

A Compiler-Assisted On-Chip Assigned-Signature Control Flow Checking*

Xiaobin Li and Jean-Luc Gaudiot

Department of Electrical Engineering and Computer Science
University of California, Irvine
{xiaobinl, gaudiot}@uci.edu
<http://pascal.eng.uci.edu>

Abstract. As device sizes continue shrinking, lower charges are needed to activate gates, and consequently ever smaller external events (such as single ionizing particles of naturally occurring radiation) will be able to upset the correct functioning of complex modern microprocessors. Therefore, designers of future processors must take this new fact into account and should incorporate in their design fault-tolerant features which will allow processors to continue operating correctly even when such faults have occurred. Many faulty conditions are control flow errors which cause processors to violate the correct sequencing of instructions. Indeed, they amount to between 33% and 77% of all run-time errors. We present here a new compile-time signature assignment algorithm (the signature checking technique is a well-known approach to detect control flow errors). We also present the theoretical proof as well as the fault detection coverage analysis of our algorithm. We then describe the required enhancement to the basic microarchitecture: an on-chip assigned-signature checker which is capable of executing three additional instructions (SIC, SIJ, SIJC). This allows the processor to efficiently check the run-time sequence and detect control flow errors.

1 Introduction

As computer systems have become irreplaceable tools of modern society, with the benefits these systems bring to us comes a great potential for harm when they fail to perform their functions or perform them incorrectly. This is further exacerbated by new technologies of integration as the number of transistors and the clock rate of processors have shown an exponential growth rate [1]. However, smaller device sizes, reduced voltage levels, and higher transistor counts correspondingly raise concerns of higher transient faults rates. For one thing, radiation-induced soft errors are predicted to become increasingly significant in the near future [2,3,4]. In order to handle these inevitable errors, we must integrate in our design fault-tolerant features so that the processor can continue to correctly perform its specified tasks despite the occurrence of logic errors [5]. Such designs as Itanium [6], IBM Power4 [7], Fujitsu SPARC64 [8], etc., already include transient fault detection and recovery mechanisms.

* This paper is based upon work supported in part by NSF grants CCR-0234444 and INT-0223647. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

We concentrate here on protecting against *control flow errors* (those which cause a processor to violate the correct sequencing of instructions). Indeed, abstractions of program execution behavior can be formed based on various considerations which include control flow, memory access, I/O, and object type or range [9]. The cause of control flow errors could be the failure of any one of a variety of microarchitectural components such as instruction cache, program counter operation, branch unit, etc. Indeed, it has been found that these control flow errors account for between 33% [10] and 77% [11] of all run-time errors.

Signature checking is a well-known technique used to detect control flow errors [9, 11,12,13,14,15,16,17,18,19]. It can be implemented as either assigned-signature control flow checking or derived-signature control flow checking. In this paper, we focus on the former because it could offer better fault detection coverage. At compile time, we assign to each basic block a signature, and then at run time, an on-chip checker executes three additional signature checking instructions in order to check run time-computed signatures against assigned signatures. Any discrepancy indicates that an error occurred.

The goal of this paper is to describe the algorithm which protects against run-time control flow errors and its simple implementation. In Section 2, we introduce the principles of signature checking. The compile time signature assignment algorithm is outlined in Section 3. An on-chip checker with the ability to execute three signature checking instructions and its possible implementation are described in Section 4. These three instructions are proposed additions to a conventional instruction set. Conclusions are presented in Section 5.

2 The Concept of Signature Checking

Signature control flow checking techniques are used to monitor the program execution sequence in order to determine if a legal control flow is being followed. Various signature checking techniques have been proposed in the past [9,11,12,13,14,15,16,17,18,19]. Basically, there are **two** phases of signature checking: compile-time signature generation and run-time signature validation.

In the back end stages, in order to express the program control flow, compilers usually build a control flow graph (CFG), in which a *node* or a basic block is a sequence of instructions with no branch-in except for the entry point and no branch-out except for the exit point and *directed edges* are used to represent jumps in the program control flow [20]. Fig. 1 illustrates this concept by a simple example. Thus, in the first phase of signature checking, which is based upon the CFG, the compiler pre-computes the signatures associated with each node of the CFG, and then either embeds signatures into the original codes [9,11,13,14,16,19] or provides that information directly to the watchdog [15,18]. At this point, we could have two techniques for pre-computing signatures: the first, assigned-signature control flow checking [15,16,19], associates with each node an arbitrary signature, for example, a prime number. Conversely, the second technique, derived-signature control flow checking [9,11,13,14,18], derives signatures from the nodes themselves, for example by deriving a checksum from the binary code of the instruction inside a node and then using that checksum as the signature.

