such a long computation time because a long computation time degrades the performance of a microprocessor. This paper focuses on not optimizing a set of paths but detecting them with a simple hardware mechanism. We categorize paths into two kinds. The first kind is for the paths in which the addresses of an instruction stream is consecutive. The paths of the first kind satisfy the requirement that a path is a unique instruction stream under the condition that any of conditioned jump and branch instructions is *not taken*. The second kind is for the paths which commence with a conditioned jump or branch instruction and finish with the destination of the conditioned jump or branch instruction. The paths of the second kind also satisfy the path requirement under the condition that the corresponding conditioned jump or branch instruction is always *taken*.

The summarized procedure for detecting the paths of the first kind is shown as follows.

- Regard the beginning address of a program as the beginning of a path.
- Regard the destination instruction of a conditioned jump or branch instruction as the beginning of a path.
- Regard a taken jump or branch instruction as the end of a path if a microprocessor does not adopt delay slots.
- Regard the end of instructions within its delay slots as the end of a path if a microprocessor adopts delay slots.

The summarized procedure for detecting the paths of the second kind is shown as follows.

- Regard a conditioned jump or branch instruction as the beginning of a path if the instruction is taken.
- Regard the destination instruction of the above conditioned jump or branch instruction as the end of the path.

The above two procedures to detect paths are just examples. The unique instruction stream which appears before the taken jump or branch instruction may be included in the path of the second. The unique stream which appears after the destination instruction of the taken jump or branch may also be included in the path of the second. These paths also satisfy the path requirement under the condition that the corresponding conditioned jump or branch is taken. The above procedures are summarized in Figure 5.

#### Dynamic continuous signature monitoring algorithm

##### Procedure DCSM

**Input**  $C$ : the given program code.

**Input**  $\mathcal{D}_{\text{given}}$ : the number of delay slots of a microprocessor.

**Variable**  $\mathcal{I}$ : the instruction currently being executed.

**Variable**  $\mathcal{I}_{B1}$ : the beginning instruction of a path of the first kind.

**Variable**  $\mathcal{I}_{B2}$ : the beginning instruction of a path of the second kind.

**Variable**  $\mathcal{I}_{E1}$ : the end instruction of a path of the first kind.

**Variable**  $\mathcal{I}_{E2}$ : the end instruction of a path of the second kind.

**Variable**  $\mathcal{S}_{R1}, \mathcal{S}_{R2}$ : the runtime signatures for the first and second kinds.

**Variable**  $\mathcal{S}_{I1}, \mathcal{S}_{I2}$ : the init. signatures for the first and second kinds.

**Variable**  $\mathcal{P}_1, \mathcal{P}_2$ : the current paths for the first and second kinds. Given by a pair of beginning and end addresses.

**Variable**  $\mathcal{D}$ : the remaining delay slots.

**Variable** *jump\_flag*: the flag which indicates the PC has just jumped.

**begin**

*/\* Initialize the variables. \*/*

Set  $\mathcal{I}$  to the first instrn. of the program  $C$ .

Initialize  $\mathcal{S}_{I1}$  and  $\mathcal{S}_{I2}$ .

$\mathcal{D} := 0$ ;  $\mathcal{I}_{B1} := \mathcal{I}$ ; *jump\_flag* := false;

**repeat**

*/\* A path of the second kind is registered just after jump/branch \*/*

**if** *jump\_flag* == true **then**

$\mathcal{I}_{E2} := \mathcal{I}$ ;  $\mathcal{P}_2 := (\mathcal{I}_{B2}, \mathcal{I}_{E2})$ ;

**if** the sig. of path  $\mathcal{P}_2$  exists in the sig. table  $\mathcal{T}_2$  **then**

Compare the ref. and runtime sigs.

**else**

Register path  $\mathcal{P}_2$ , its init. sig., its ref. sig. to the sig. table  $\mathcal{T}_2$ .

**end if**

**if** there exists a path whose beginning instrn. is  $\mathcal{I}$  **then**

Set  $\mathcal{S}_{I1}$  to the init. sig. of the path.

