Int J Parallel Prog (2010) 38:85–116
DOI 10.1007/s10766-009-0114-9

Tolerating Radiation-Induced Transient Faults in Modern Processors

Xiaobin Li · Jean-Luc Gaudiot

Received: 19 March 2009 / Accepted: 7 July 2009 / Published online: 24 July 2009

Abstract As MOS device sizes continue shrinking, lower charges, for example those charges carried by single ionizing particles of naturally occurring radiation, are sufficient to upset the functioning of complex modern microprocessors. In order to handle these inevitable errors, designs should include fault-tolerant features so that the processors can continue to correctly perform despite the occurrence of errors. The main goal of this work is to develop architecture mechanisms to protect processors against the effect of such radiation-induced transient faults. It should first be noted that, from a program execution perspective, many faults manifest themselves as control flow errors that cause processors to violate the correct sequencing of instructions. We present here at first a basic compile-time signature assignment algorithm and describe a novel approach to improve the fault detection coverage of the basic algorithm. Moreover, to allow the processor to efficiently check the run-time sequence and detect control flow errors, we introduce an on-chip assigned-signature checker which is capable of executing three additional instructions (SIC, SIJ, SIJC). Second, since the very concept of simultaneous multi-threading (SMT) provides the necessary redundancy, some proposals have been made to run two copies of the same thread on top of SMT platforms in order to detect and correct soft errors. This allows, upon detection of an error, the rolling back of the processor state to a known safe point, and then a retry of the instructions, thereby effecting a completely error-free execution. This paper has focused on two crucial implementation issues introduced by this

X. Li (✉)
Enterprise Microprocessor Group, Intel Corporation, Santa Clara, CA, USA
e-mail: xiaobin.li@intel.com

J.-L. Gaudiot
Department of Electrical Engineering and Computer Science,
University of California, Irvine, CA, USA
e-mail: gaudiot@uci.edu

scheme: (1) the design trade-off between the fault detection coverage versus design costs; (2) the possible occurrence of deadlock situations.

Keywords Soft-error · Computer architecture · Fault-tolerant · Control flow checking · Multi-threading

1 Introduction

As MOS device sizes continuing shrinking, the reliability of the computer systems that are built upon these devices is of increasing concern [1]. For one thing, radiation-induced soft errors will become more significant in the near future [2–4]. In order to handle these inevitable errors, we must integrate in our design fault-tolerant features so that processors can continue to correctly perform their specified tasks despite the occurrence of logic errors [5]. Such designs as the Intel Itanium [6,7], the IBM Power6 [8], the z10 [9], the Fujitsu SPARC64 [10], etc., already include transient fault detection and recovery mechanisms.

Depending on the kind of micro-architectural components where the error occurs, from a program execution perspective, the soft error induced may manifest itself in different ways. For instance, an ALU computation error would lead to a data error whereas if the program counter is corrupted by the incoming particle, the execution would end up an incorrect execution sequence. Therefore, the fault-tolerance design is as usual an ad-hoc process and typically different approaches are needed to protect different parts of the processor against different failure modes.

First, many faults manifest themselves as control flow errors that cause processors to violate the correct sequencing of instructions. Signature checking is a well-known scheme used to detect this type of error. We will present a basic compile-time signature assignment algorithm and describe a novel approach to improve the fault detection coverage of the basic algorithm.

As far as data errors are concerned, since the very concept of Simultaneous Multi-Threading (SMT) provides the necessary redundancy, some proposed schemes entail running two copies of the same thread on top of SMT platforms in order to detect and correct soft errors. This allows, upon detection of an error, the rolling back of the processor state to a known safe point, and then a retry of the instructions, thereby effecting a completely error-free execution.

The goal of this paper is thus to describe two approaches which will enhance the robustness of processors in the presence of transient faults such as those caused by cosmic rays. Section 2 defines control flow errors and outlines our signature approach to protect against them. The complete algorithm is described in Sect. 3 and the hardware enhancement is present in Sect. 4. In Sect. 5, we turn our attention to those errors which have not been caught by the signature approach and use the inherent redundancy in Simultaneous Multi-Threaded processors to produce redundant execution of threads and verify the correctness of program execution. Two crucial implementation issues are then addressed: Sect. 6 describes the design trade-off between the fault detection coverage and design costs while Sect. 7 discusses the possible occurrence of deadlock situations. The design overhead evaluation of the proposed design is presented in Sect. 8. We conclude our work in Sect. 9.