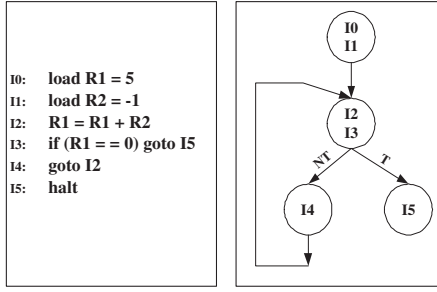


Fig. 1. An example program and its CFG

During the second phase, the checking engine, which can be either the watchdog or the host CPU, computes run time signatures and then check them against the compile time pre-computed signatures. If the signatures differ, it means that an error occurred.

Although the second phase of the assigned-signature checking algorithm and the derived-signature checking algorithm are essentially the same, assigned signature checking techniques have two major drawbacks: the need for registers to hold signatures and the performance overhead due to the need to execute extra instructions related to the assigned-signature checking [9,19]. For example, in [19], the overhead in terms of code size ranges from 26.6% to 61.9% while the overhead in terms of execution time ranges from 16.2% to 58.1%. Conversely, derived signature checking techniques require a signature generator/checker circuit to process the signature and might not guarantee that each node has a unique signature, which might consequently impact the fault coverage.

3 The Compiler-Assisted Signature Assignment Algorithm

The assigned-signature checking technique is based on a comparison between the compiler assigned signature with the one calculated at run time. Any difference between these two signatures indicates that a control flow error has occurred. To address the performance overhead associated with assigned-signature checking, we use additional hardware to trade this off, as will be explained in Section 4.

3.1 The Control Flow Checking Algorithm

Compiler time assigned reference signature (S). As discussed before, the program control flow can be expressed as a CFG. We start with a given node V_i of the CFG, and assign to it a **unique** number, which is called the *state code* of the node¹. This code is denoted by $D(i)$. Then, we compute the reference signature $S(i)$ of this node by using the following formula:

$$S(i) = D(i) \oplus D(pred(V_i)) \quad . \quad (1)$$

¹ A simple way to assign unique state codes to nodes may be to number each node of the CFG in sequence, as shown in Fig. 7.

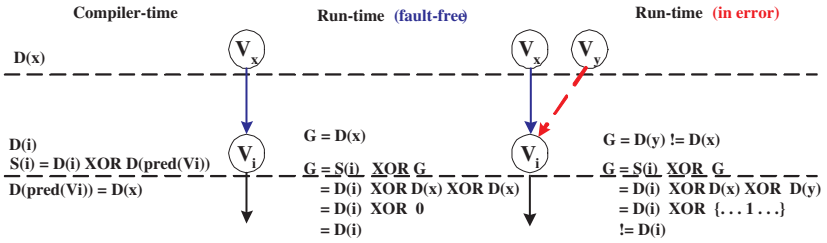


Fig. 2. Proof of the control flow checking algorithm

where $pred(V_i)$ is the immediate predecessor of node V_i in the CFG (note that \oplus is an exclusive-OR function). Furthermore, we assume for the moment that each node has only one immediate predecessor. More complex cases will be discussed later.

Run time signature (G). A global register holds the *run time signature* G of the node currently executing. When the program execution changes the control flow to a new node, e.g., V_i , G is updated by the following formula:

$$G = G \oplus S(i) \quad (2)$$

where $S(i)$ is the reference signature of the current new node V_i .

Then, the core of the control flow checking mechanism consists in checking the run time signature G against the static state code $D(i)$ (the one assigned by the compiler) as follows²:

```

Do    G ⊕ D(i)
      BNEZ exception-handler
    
```

Note that this comparison would take place whenever the run time control flow enters a new node of the CFG.