**end if**

*jump\_flag* := false;  $\mathcal{I}_{B1} := \mathcal{I}$ ;

**else** */\* jump\_flag := false \*/*

*/\* For the instrn. within the delay slots. \*/*

**if**  $\mathcal{D} > 0$  **then** */\* Currently within delay slots \*/*

$\mathcal{D} - 1$ ;

*/\* A path of the first kind is registered just before jumping. \*/*

**if**  $\mathcal{D} == 0$  **then**

**if** the conditioned jump/branch is taken **then**

*/\* for registering a path of the second kind. \*/*

*jump\_flag* := true;

**if** there exists a path whose beginning instrn. is  $\mathcal{I}$  **then**

Set  $\mathcal{S}_{I2}$  to the init. sig. of the path.

**end if**

*/\* Regard the current instrn. as the end of a path. \*/*

*/\* of the first kind. \*/*

$\mathcal{I}_{E1} := \mathcal{I}$ ;  $\mathcal{P}_1 := (\mathcal{I}_{B1}, \mathcal{I}_{E1})$ ;

**if** the sig. of path  $\mathcal{P}_1$  exists in the sig. table  $\mathcal{T}_1$  **then**

Compare the ref. and runtime sigs.

**else**

Register path  $\mathcal{P}_1$ , its init. sig., its ref. sig. to the

sig. table  $\mathcal{T}_1$ .

**end if**

**end if**

**end if**

**end if**

**if** the instrn.  $\mathcal{I}$  is a conditioned jump/branch **then**

*/\* The beginning instrn. for the path of the second kind. \*/*

Set  $\mathcal{I}_{B2}$  to the current instrn.  $\mathcal{I}$ .

*/\* Set the variable  $\mathcal{D}$  for counting delay slots down. \*/*

$\mathcal{D} := \mathcal{D}_{\text{given}}$ ;

**end if**

Set variable  $\mathcal{I}$  to the next instrn.

Compute new runtime sigs. with the current runtime sigs.

and control signals and set them to  $\mathcal{S}_{R1}$  and  $\mathcal{S}_{R2}$ .

**until** the given program  $C$  finishes

**end**

**Fig. 5** Pseudo-code for dynamic continuous signature monitoring.



#### 4. Experiment

This section provides experimental evaluation and discussion on the DCSM technique. Subsection 4.1 explains the program trace, simulator and signature table which we utilized for experiments. Subsection 4.2 details the metric for experiments and shows experimental results on four benchmark programs and the different sizes of a signature table. We discuss our DCSM technique in Subsection 4.3.

##### 4.1 Experimental Setup

We developed a trace-driven simulator in C++ which simulates the behavior of a DCSM-compliant CPU. We generated program traces with Imperas OVPsim, the Imperas instruction set simulator [5], and used them as the input to our trace-driven simulator. We excluded the program trace of system calls because of a restriction of OVPsim. We used two signature tables: one for consecutively executed instruction streams (paths of the first kind), and the other for jump and branch instruction streams (paths of the second kind). We adopted the set associative mechanism as the storing mechanism of the signature tables. We also adopted the LRU policy as the line replacement policy of the signature tables. We experimented on the numbers of sets and ways as shown in Table 1. The number of entries is calculated by the product of the number of sets and that of ways. We utilized four programs, dhrystone, fibonacci, linpack, and peakSpeed1 as benchmark programs. Table 2 shows the number of instruction executions and that of instructions for each benchmark program.

**Table 1** Parameters for a signature table.

# Sets	1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1K
# Ways	1, 2, 4, 8, 16, 32, 64
# Entries	1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1K, 2K, 4K, 8K, 32K, 64K

**Table 2** Benchmark programs.

	dhrystone	fibonacci	linpack	peakSpeed1
# Instrn executions [M executions]	1,012	747	1,704	3,440
# Instrns	6,944	5,730	10,770	5,778

##### 4.2 Experimental Results

We introduce the instruction coverage  $C_{insts}$  as the reliability measure for our DCSM technique as follows.

$$C_{insts} = \frac{INST_{covered}}{INST_{all}} \times 100, \quad (5)$$

where  $INST_{covered}$  is the number of instruction executions protected by the DCSM technique, and  $INST_{all}$  is the number of all instruction executions.

Table 3 shows the instruction coverages for consecutive instruction streams (paths of the first kind) and jump/branch instruction streams (paths of the second kind) of the four benchmark programs. The instruction coverages were examined by simulating the DCSM-compliant microprocessor on the different numbers of sets and ways. Table 3 shows that the fraction of instruction executions for consecutively executed instruction streams amounts to a large portion of the total of instruction executions. Control signal errors in jump/branch instruction streams are not negligible as the fraction of them amounts to more than 10% of instruction coverage at most.