2 Control Flow Errors

From an instruction set architecture standpoint, an abstraction of program execution behavior can be based on various considerations which include control flow, memory and I/O access, and object type and range [11,12]. We first examine how to protect against *control flow errors* (those which cause a processor to violate the correct sequencing of instructions).

The causes of control flow errors can be traced to the failure of one of many micro-architecture components such as the instruction cache, the program counter unit, the jump execution unit, etc. The instruction cache stores the program content; for instance, if an error causes a stored branch instruction content to change, the pipeline would then be executed in the wrong program order. The program counter unit is merely a pointer to the instructions to be fetched. Conceivably, if its content is corrupted by some radiation-induced error, incorrect instructions would be fed into the execution stage, which in turn would end up as an instruction sequencing violation. Similarly, the jump execution unit computes the target address of the jump instruction. When an error causes an incorrect computation, we would consequently encounter an unexpected program order. Indeed, it has been found that these control flow errors account for between 33% [13] and 77% [14] of all run-time errors.

Signature control flow checking techniques are used to monitor the program execution sequence in order to determine if the legal control flow is being followed. Various signature checking techniques have been proposed in the recent past [12,14–22]. Basically, there are two steps in signature checking: compile-time signature generation followed by run-time signature verification.

In the back-end stages, in order to express the program control flow structures, compilers typically 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 where *directed edges* are used to represent jumps in the program control flow [23]. Figure 1 illustrates the CFG concept by a simple example. Thus, in the first step 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 those signatures into the original codes [12,14,16,17,19,22], or provides that information directly to the watchdog [18,21]. It should be noted that at this step, we

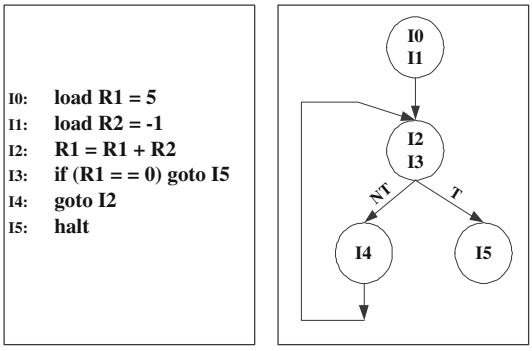


Fig. 1 An example program and its control flow graph

could have two signature pre-computing techniques: the first, *assigned-signature control flow checking* [18,19,22], associates with each node an arbitrary signature, for example, a prime number. Conversely, the second technique, *derived-signature control flow checking* [12,14,16,17,21], derives signatures from the nodes themselves. For example, we derive a checksum from the binary code of the instruction inside a node and then use that checksum as the corresponding node signature.

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

Although the second step of both 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 [12,22]. For example, Oh et al. [22] have shown that 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 might not guarantee that each node has a unique signature, which might consequently impact the fault coverage. For instance, assume two nodes with identical signatures and a control flow error that is transferring execution into an incorrect node but with the same signature. Such an error cannot be detected.[1] It should be noted that Borin et al. [24] have demonstrated that the performance slowdown of the software-only derived-signature checking would be an average of 30%.

3 Algorithm for Compiler-Assisted Signature Checking

As discussed above, 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. This section describes in detail a compiler-assisted signature assignment and checking algorithm. As for how to address the inherent performance overhead associated with assigned-signature checking, we propose to use additional hardware to trade this off, as will be explained in Sect. 4.

3.1 The Basic Assigned-Signature Control Flow Checking Algorithm

3.1.1 Compiler Time Assigned Reference Signature (S)

As discussed before, the program control flow structure 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 node.[2] This state code is denoted *D(i)*. Then, we compute

[1] Refer to Sect. 3.2 for a detailed discussion.

[2] 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.

the reference signature *S(i)* of this node by using the following formula:

$$S(i) = D(i) \oplus D(pred(V_i)). \tag{1}$$

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.

3.1.2 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:

$$G = G \oplus S(i). \tag{2}$$

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 the checking of the run time signature G against the static state code *D(i)* (the one assigned by the compiler) as follows:[3]

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

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

3.1.3 Justifying Signature (J)—Handling Multiple-Branch-In Nodes

Now, we start considering the more complex 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)* as:[4]

$$\mathbb{S} = \left\{ V_k \,\middle|\, V_k \text{ is an immediate predecessor of MBI} \right\}. \tag{3}$$

When dealing with such MBI nodes, as required in eq. (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

[3] JNZ is equivalent to "jump to target if the result is not equal to zero." As such, if $G \oplus D(i) \neq 0$, the exception handler is triggered.