Proof (of the control flow checking algorithm). Assume (Fig. 2) that, instead of transferring control from V_x to V_i (left side of Fig. 2), we had erroneously entered V_i from V_y (right side of Fig. 2). Further assume that the reference signature of V_i had been assigned by the compiler to be $S(i) = D(x) \oplus D(i)$. The run-time signature G when the control of program is at node V_y is: $G = D(y)$ which would be different from $D(x)$ since the state code is *unique* to each node. Then, after entering node V_i , the run time signature would be updated by the following formula:

$$\begin{aligned}
 G &= G \oplus S(i) \\
 &= D(y) \oplus D(x) \oplus D(i)
 \end{aligned}$$

Since $D(x) \neq D(y)$, some bits of the result from $D(x) \oplus D(y)$ are 1s, as shown by $\{ \dots 1 \dots \}$ in Fig. 2. Because of these bits which have the value ‘1’ instead of ‘0’, when exclusive-ORed with $D(i)$, they will flip the corresponding bits of $D(i)$. As such, the final result is: $G \neq D(i)$ which means that a fault has been detected. \square

² BNEZ is equivalent to “branch to target if the result is not equal to zero.” As such, if $G \oplus D(i) \neq 0$, the exception handler is triggered.

Justifying signature (J) — Handling multiple-branch-in nodes. Now, we need to consider the case when a node has multiple immediate predecessors. Indeed, in normally complex CFGs, a node may have multiple immediate predecessors. We would call such a node a *multiple-branch-in* (MBI) node: it is a node whose number of immediate predecessors is greater than one. To simplify the discussion, we denote the set of $pred(MBI)$ as³:

$$\mathcal{S} = \{V_k \mid V_k \text{ is an immediate predecessor of MBI}\} . \tag{3}$$

When dealing with such MBI nodes, as required in (1), we must choose one of the immediate predecessors as the *primary immediate predecessor* (or primary node, for short). Also, since there is more than one path *up* from an MBI node, we associate with each immediate predecessor an additional parameter which we call the *justifying signature*. The justifying signature is used at run time to verify that all immediate predecessors to the MBI node are *legal* antecedents to that node.

The following outlines the *compile-time* MBI node handling algorithm:

1. Arbitrarily select a node from \mathcal{S} as the MBI node primary node (assume V_j for the rest of this discussion). Note that we leave the discussion of primary node selection later.
2. The reference signature of the MBI node is governed by the selected primary node V_j as:

$$S(MBI) = D(MBI) \oplus D(j) . \tag{4}$$

3. For every node $V_k \in \mathcal{S}$, associate it with the justifying signature given by the following formula:

$$J(k) = D(k) \oplus D(j) . \tag{5}$$

Note that now each node $V_k \in \mathcal{S}$ has **two** signatures: the reference signature $S(k)$ and the justifying signature $J(k)$ as illustrated in Fig. 3 where V_7, V_8 and V_9 are the MBI nodes⁴.

The *run-time* MBI node handling algorithm could be described as follows:

1. We denote control flow changes as: $V_k \rightarrow MBI$. First, the run time signature is updated according to the following formula:

$$G = G \oplus S(MBI) \oplus J(k) . \tag{6}$$

2. Finally, the MBI node run-time control flow checking can be applied as discussed before:

```
Do    G ⊕ D(MBI)
      BNEZ exception-handler
```

3.2 The Fault Detection Coverage of the MBI Node Handling Algorithm

Consider a general case of MBI node control flow change: $V_k \rightarrow MBI$. According to the relationship between the node V_k and the node “MBI,” there are three possible cases:

³ Then the definition of an MBI node can be given as the cardinality of \mathcal{S} , i.e., the number of elements in the set \mathcal{S} , is greater than one: $|\mathcal{S}| > 1$.

⁴ We use doubly circled nodes to represent primary nodes.

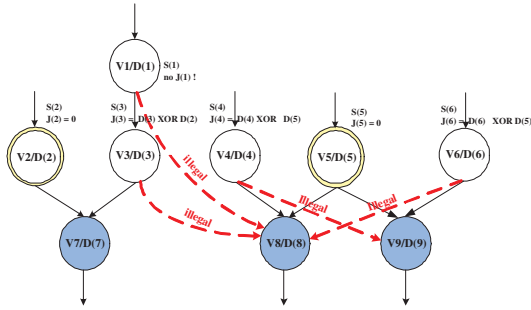


Fig. 3. Analysis of the MBI node handling algorithm

1. If $V_k \in \mathcal{S}$, which means that the control flow change is legal, as discussed before, we can easily prove that the updated run time signature is: $G = D(MBI)$.
2. If $V_k \notin \mathcal{S}$, which means that the control flow change is illegal, two cases must be separately considered:
 - a) V_k is an immediate predecessor of *another* MBI node, which means that $J(K)$ has been defined;
 - b) V_k is not an immediate predecessor of *any* MBI node. Then, $J(k)$ is null at compile-time and then without loss of generality, $J(k)$ will be a random number at run-time (whatever is left in the corresponding storage cell at that time).