**Table 3** Instruction coverage.

	dhrystone	fibonacci	linpack	peakSpeed1
$C_{insts}$ (consecutive instrns)	39.1-91.3	22.2-88.0	1.2-93.5	97.7-97.7
$C_{insts}$ (jump/branch)	2.6-8.7	1.7-12.0	0.2-6.5	2.3-2.3
$C_{insts}$ (all)	41.8-100.0	23.9-100.0	1.4-100.0	100.0-100.0

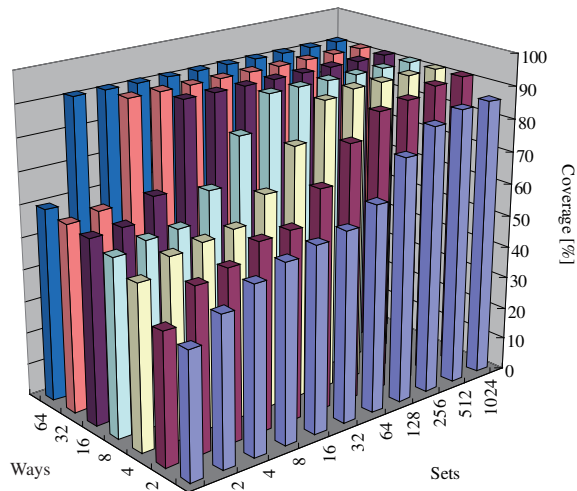
Figures 6-11 show the instruction coverage for dhrystone, fibonacci and linpack on 77 configurations of signature tables. Each configuration of a signature table is different from others regarding the numbers of sets and ways. The figures for peakSpeed1 is omitted because the DCSM technique achieved almost 100.0% of the instruction coverage for peakSpeed1 with any signature table parameter. The instruction coverages generally increase as the size of a signature table increases.

Table 4 shows the number and ratio of instructions sorted by execution count. Instructions are sorted threefold: (i) Never executed, (ii) Executed once, and (iii) Executed twice or more. No errors were assumed for the instructions sorted in the item (i) because the instructions were never executed. The DCSM technique is incapable of detecting errors being occurred on executing instructions sorted in the item (ii) because it generates only a reference signature. The DCSM technique is expected to detect errors being occurred on executing instructions sorted in the item (iii) because it ideally generates reference and runtime signatures for an instruction stream and compares them for detecting an error.

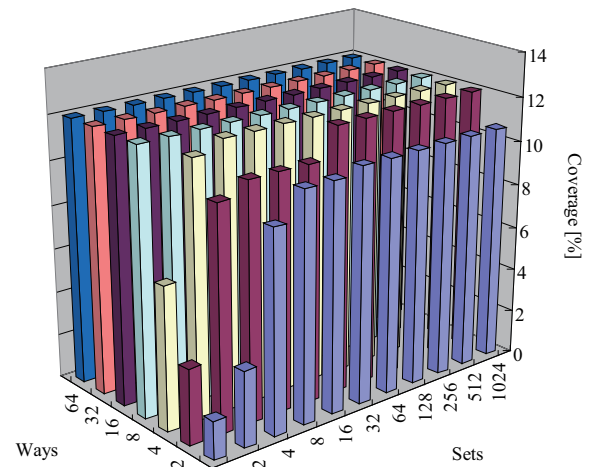
**Table 4** The number of instructions sorted by execution count.

	dhrystone	fibonacci	linpack	peakSpeed1
Never executed	5,011 (72.2%)	4,908 (85.7%)	5,753 (53.4%)	4,915 (85.1%)
Executed once	661 (9.5%)	293 (5.1%)	1,335 (12.4%)	653 (11.3%)
Executed twice or more	1,272 (18.3%)	529 (9.2%)	3,682 (34.2%)	210 (3.6%)
Total	6,944	5,730	10,770	5,778