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

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 $\mathbb{S}$ 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 $V_k \in \mathbb{S}$, 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 $V_k \in \mathbb{S}$ 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 nodes.[5]

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)
JNZ exception-handler
```

We use Fig. 2 to illustrate the simple example of an MBI node and how justifying signature works. Node V_3 is an MBI node and its immediate predecessor set is $\mathbb{S} = \{V_1, V_2\}$ whereas V_1 is arbitrarily selected as the primary node. At compile time, after assigning the reference signatures *S(1)* and *S(2)* to nodes V_1 and V_2, the justifying signatures *J(1)* and *J(2)* are also evaluated. As we can see from Formula 6, the run time signature G carried when entering the MBI node V_3 is updated by **two** exclusive-OR functions (Refer to Formula 2, only one exclusive-OR function is required there): first, perform an exclusive-OR with the reference signature of node V_3, and then perform

[5] Hereafter, we use doubly circled nodes to represent primary nodes.

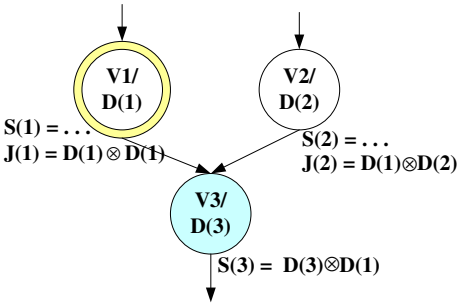


Fig. 2 An example of MBI node and justifying signatures

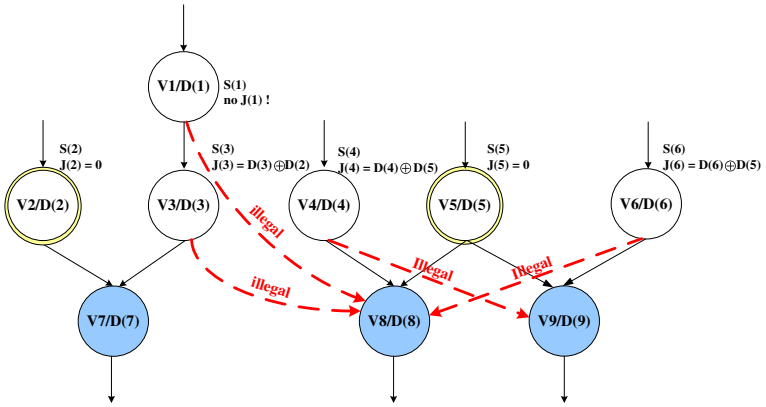


Fig. 3 Fault detection coverage analysis for the MBI node handling algorithm

another exclusive-OR with the justifying signature of the jumping node. For example, when the control flow change is $V_2 \rightarrow V_3$, the run time signature is calculated by:

$$\begin{aligned} G &= G \oplus S(3) \oplus J(2) \\ &= D(2) \oplus D(3) \oplus D(1) \oplus D(1) \oplus D(2) \\ &= D(3) \end{aligned}$$

3.2 Analyzing the Fault Detection Coverage of the Basic Algorithm

To the end of improving the fault detection coverage of the above basic algorithm, we start by first analyzing the coverage: consider the general case of an 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:

1. If $V_k \in \mathbb{S}$, 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 $V_k \notin \mathbb{S}$, which means that the control flow change is illegal and that 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).

3.2.1 Justifying Signature has been Defined

Consider the following control flow change: $V_i \rightarrow V_j$ where V_j is an MBI node and its set of $pred(V_j)$ is $\mathbb{S}_j$. However, $V_i \notin \mathbb{S}_j$, *i.e.*, the control flow change is illegal, V_i is instead an immediate predecessor of another MBI node, say V_m, hence $V_i \in \mathbb{S}_m$. 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: $V_i (\in \mathbb{S}_m$ and $\notin \mathbb{S}_j) \rightarrow V_j$ and any illegal control flow change: $V_l (\in \mathbb{S}_j$ and $\notin \mathbb{S}_m) \rightarrow V_m$ 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 an illegal control flow change, from $V_3 \rightarrow V_8$.

   To summarize the above two cases, we can state the following:

   Observation 2

   The fault detection coverage may decrease if two MBI nodes share their primary node. In other 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.

3.2.2 Random Justifying Signature

Consider the following control flow change: $V_i \rightarrow V_j$ where V_j is an MBI node and its set of $pred(V_j) = \mathbb{S}_j$. Further assume that V_x has been selected as the primary node of V_j. However, $V_i \notin \mathbb{S}_j$, *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 would be $V_1 \rightarrow V_8$, which 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:

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

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

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

3.3 Improving the Fault Detection Coverage of the Basic Algorithm

Based on the above discussion, we know that to improve the fault detection coverage, reducing $\mathrm{P}_{\mathrm{ND1}}$ is the primary approach, which in turn requires selecting the primary node from those not being shared by MBI nodes. In other words, by carefully handing those cases we analyzed in Sect. 3.2, we can improve the fault detection coverage over the previous work, such as described by Oh et al. [22]. Therefore, we propose a two-pass algorithm for the purpose of selecting the primary node for MBI nodes, which is specified in Algorithm 1: given an MBI's immediate predecessor list, we choose the one that has the smallest chance of being shared. To this end, we first traverse CFG to count how many children nodes a node has, and then the node with the least number of children nodes is the best candidate to be the primary node.

Algorithm 1: Two-pass algorithm for selecting the primary node