Justifying signature has been defined. Consider the control flow change: $V_i \rightarrow V_j$ where V_j is an MBI node and its set of $pred(V_j)$ is \mathcal{S}_j . However, $V_i \notin \mathcal{S}_j$, i.e., the control flow change is illegal, V_i is instead an immediate predecessor of another MBI node, say V_m , hence $V_i \in \mathcal{S}_m$. Moreover, V_x has been selected as the primary node of V_j and V_y as the primary node of V_m . Then, when entering V_j , the run time signature is updated by using the following formula:

$$\begin{aligned}
 G &= G \oplus S(j) \oplus J(i) \\
 &= D(i) \oplus D(j) \oplus D(x) \oplus D(i) \oplus D(y) \\
 &= D(j) \oplus D(x) \oplus D(y)
 \end{aligned}$$

Therefore, two cases need to be considered:

1. As long as $D(x) \oplus D(y) = 0$, we end up with $G = D(j)$, which means that a control flow error has escaped detection. The faulty condition $D(x) \oplus D(y) = 0$ is satisfied only if $D(x) = D(y)$, i.e., node V_x is the same as node V_y (remember that the state code of each node is unique). Examples of illegal control flow changes such as $V_6 \rightarrow V_8$ and $V_4 \rightarrow V_9$, are shown in Fig. 3. In both cases, two MBI nodes V_8 and V_9 share node V_5 as their primary node.

Observation 1. When two MBI nodes, V_j and V_m , share their primary node, V_x ($= V_y$), any illegal control flow change: $V_i (\in \mathcal{S}_m \text{ and } \notin \mathcal{S}_j) \rightarrow V_j$ and any illegal control flow change: $V_l (\in \mathcal{S}_j \text{ and } \notin \mathcal{S}_m) \rightarrow V_m$ cannot be detected.

Further, we denote the probability of these illegal control flow changes as P_{ND1} .

2. On the other hand, if $V_x \neq V_y$, i.e., **no** primary node sharing, we have: $G \neq D(j)$ which means that the control flow error can be successfully detected. An example for this case is shown in Fig. 3 as the illegal control flow change: $V_3 \rightarrow V_8$. To summarize the above two cases, we can state the following:

Observation 2. *The fault detection coverage may decrease if two MBI nodes share the primary node. In another words, if a node V_x has multiple branch-outs, for example, the exit statement of the node is a conditional branch, and if more than two (including two) branch destination nodes are MBI nodes, the node V_x should not be selected as a primary node.*

Justifying signature is random. Consider the control flow change: $V_i \rightarrow V_j$ where V_j is an MBI node and its set of $pred(V_j) = \mathcal{S}_j$. Further assume that V_x has been selected as the primary node of V_j . However, $V_i \notin \mathcal{S}_j$, i.e., the control flow change $V_i \rightarrow V_j$ is illegal. Also, V_i is not an immediate predecessor of any MBI node such that $J(i)$ has not been defined and we deal with it as a random number. An example of an illegal control flow change: $V_1 \rightarrow V_8$ is shown in Fig. 3. In this case, when entering V_j , the run-time signature is updated by using the following formula:

$$\begin{aligned} G &= G \oplus S(j) \oplus J(i) \\ &= D(i) \oplus D(j) \oplus D(x) \oplus J(i) \end{aligned}$$

Because of the randomness of $J(i)$, two cases must be considered:

1. If $D(i) \oplus D(x) \oplus J(i) = 0$, we have $G = D(j)$ which means that the control flow error escapes detection;
2. If $D(i) \oplus D(x) \oplus J(i) \neq 0$, we have $G \neq D(j)$ which means that the algorithm has successfully detected the control flow error.

Fortunately, the probability for $D(i) \oplus D(x) \oplus J(i)$ to be zero is very low: it can happen only if $J(i) = D(i) \oplus D(x)$. Given the n-bit size of state codes and signatures, the probability is:

$$P_{ND2} = P\{J(i) = D(i) \oplus D(x)\} = 2^{-n} . \tag{7}$$

In summary, the fault detection coverage of the MBI node handling algorithm is:

$$\begin{aligned} C &\equiv P\{\text{fault detection} | \text{fault existence}\} & (8) \\ &= P\{\text{control flow error detection} | V_k \rightarrow \text{MBI and } V_k \notin \mathcal{S}\} & (9) \\ &= 1 - P_{ND1} - P_{ND2} . & (10) \end{aligned}$$