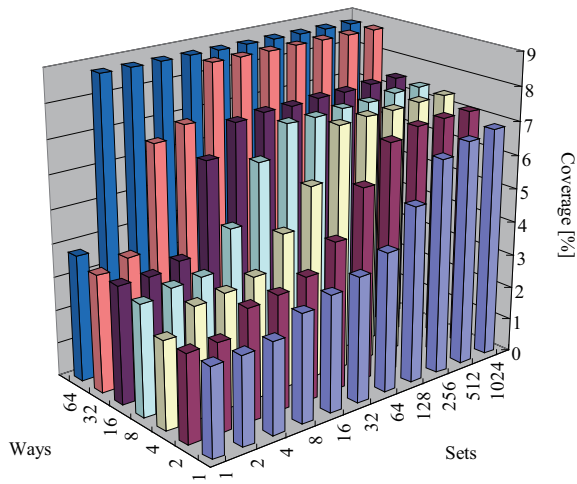
Table 5 shows the number and ratio of unprotected instruction executions sorted by two factors: (a) the onetime



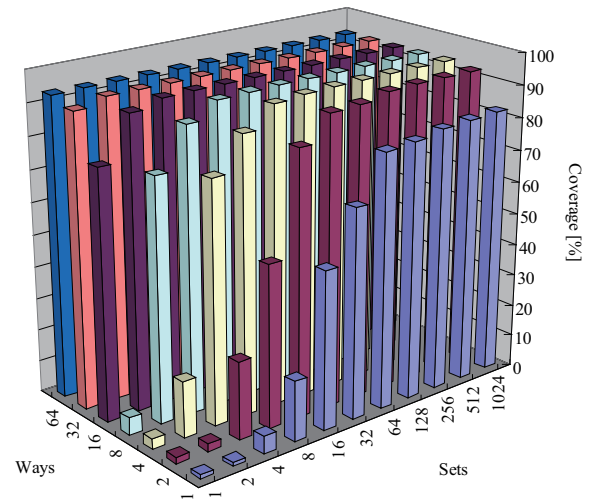
**Fig. 6** Instruction coverage for consecutive instructions (dhrystone).



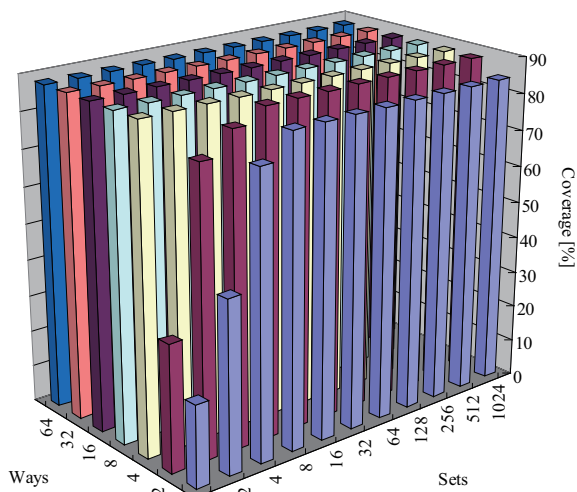
**Fig. 9** Instruction coverage for jump/branch (fibonacci).



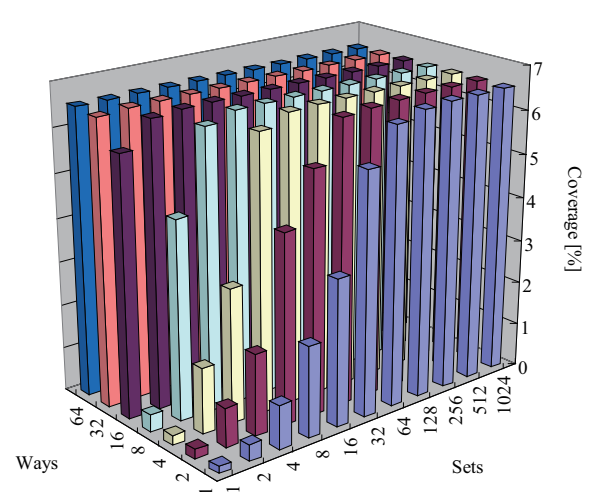
**Fig. 7** Instruction coverage for jump/branch (dhrystone).



**Fig. 10** Instruction coverage for consecutive instructions (linpack).



**Fig. 8** Instruction coverage for consecutive instructions (fibonacci).



**Fig. 11** Instruction coverage for jump/branch (linpack).



instruction execution, and (b) the incapability of signature tables to keep storing reference signatures. The table shows that the number of unprotected instruction executions due to the item (a) is lower than that due to the item (b) in all benchmark programs. The size and the set associative mechanism of signature tables are the major factor to lower the instruction coverage.