```
Initialization: for every node in CFG do
    pn_cnt = 0;
    pn_selected = FALSE;
end
First-Pass: for every MBI node do
    for each of its primary node do
        pn_cnt ++;
    end
end
Second-Pass: for each MBI node do
    select primary node with min(pn_cnt) and pn_selected = FALSE ;
    mark the selected primary node's pn_selected = TRUE;
    for each of this MBI's immediate predecessor but not being selected do
        pn_cnt - -;
    end
end
```

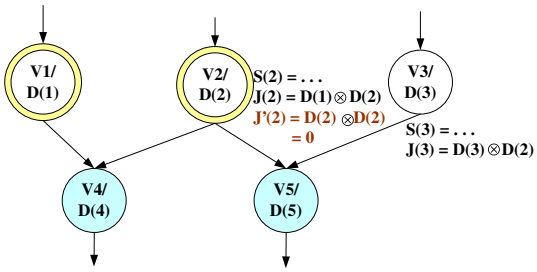


Fig. 4 An example of ITE node with two justifying signatures

3.4 Handling the If-Then-Else Node

In addition to the fault detection coverage improvement, we feel it is important to remark that randomly selecting the primary node may 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. Such situations[6] have not been addressed by Oh et al. [22].

[6] These situations are not rare: conditional branches are extremely common in regular programs. From the following discussion, we will see that a node with a conditional branch could be associated with two justifying signatures.

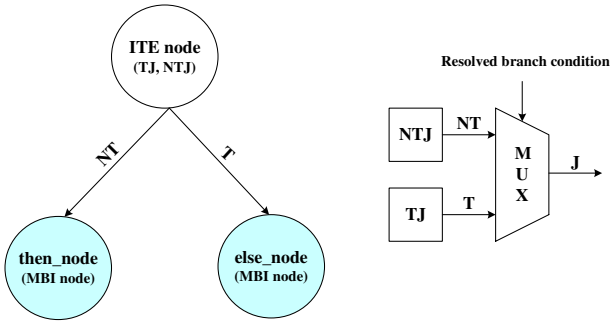


Fig. 5 An hardware approach for ITE node with two justifying signatures (T = taken; NT = not-taken; TJ = justifying signature for ITE_node → else_node; NTJ = justifying signature for ITE_node → then_node)

The necessary conditions for a node associated with two[7] 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 algorithm:[8] 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 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 Sect. 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

[7] If switch statements are allowed, having more than two justifying signatures 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 [17].

[8] 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 incorrectly to the else_node whereas it should have branched to the then_node, has not been considered.

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 ⊕ imm1 2 **If** (G == imm2) fault free, **Else** control flow error;
2	`SIJ`	`SIJ imm1, imm2` where `imm1` / `imm2` = D(i) ⊕ D(j) **If** (ITE node) `imm1` for NTJ, `imm2` for TJ, **Else** `imm1` for J	**Signature justifying:** Update J as: J = imm1/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 ⊕ imm1 ⊕ J 2 **If** (G == imm2) fault free, **Else** control flow error;

these instructions.[9] 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 sect. 4.3.

4.2 Implementation of Additional Instructions

The on-chip control flow checker to execute the above three additional instructions is relatively 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.

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).

[9] The microprocessor instruction set is often extended to implement new features, a fact which can be exemplified by MMX technology and Intel Streaming SIMD Extensions [11].

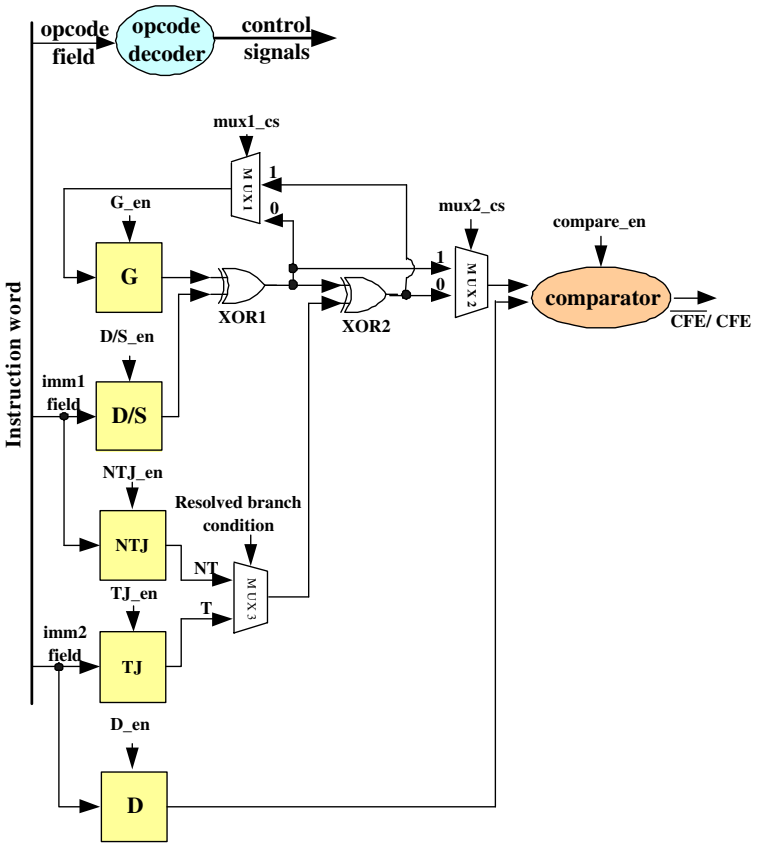


Fig. 6 On-chip control flow checker block diagram

Table 2 On-chip control flow checker control signals

Instruction	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

en = enable; $\overline{en}$ = disable; X = don't care; BRU = branch unit

4.2.1 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` filed 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.

4.2.2 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 = $\overline{enable}$).

4.2.3 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 results 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 3 is 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 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.

4.4 Comparing Code Size Overhead