3.3 The If-Then-Else Node Handling Algorithm

As mentioned by Oh et al. in [19], primary nodes are randomly selected which would contradict our Observation 2. Furthermore, randomly selecting the primary node may

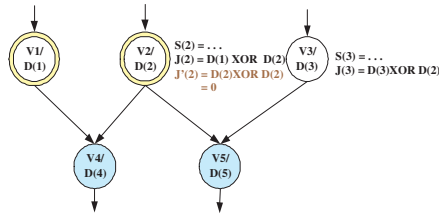


Fig. 4. An example of ITE node with two justifying signatures

result in conflicts as illustrated in Fig. 4. Indeed, if V_1 had been selected as the primary node for V_4 and V_2 for V_5 , respectively, we would have to create two justifying signatures for node V_2 : as far as MBI node V_4 is concerned, the justifying signature of V_2 is: $J(2) = D(1) \oplus D(2)$; whereas as far as MBI node V_5 is concerned, the justifying signature of V_2 is: $J'(2) = D(2) \oplus D(2) = 0$.

Hence, for the control flow change: $V_2 \rightarrow V_4$, $J(2)$ should be used to update the run-time signature whereas for the control flow change: $V_2 \rightarrow V_5$, only $J'(2)$ is the correct choice. Anything corrupted up to this level could result in faulty control flow error detection. Simply speaking, for the **legal** control flow change: $V_2 \rightarrow V_4$, if the justifying signature $J'(2)$ had been used to update the run time signature, we would end up with $G \neq D(4)$ such that a control flow error could be flagged, a false alarm. Unfortunately, such situations⁵ have not been addressed in [19].

The necessary conditions for a node associated with two⁶ justifying signatures are:

1. The exit of the node is a conditional branch, that is to say the node is an if-then-else (ITE) node;
2. Both branch destinations are MBI nodes.

In short, we need to distinguish the two justifying signatures: one for the then-branch flow (the resolved branch condition is not-taken), the other for the else-branch flow (the resolved branch condition is taken).

Figure 5 shows a hardware-based algorithm⁷: at **compile time**, when an MBI node traces back its immediate predecessors for the purpose of justifying signatures, the associated directed edges are checked (directed edges are given by the CFG): if the edge is a “taken” path, the associated justifying signature will be placed into the TJ register; whereas if it is a “not-taken” path, the NTJ register is used for the associated justifying

⁵ These situations are not rare: conditional branches are extremely common in regular programs.

From the following discussions, we will see that a node with a conditional branch could be associated with two justifying signatures.

⁶ If switch statements are allowed, that more than two justifying signatures are associated with a node is possible. However, we assume that the compiler has converted all switch statements into the equivalent if-then-else constructs, as presented in [14].

⁷ We have not, in this work, considered checking the flow of conditional branches. More specifically, refer to Fig. 5, the case when a transient fault causes the ITE node to branch to the else_node incorrectly whereas it should have branched to the then_node, has not been considered.

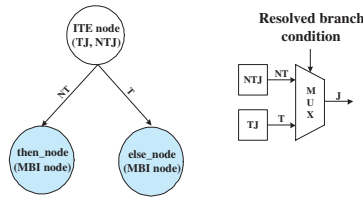


Fig. 5. An hardware approach for ITE node with two justifying signatures (T = taken; NT = not-taken; TJ = justifying signature for ITE_node → else_node; NTJ = justifying signature for ITE_node → then_node)

signature. At **run time**, the resolved branch condition is used to select the appropriate justifying signature for updating the run time signature. More details will be given in Section 4.

4 Hardware Enhancement for Control Flow Checking

As discussed before, the assigned-signature checking technique has an inherent performance overhead drawback. However, with advances in CMOS technology, we have an abundance of cheap hardware resources [1]. Moreover, our proposed mechanism can be simply implemented in any modern microprocessor at little additional cost. Hence, in this section, we first introduce three additional instructions dedicated to control flow checking, and then design a simple hardware implementation to execute these instructions. We will also provide a comprehensive control-flow checking algorithm based on these hardware enhancements. In the end, we will show the benefit from trading hardware off a reduction in performance overhead.

4.1 Additional Instructions

The three additional instructions dedicated to the assigned-signature control flow checking are succinctly described in Table 1. Instruction SIC is used to check for control flow errors in non-MBI nodes. Instruction SIJ is dedicated to signature justification. Instruction SIJC is used to check for control flow errors in MBI nodes. The compiler is responsible for the insertion of these additional instructions into the original program so as to achieve run-time control flow checking. The detailed algorithm will be presented in section 4.3.