**Table 5** The number of unprotected instruction executions.

		dhrystone	fibonacci	linpack	peakSpeed1
(a)		661	293	1,335	653
		653.4 ppb	392.5 ppb	783.2 ppb	189.8 ppb
(b)	min	926	447	2,956	867
		915.4 ppb	598.7 ppb	1734.3 ppb	252.0 ppb
	max	589 M	568 M	1,681 M	1,337
		58.23%	76.09%	98.64%	388.7 ppb
# Total executions		1,012 M	747 M	1,704 M	3,440 M

### 4.3 Discussion

#### 4.3.1 Instruction Coverage

The instruction coverage tends to be high in the programs whose execution time is occupied by a lot of loops. The instruction coverages of consecutively executed instruction streams in dhrystone and fibonacci (91.3% and 88.0% respectively) became lower than linpack and peakSpeed1 (93.5% and 97.7% respectively) because dhrystone and fibonacci presumably executed the specific part of them less frequently than linpack and peakSpeed1 did. The instruction coverages of jump/branch instructions streams show a similar tendency to consecutively executed instruction streams.

The static signature techniques like Namjoo's, Shen's, and Wilken's techniques [8], [10], [18] basically achieve 100% of the instruction coverage because they statically embed reference signatures for all instruction streams into a program at the expense of program memory space. Our technique is expected to achieve a high instruction coverage close to 100.0% with two large signature tables. Our technique, however, has a definite drawback in covering all executed instruction streams because reference and runtime signatures are not compared for some instruction streams which are executed only one time, i.e. an initializing instruction sequence. Control signal errors on executing such instruction streams seldom occur as shown in Table 5.

#### 4.3.2 Memory Overheads

The DCSM technique causes no program memory overheads unlike the conventional static signature ones. Table 6 shows the program memory overheads of a basic technique, PSA [8], SIS [10], CSM [18] and DCSM. The basic technique initializes a signature in the signature generator just before the first instruction of every basic block is executed, completes generating a runtime signature just after executing the last instruction of the basic block, and compares the

**Table 6** Estimated program memory overhead.

Basic	PSA	SIS	CSM	DCSM
10-25%	12-21%	6-15%	4-11%	0%

runtime signature with the reference one which is embedded with a signature instruction just after the instruction stream in the program memory. We assumed the program which was given by Wilken [18] for Table 6. The program memory overheads for the basic technique, PSA, SIS, and CSM were originally presented by Wilken [18] and that for the DCSM technique is added in the table. Table 6 shows that the conventional CSM technique causes 4-11% of memory overheads while the DCSM technique causes no program memory overheads. We emphasize that the conventional static techniques cause program memory overheads linear to the size of a program because the number of reference signatures and signature instructions is linear to the size of a program. In contrast, the DCSM technique requires a constant size of a signature table because it exploits temporal locality of instructions for storing signatures. The DCSM technique efficiently stores and discards reference signatures. The size of signature tables is the most dominant factor in the chip area overheads of the DCSM technique.

The DCSM technique becomes more and more beneficial as the size of a program becomes larger. We show an example in which the overheads in chip area and performance are compared between the DCSM and conventional techniques. We assume two signature tables for paths of the first and second kinds as shown in Table 7. We further assume that an entry consists of four words: a beginning address, an end address, an initialization signature, and a reference signature. It takes 16 bytes to store an entry in a signature table if a 32-bit signature is utilized for a 32-bit CPU. It costs chip area for a 4K-byte storage to implement the two signature table. On the other hand, it takes the program memory overheads shown in Table 8 to adopt the static CSM technique for the four benchmark programs assuming that its memory overhead is 4-11%. Table 8 shows that the DCSM technique may be beneficial only to executing a single program for *linpack* regarding the memory overheads with the number of unprotected instruction executions increased. What if all the four benchmark programs are executed in the system? The DCSM technique definitely becomes more beneficial than the conventional CSM one regarding the program memory overheads with about 37 ppm of instruction executions unprotected as shown in the last column of Table 8.