To compare our algorithm with that of Oh et al. in [22], consider a *typical* node consisting of 7–8 instructions [1]. In order to check for control flow errors, the algorithm in [22] adds 2–4 instructions to each node. The overhead is between 27 and 53%. For a number of benchmarks, the code size overhead is between 26.6 and 61.9% whereas

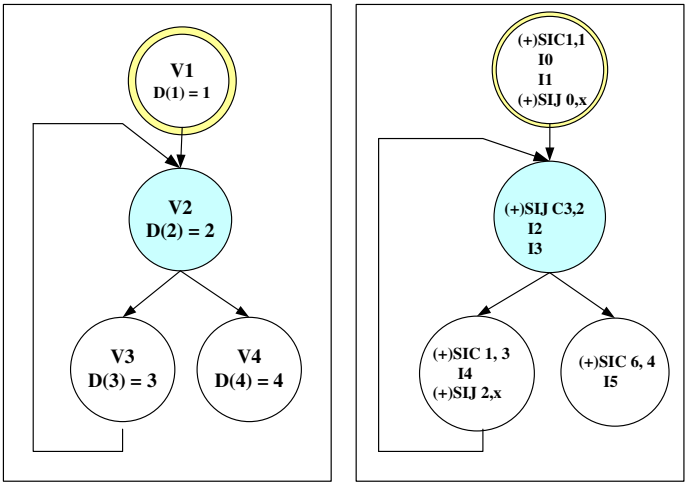


Fig. 7 State code and signature assignment example

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 13–27%, which is a significant improvement over [22]. Furthermore, the execution time is given by the following formula [1]:

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

With the assistance of our on-chip control flow checker, we could expect a lower execution time than that obtained in [22] when executing the program with the signature checking. This is because we have a smaller instruction count given the same clock cycle time and number of cycles per instructions.

5 Transient Fault-Tolerant SMT Processors

The control flow checking targets faults which result in the correct program sequencing being violated. To increase the fault coverage to tolerate those errors manifested as data errors, we propose in this section a transient fault-tolerant Simultaneous Multi-Threading processor design.

The multi-threading execution paradigm inherently provides the spatial and temporal redundancy necessary for fault-tolerance: the basic idea is that we run two copies of the same threads on Simultaneous Multi-Threading platforms [25,26,4,27,28] and then the results of the two copied threads are compared to detect transient faults. This allows, upon detection of an error, the rolling back of the processor state to a known safe point, and then a retry of the instructions, thereby effecting a completely error-free execution.

5.1 Trade-Off Between Fault Detection Coverage and Design Costs

A design trade-off is associated with the above basic idea: Generally speaking, it requires redundant execution of the two identical threads to cover a given processor component. Hence, the higher the fault coverage desired, the more redundant the execution required. However, redundant execution inevitably comes at the cost of performance overhead, hardware overhead, design complexity, *etc*. Consequently, how to trade off between fault detection coverage and the associated costs is essential for the applicability of the basic idea. Consider a general five-stage pipeline [1]: the instruction fetch stage, the decode stage, the issue stage, the execution stage, and the retire stage can all be exploited to generate the redundant instructions [29,30,26,4,27,28]. Take the fetch stage as an example, we can generate the redundant threads by fetching instructions twice [26,4,27,28]. Since the instruction fetch stage is the first pipeline stage, the redundant execution would then cover all the pipeline stages, thus, the largest possible pipeline fault coverage could be achieved. However, allowing two redundant threads to fetch instructions would mean halving the effective fetch bandwidth. Furthermore, given N/2 effective fetch bandwidth, the maximum pipeline throughput is limited to N/2. Additionally, the redundant thread generated in the fetch stage would then compete not only for the decode bandwidth, the issue bandwidth, and the retire bandwidth, but also for IssueQ and ROB capacity, which are all identified as the key factors that affect the performance of the redundant execution [30]. Conversely, we can re-issue the retired instructions from ROB back to the functional units for redundant execution. In doing so, the bandwidth and spatial occupancy contention at IssueQ and ROB can be reduced, thus the performance overhead is lowered [30]. However, this retire stage-based design comes at the cost of reducing the fault detection coverage: only the EXE stage is covered.