4.2 Implementation of Additional Instructions — On-Chip Control Flow Checker

A simple on-chip control flow checker to execute the above three additional instructions can be easily designed. Assume a simple five-stage pipeline: Fetch → Decode → Execution → Memory access → Write back. Our on-chip control flow checker would be located in the “Decode” and “Execution” stages.

Table 1. Three additional instructions specification

No.	Mnemonic	Format	Function Description
1	SIC	SIC imm1, imm2 where imm1 = S(i); imm2 = D(i)	Signature checking : 1 Update G as: $G = G \oplus \text{imm1}$ 2 If ($G == \text{imm2}$) fault free, Else control flow error;
2	SIJ	SIJ imm1, imm2 where imm1/imm2 = D(i) \oplus D(j) and: If (ITE node) imm1 for NTJ, imm2 for TJ, Else imm1 for J	Signature justifying: Update J as: $J = \text{imm1}/\text{imm2}$ depended on resolved branch condition
3	SIJC	SIJC imm1, imm2 where imm1 = S(i); imm2 = D(i)	MBI node Signature checking: 1 Update G as: $G = G \oplus \text{imm1} \oplus J$ 2 If ($G == \text{imm2}$) fault free, Else control flow error;

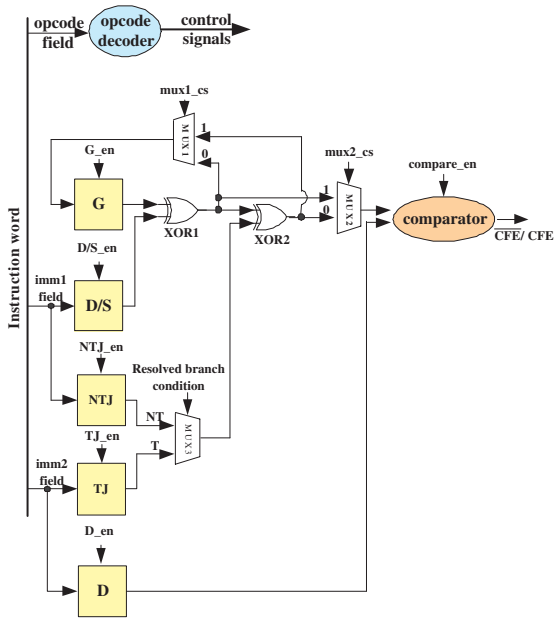


Fig. 6. On-chip control flow checker block diagram

As seen in Fig. 6, a total of five registers are needed to hold the necessary information: register **G** is used for the run time signature; registers **D/S** and **NTJ** receive immediate values from the *imm1* field of their instruction words and registers **D** and **TJ** receive immediate values from the *imm2* field of the instruction words. For each instruction, Table 2 shows the control signals generated by the opcode decoder (also shown in Fig. 6).

Table 2. On-chip control flow checker control signals (en = enable; \overline{en} = disable; X = don't care; BRU = branch unit)

Instr.	G_en	D/S_en	D_en	NTJ_en	TJ_en	mux1_cs	mux2_cs	br. cond.	compare_en
SIC	en	en	en	\overline{en}	\overline{en}	0	1	\overline{en}	en
SIJ	\overline{en}	\overline{en}	\overline{en}	en	en	X	X	\overline{en}	\overline{en}
SIJC	en	en	en	\overline{en}	\overline{en}	1	0	from BRU	en

Operation of SIC instructions. When an instruction word SIC imm1, imm2 is decoded, its opcode field is fed into the opcode decoder. The decoder then generates the control signals specified in Table 2. The imm1 field is received by the enabled register D/S (D/S_en = enable and NTJ_en = \overline{enable}) while the imm2 field is received by the enabled register D (D_en = enable and TJ_en = \overline{enable}).

The content of register D/S goes into XOR1 along with the content of register G (G_en = enable). The result is selected by mux2 (mux2_cs = 1). Now the enabled comparator compares the two inputs which are received from mux2 and register D. These have performed the run-time control flow checking. Also, we can see the result of XOR1 is sent to modify register G since we have mux1_cs = 0 and G_en = enable.