**Table 7** An example of signature tables.

	# Sets	# Ways
Sig. table for paths of the first kind	2	64
Sig. table For paths of the second kind	2	64

The DCSM technique has an advantage in sizing the width of signatures. The exploitation of a large width of

**Table 8** Chip area overheads in bytes and unprotected instruction executions in the CSM and DCSM techniques.

		dhystone	fibonacci	linpack	peakSpeed1	all programs
Chip area overhead [Bytes]	CSM	1,111 – 3,055	916 – 2,521	1,723 – 4738	924 – 2,542	4,674 – 12,856
	DCSM	4,096	4,096	4,096	4,096	4,096
# Unprotected executions	CSM	0	0	0	0	0
	DCSM	1,058	447	250,907	867	253,279
Unprotected executions ratio [ppb]	CSM	0	0	0	0	0
	DCSM	1,045.9	598.7	147,2052	252.0	36,693.2

signatures generally causes many control signals to be monitored and is expected to find many errors on control signal lines. It also causes high program memory overheads and performance degradation in the conventional techniques. The conventional techniques fold a signature into words if the signature is larger than the width of a word. Folding a signature into words causes to increase the overheads in not only a program memory size but also instruction fetch time. Signals from the program counter are usually chosen as monitored control signals in the conventional techniques due to the limitation of the overheads in both program memory size and runtime. In contrast, the exploitation of a large width of signatures causes to increase the size of signature tables in the DCSM technique. It does not cause performance degradation as far as an access to an entry finishes within given clock cycles. The overheads in the size of signature tables are linear to the width of signatures. The DCSM technique is, however, easier to adopt than the conventional ones because the chip area overheads in signature tables for the DCSM technique are much lower than the program memory overheads for the conventional ones. As the size of a program and the width of signatures are  $N$  and  $M$  respectively, the chip area overheads for the conventional techniques are  $O(NM)$  and those for the DCSM one is  $O(M)$ .

#### 4.3.3 Performance Overheads

The DCSM technique does not have overheads in instruction fetches while the existing techniques have ones. The conventional CSM technique embeds justifying-signature instructions into a program memory [18]. We introduce two assumptions: (i) All the memory overheads in the conventional CSM technique are attributed to justifying-signature instructions. (ii) Justifying-signature instructions are uniformly executed. The two assumptions and Table 6 imply that the conventional CSM technique increases 4-11% of instruction executions that result in performance degradation. In contrast, the DCSM technique does not use any signature instruction at all and does not increase instruction executions. The performance degradation is the major drawback in the existing techniques.

## 5. Conclusion

This paper proposed a dynamic continuous signature mon-

itoring technique for detecting a control signal error at the expense of small chip area. The DCSM technique holds program compatibility between microprocessors as the DCSM technique embeds no signature instructions unlike conventional ones. The DCSM technique requires no space to store reference signatures in a program memory. A signature table is provided to store reference signatures instead. The principle of locality suggests that a signature table need store a reference signature for a part of a program for a short time. The principle of locality helps the size of a signature table to be relatively smaller than the size of all reference signatures which conventional techniques store in a program memory. We presented an instance of reference signature generations and examined how many instruction executions it covers to protect from control signal errors. Our experimental results showed that the usage of two 64K-entry signature tables protected about 100.0% of instructions. Even a 128-entry signature table achieves as nearly high as a 64K-entry signature table. System designers should determine the size of a signature table considering the trade-off between reliability and chip cost.

The DCSM technique may not cover detection of errors on control signal lines which are out of signaturing. It does not cover detection of errors on data signal lines either because a value on a data signal line is transitory and is not temporally redundant. We think that an error on a control signal line is more serious than that on a data signal line from the following reason. An error on a control signal line presumably generates a wrong computation result which has a high probability to be used by the subsequent instruction executions. In contrast, it depends on a program that an erroneous value on a data signal line is used by the subsequent instruction executions. The erroneous value would often be unused or masked by overwriting it with another value. We think that circuitry for data calculation should be made spatially redundant for the products for which system designers want extremely high reliability.

Future work includes a compilation technique which explicitly makes onetime instruction executions temporally redundant for increasing the instruction coverage. A combined static and dynamic CSM technique should also be studied for higher reliability at the expense of increasing the memory size and losing program compatibility.

## Acknowledgment

We would like to express our gratitude to the Japan Science and Technology Agency (JST), CREST for funding and supporting our research.