Hence, considering the trade-off between fault detection coverage and design cost, we simply fetch the instructions once and then copy the fetched instructions to generate the redundant thread, In so doing, there is no need for partitioning the fetch bandwidth between the redundant threads. Furthermore, we rely on the dispatch thread scheduling and redundant thread reduction to relieve the contention in the IssueQ and ROB (details can be found in Subsects. 6.2 and 6.3).

5.2 Possible Occurrence of Deadlocks

Other than the design trade-off, another issue associated with the basic idea is the need to prevent deadlocks. In a fault-tolerant SMT design, two copies of the same thread are now cooperating with each other. Such cooperation could cause deadlock situations [27]. We thus present a systematic deadlock analysis and conclude that as long as ROB, LQ, and SQ have allocated some dedicated entries for the trailing thread, the deadlock situations identified can be prevented. Based on this observation, we propose two ways of preventing any possible deadlock situations (more details in Sect. 7): (1) static allocation of entries in ROB, LQ, and SQ for the redundant thread copy; (2) dynamic deadlock monitoring.

6 Lowering the Performance Overhead of Transient Fault-Tolerant SMT Processors

This section presents our proposed fault-tolerant SMT datapath designs. As discussed, to consider the trade-off between the fault detection coverage and the design costs, we simply fetch the instructions once and then copy the fetched instructions to generate the redundant thread. As far as the fault detection coverage is concerned, there are three major components in the fetch stage: I-cache, PC, and BPUs. By copying fetched instructions to generate the redundant threads, any transient faults which would happen inside the I-Cache might not be detected. However, ECC-like mechanisms that are very effective in handling transient faults in memory structures are now widely implemented in modern microprocessors [7,31,10]. Further, the fault which occurs in the BPUs will have no effect on the ultimate correct program execution [32], whereas the critical component PCs must be protected by ECC-like mechanisms.

6.1 Copy Fetched Instructions to Generate the Redundant Thread

As shown in Fig. 8, our instruction copy operation is simple: simply buffer in two instruction queues the instructions fetched, hence, the copy operation would not increase the pipeline cycle time, nor would another pipeline stage be added. To be specific, each instruction fetched is bound to a sequential number and a unique thread ID. For instructions that are stored in IFQ, the "leading thread" (LT) is used as the thread ID, whereas for those stored in another IFQ, called trace queue (traceQ), the "trailing thread" (TT) is used. It should be noted that traceQ also serves in the two performance overhead lowering techniques which will be described in detail in the following subsections.

6.2 Reducing the Complexity of TT

Focusing on our redundant execution mode, the key factors that affect the performance of redundant execution could be identified as: the bandwidth contention for issue, FUs, and retire operations; and the capacity contention in IssueQ and ROB [33,30]. This subsection addresses those resource contention problems by making TT "lighter" (recall that the purpose of executing TT is just fault detection). In doing so, the contention in IssueQ, ROB and FUs could be reduced accordingly.

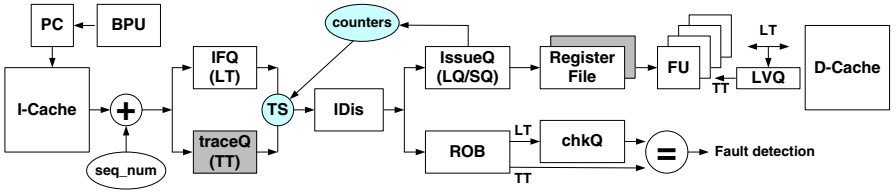


Fig. 8 Functional diagram of our proposed fault-tolerant SMT data path

6.2.1 Preventing the Dispatch of TT Wrong-Path Instructions