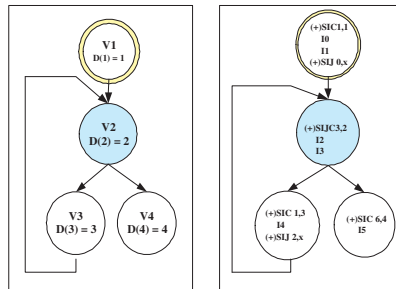
Operation of SIJ instructions. When an instruction word SIJ imm1, imm2 is decoded, its opcode field is fed into the opcode decoder. The decoder then generates the control signals specified in Table 2. The imm1 field is received by the enabled register NTJ (NTJ_en = enable and D/S_en = \overline{enable}) while the imm2 field is received by the enabled register TJ (TJ_en = enable and D_en = \overline{enable}).

Operation of SIJC instructions. When an instruction word SIJC imm1, imm2 is decoded, its opcode field is fed into the opcode decoder. The decoder then generates the control signals specified in Table 2. The imm1 field is received by the enabled register D/S (D/S_en = enable and NTJ_en = \overline{enable}) while the imm2 field is received by the enabled register D (D_en = enable and TJ_en = \overline{enable}).

The content of register D/S goes into XOR1 along with the content of register G (G_en = enable). Based on the resolved branch condition, either the content of register NTJ or that of register TJ XOR2 with the result of XOR1. Once again, if no conditional branch result from the branch unit, i.e., not an ITE node, the default “resolved branch condition = NT” such that the content of register NTJ is selected at this point. MUX2 selects the result of XOR2 (mux2_cs = 0). Now the enabled comparator compares the two inputs which are received from mux2 and register D. These have performed the run-time control flow checking. Furthermore, we can see that the result of XOR2 is sent to modify register G since we have mux1_cs = 1 and G_en = enable.

4.3 Using Additional Instructions

To summarize the above discussion, Algorithm 1 shows a comprehensive signature assignment algorithm based on our hardware enhancement instructions. Returning to the example of Fig. 1, our compiler algorithm would produce the modified diagram

Algorithm 1 Embedding three additional instructions into programs**Signature-Assignment(program)****Derive** CFG of the given programAssuming we have nodes: V_i ($i = 1, 2, 3, \dots, N$) where N is the total number of nodes in the CFG**Assign** a unique state code $D(i)$ to every node V_i **for** every node V_i in the CFG **do** **if** V_i is not an MBI node **then** **Compute** its assigned reference signature as: $S(i) = D(i) \oplus D(pred(V_i))$; **Place** the instruction “SIC imm1, imm2” at the beginning of node V_i and before the SIJ instruction, if any **Assign** the values of imm1 and imm2 as: imm1 = S(i) and imm2 = D(i) **else** **Select** a primary node from: \mathcal{S} = the set of $pred(V_i)$ (assume V_j is selected) **Assign** the reference signature of V_i as: $S(i) = D(i) \oplus D(j)$ **Place** the instruction “SIJC imm1, imm2” at the beginning of node V_i and before the SIJ instruction, if any **Assign** the values of imm1 and imm2 as: imm1 = S(i), imm2 = D(i) **for** every node $V_k \in \mathcal{S}$ (including V_j) **do** **Place** instruction “SIJ imm1, imm2” into the node V_k and after the SIC and/or SIJC instructions **Assign** the values of imm1 and imm2 as follows: **if** $V_k \rightarrow V_i$ is a taken path **then** imm1 = X, imm2 = $D(k) \oplus D(j)$ **else if** $V_k \rightarrow V_i$ is a not-taken path **then** imm1 = $D(k) \oplus D(j)$, imm2 = X **else** imm1 = $D(k) \oplus D(j)$, imm2 = X { $V_k \rightarrow V_i$ is not a conditional branch path } **end if** **end for** **end if****end for****Fig. 7.** State code and signature assignment example

shown in Fig. 7. The left-hand side illustrates the state code assignment results and the primary node selection of the MBI node V_2 . The right-hand side shows the CFG after insertion of our control flow checking instructions.

Comparing code size overhead. To compare our algorithm with that of Oh et al. in [19], consider a *typical* node consisting of 7 to 8 instructions [1]. In order to check for control flow errors, [19] adds 2 to 4 instructions to each node. The overhead is between 27% and 53%. (As shown in [19], for a number of benchmarks, the code size overhead is between 26.6% and 61.9% whereas the execution time overhead is between 16.2% and 58.1%). Conversely, in our hardware-enhanced approach, the additional instructions are a maximum of 1 or 2 for each node. Therefore, the overhead is only 13% to 27%, which is a significant improvement over [19]. Furthermore, the execution time is given by the following formula [1]:

$$\text{Execution time} = \text{Instr count} \times \text{Clock cycle time} \times \text{Cycles per instr} \quad (11)$$

With help from our on-chip control flow checker, we could expect a lower execution time than that obtained in [19] when executing the program with the signature checking. This is because we have a smaller instruction count given the same clock cycle time and cycles per instructions.