## References

- [1] V. D. Agrawal, C. R. Kime, and K. K. Saluja, "A tutorial on built-in self-test," *IEEE Design and Test of Computers*, Vol. 10, Issue 1, pp. 73-82, March 1993.
- [2] A. Aho, M. Lam, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison Wesley, October 2006.
- [3] J. DeRosa and H. Levy, "An evaluation of branch architectures," *Proc. IEEE 14th Comp. Arch.*, pp. 10-16, 1987.
- [4] J. L. Hennessy and D. A. Patterson, *Computer Architecture*, p. 38, Morgan Kaufmann, 2007.
- [5] Imperas Ltd., OVPsim and Imperas CpuManager user guide, 2009.
- [6] M. Kobayashi, "Dynamic profile of instruction sequences for the IBM system/370," *IEEE Transactions on Computers*, Vol. C-32, pp. 859-861, September 1983.
- [7] A. Mahmood and E. McCluskey, "Watchdog processors: error coverage and overhead," *Proc. IEEE 15th FTCS*, pp. 214-219, 1985.
- [8] M. Namjoo, "Techniques for testing of VLSI processor operation," *Proc. IEEE International Test Conference*, pp. 461-468, 1982.
- [9] D. K. Schroder and J. A. Babcock, "Negative bias temperature instability: road to cross in deep submicron silicon semiconductor manufacturing," *Journal of Applied Physics*, Vol. 94, pp. 1-18, July 2003.
- [10] M. Schuette and J. Shen, "Processor control flow monitoring using signed instruction streams," *IEEE Transactions on Computers*, Vol. C-36, pp. 264-276, March 1987.
- [11] N. Seifert, X. Zhu, D. Moyer, R. Mueller, R. Hokinson, N. Leland, M. Shade, and L. Massengill, "Frequency dependence of soft error rates for sub-micron CMOS technologies," in the Technical Digest of *IEEE International Electron Devices Meeting (IEDM)*, pp. 14.4.1-14.4.4, December 2001.
- [12] J. P. Shen, and M. A. Schuette, "On-line monitoring using signed instruction streams," *Proc. IEEE International Test Conference*, pp. 275-282, October 1983.
- [13] M. Sugihara, T. Ishihara, and K. Murakami, "Task scheduling for reliable cache architectures of multiprocessor systems," *Proc. IEEE/ACM Design, Automation and Test in Europe Conference*, pp. 1490-1495, Nice, France, April 2007.
- [14] M. Sugihara, "SEU vulnerability of multiprocessor systems and task scheduling for heterogeneous multiprocessor systems," *Proc. IEEE International Symposium on Quality Electronic Design*, pp. 757-762, San Jose, CA, USA, March 2008.
- [15] M. Sugihara, T. Ishihara, and K. Murakami, "Reliable cache architectures and task scheduling for multiprocessor systems," *IEICE Transactions on Electronics*, Vol. E91-C, No. 4, pp. 410-417, April 2008.
- [16] M. Sugihara, "Reliability inherent in heterogeneous multiprocessor systems and task scheduling for ameliorating their reliability," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, Vol. E92-A, No. 4, pp. 1121-1128, April 2009.
- [17] M. Sugihara, "Heterogeneous multiprocessor synthesis under performance and reliability constraints," *Proc. EUROMICRO Conference on Digital System Design*, pp. 333-340, Paris, Greece, August 2009.
- [18] K. Wilken and J. P. Shen, "Continuous signature monitoring: low-cost concurrent detection of processor control errors," *IEEE Transactions on Computer-Aided Design*, Vol. 9, No. 6, pp. 629-641, June 1990.



**Makoto Sugihara** was born in Tokyo, Japan in 1974. He received the B.E., M.E. and Ph.D. degrees in Computer Science and Communication Engineering from Kyushu University, Japan in 1996, 1998 and 2001, respectively. He joined the Fujitsu Laboratories, Japan, in 1998. From 2002 to 2003, He was a visiting researcher at Duke University, NC, USA. From 2003 to 2007, he was a Researcher at the ISIT, Fukuoka, Japan, and was also a Visiting Associate Professor at Kyushu University, Fukuoka, Japan. From 2007 to 2009, he was a Lecturer at Toyohashi University of Technology, Aichi, Japan. He is currently an Associate Professor at Toyohashi University of Technology, Aichi, Japan. His research interest includes design and test methodologies for VLSI chips and embedded systems. He is a member of the IEEE and the IEEE Computer Society, the IEICE, and the IPSJ. He received the IPSJ 40th Anniversary Best Paper Award, the Hiroshi Ando Memorial Award, and the Funai Information Science Encouragement Award in 2001, 2008, and 2009 respectively.