The number of dynamic wrong-path instructions might be a significant portion of all the instructions fetched. For example, Kang et al. [34] observed that 16.2–28.8% of the instructions fetched would be discarded from the pipeline even with a high branch prediction accuracy. Hence, if we could prevent TT wrong-path instructions from being dispatched, the effective utilization of IssueQ, ROB, and FUs would be improved accordingly. Based on this observation, we leverage LT branch resolution results to prevent all the wrong-path instructions of TT from being dispatched. It should also be pointed out that in our designs, neither a branch outcome queue [4] nor a branch prediction queue [28] are needed.

If a branch instruction is encountered in traceQ, the dispatch operation will check the status of this TT branch instruction: if its prediction outcome has been resolved by its counterpart from LT, we continue its dispatch operation; otherwise, the TT dispatch operation will be paused (from here, we can see that in order not to pause TT dispatch operation, LT must be executed ahead of TT. The LT ahead of TT execution mode is called the *staggered execution mode* (more on this in Subsect. 6.3.2).

To set the TT branch instruction status (the initial status is set as "unresolved"), every completed branch instruction from LT will search traceQ to match its TT counterpart. Furthermore, if the branch has been mispredicted, LT will perform its usual branch misprediction recovery, and at the same time it will flush all those instructions inside traceQ that are located behind the matched counterpart branch instruction. In other words, LT performs the branch misprediction recovery for both LT and TT, thus, TT does not recover any branch misprediction by itself. After recovery, the status of the TT branch instruction will be set as "resolved." On the other hand, if the branch has been correctly predicted, the status of the matched counterpart TT branch instruction will be set as "resolved" as well.

In addition, the sequential numbers provide the mean for matching two redundant threads instructions.[10] As we have seen, each instruction fetched is associated with a sequential number at first, then the fetched instruction is replicated to generate the redundant thread. In doing this, two copied instructions will have the same sequential numbers in different threads.

6.2.2 Load Value Queue (LVQ) Design

We adopt the LVQ design [4] and include it in our design as shown in Fig. 8. Basically, when an LT load fetches data from the cache (or the main memory), the data fetched and the matching tag associated are also buffered into LVQ. Instead of accessing the memory hierarchy, TT loads simply checks and matches the LVQ for the data fetched. In doing so, the TT might reduce the D-cache miss penalties and then improve its performance. It should be noted here that in order to fully benefit from the LT data

[10] Such a sequential number feature has been implemented, for example, in the Alpha and the PowerPC processors. Indeed, the Alpha21264 manual states "*... also, each instruction is assigned a unique 8-bit number, called an inum, which is used to identify the instruction and its program order with respect to other instructions during the time that it is in flight.*" [35].

prefetching, we must guarantee that LT is always ahead of TT, which requires a staggered execution mode.

6.3 Dispatch Thread Scheduling

It is well known that there are many idle slots in the execution pipeline. Hence, we must design for the redundant execution to exploit those idle slots as much as possible in order to circumvent the contention which would otherwise degrade performance [33,30].

6.3.1 Modified ICOUNT Policy Applied at the Dispatch Stage