5 Conclusions

Control flow errors have a high error occurrence ratio relative to other kinds of errors. This is expected to continue increasing as design rules continue decreasing. Signature checking is a well-known and effective technique to detect such errors. We have used this approach to demonstrate our compiler-assisted assignment signature analysis. It includes a compile time algorithm based on a control flow graph which assigns signatures to nodes. We have also designed an on-chip checker for our dedicated instructions used for control flow checking. A comprehensive signature assignment algorithm has also been introduced, and a detailed performance overhead analysis has been presented.

References

1. Hennessy, J.L., Patterson, D.A.: *Computer Architecture: A Quantitative Approach*. Third edn. Morgan Kaufmann Publishers, Inc. (2002)
2. Borkar, S.: Design Challenges of Technology Scaling. *IEEE Micro* (1999)
3. Yang, P., Chern, J.H.: Design for Reliability: The Major Challenge for VLSI. *Proceedings of the IEEE* (1999)
4. Reinhardt, S.K., Mukherjee, S.S.: Transient Fault Detection via Simultaneous Multithreading. In: *27th International Symposium on Computer Architecture*. (2000)
5. Hennessy, J.: The Future of Systems Research. *IEEE Computer* (1999)
6. Quach, N.: High Availability and Reliability in the Itanium Processor. *IEEE Micro* (2000)
7. Bossen, D.C., Tendler, J.M., Reick, K.: Power4 System Design for High Reliability. *IEEE Micro* (2002)
8. Ando, H., Yoshida, Y., Inoue, A., Sugiyama, I., Asakawa, T., Morita, K., Muta, T., Motokurumada, T., Okada, S., Yamashita, H., Satsukawa, Y., Konmoto, A., Yamashita, R., Sugiyama, H.: A 1.3-GHz Fifth-Generation SPARC64 Microprocessor. *IEEE Journal of Solid-State Circuits* (2003)
9. Wilken, K., Shen, J.P.: Continuous signature monitoring: Low-Cost Concurrent-Detection of Processor Control Errors. *IEEE Transactions on Computer-Aided Design* (1990)

10. Ohlsson, J., Rimen, M., Gunneflo, U.: A Study of the Effects of Transient Fault Injection Into a 32-bit RISC with Built-in Watchdog. In: 29th International Symposium on Fault-Tolerant Computing. (1991)
11. Schuette, M.A., Shen, J.P.: Processor Control Flow Monitoring Using Signed Instruction Streams. *IEEE Transactions on Computers* (1987)
12. Mohmood, A., McCluskey, E.J.: Concurrent Error Detection Using Watchdog Processors – A Survey. *IEEE Transactions on Computers* (1988)
13. Schuette, M.A., Shen, J.P.: Exploiting Instruction-Level Parallelism for Integrated Control-Flow Checking. *IEEE Transactions on Computers* (1994)
14. Warter, N.J., Hwu, W.M.W.: A Software Based Approach to Achieving Optimal Performance for Signature Control Flow Checking. In: 20th International Symposium on Fault-Tolerant Computing. (1990)
15. Michel, T., Leveugle, R., Saucier, G.: A New Approach to Control Flow Checking without Program Modification. In: 21st International Symposium on Fault-Tolerant Computing. (1991)
16. Alkhalifa, Z., Nair, S., Krishnamurthy, N., Abraham, J.A.: Design and Evaluation of System-Level Checks for On-Line Control Flow Error Detection. *IEEE Transactions on Parallel and Distributed Systems* (1999)
17. Shirvani, P.P., McCluskey, E.J.: Fault-Tolerant Systems in a Space Environment: The CRC ARGOS Project. Technical Report CRC-TR 98-2, Stanford University (1998)
18. Bagchi, S., Srinivasan, B., Whisnant, K., Kalbarczyk, Z., Iyer, R.K.: Hierarchical Error Detection in a Software Implemented Fault Tolerance (SIFT) Environment. *IEEE Transactions on Knowledge and Data Engineering* (2000)
19. Oh, N., Shirvani, P.P., McCluskey, E.J.: Control-Flow Checking by Software Signatures. *IEEE Transactions on Reliability* (2002)
20. Aho, A.V., Sethi, R., Ullman, J.D.: *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company (1986)