To exploit the idle slots, we must ensure that when one thread is idle for some reason, the execution resources can be *promptly* allocated to another thread which can utilize those resources more efficiently. The ICOUNT policy was proposed to schedule threads in order to fill IssueQ with issuable instructions, *i.e.*, restrict threads from clogging IssueQ [36]. However, we argue that it is the dispatch stage that directly feeds IssueQ with useful instructions, hence, scheduling threads in the dispatch stage level would cause a prompter reaction to the thread idleness in IssueQ. Therefore, we modify the ICOUNT policy as follows: as illustrated in Fig. 8, at each clock cycle, we count the number of instructions that are still waiting in IssueQ from LT and TT individually. A higher dispatch priority is assigned to the thread with the lower instruction count. To be specific, when the dispatch rate is eight instructions per cycle, the selected thread is allowed to dispatch as many instructions as possible (up to eight). If any dispatch slot is left from the selected thread, the alternative thread would consume the remaining slots. We call the above policy "ICOUNT.2.8.dispatch."

6.3.2 Staggering Execution

Having shown how to reduce the complexity of TT and made it "light," a staggered execution mode [4,28] is required. To this end, we developed the "slack dispatch" scheme: in the instruction dispatch stage, if the selected thread is TT, we check the instruction distance between LT and TT. If the distance is less than a predefined threshold, we skip the TT dispatch operation and continue buffering TT in traceQ. This means that the size of traceQ (the number of entries in traceQ) must meet the following requirement:

$$\text{size of (traceQ)} > \text{size of (IFQ)} + \text{predefined distance} \tag{12}$$

Moreover, for purposes of fault-detection, all retired LT instructions and their execution results are buffered into the checking queue (chkQ), as shown in Fig. 8. Hence, TT is responsible for triggering the result comparison.[11] We further assume that the

[11] The high chkQ bandwidth requirement could demand some special attention. To address this, we would apply an approach similar to that shown in [28].

register file of TT is protected by ECC-like mechanisms. This means that, if any fault is detected, the register file state of TT could be used to recover the LT.

7 Deadlock Analysis and Prevention

As pointed out before, the two copies of a thread cooperate with each other for fault checking and recovery. However, if not carefully synchronized, such cooperation could result in deadlocks where neither copy could make any progress [27]. To prevent such situations, a detailed analysis and the introduction of appropriate synchronization mechanisms are necessary.

7.1 Deadlock Analysis

In general, resource sharing is one of the underlying causes of deadlocks [37]. In our case, there is much resource sharing between the two thread copies. For example, IssueQ is a shared hardware resource and both thread copies contend for it. The availability of instructions being issued is another type of resource sharing: the issue bandwidth is dynamically partitioned between the two thread copies.

Figure 9 shows the resource allocation graph. TraceQ is a resource which is requested only by LT but is owned only by TT: LT needs entries in traceQ for its instruction fetching and generation of the redundant thread (see Subsect. 6.1). In contrast, chkQ is different in the following sense: only if there is a free entry in chkQ could LT retire its instruction and back up the retiring instructions and execution results there. On the other hand, the entry in chkQ can only be freed by TT: only after an instruction has been retired and compared, can the corresponding entry in chkQ be released. Further, due to the similarity between dispatch and issue operations, we combine them under the term "*issue resource*" in the discussion which follows.

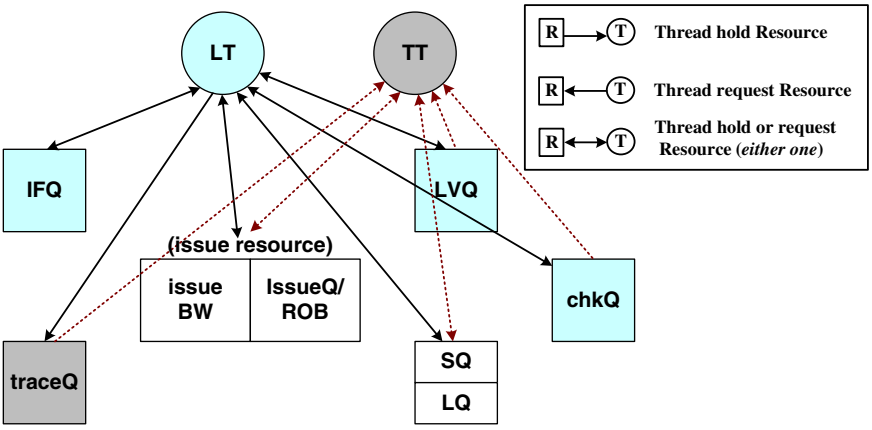


Fig. 9 Resource allocation graph for deadlock analysis

Based on Fig. 9, we can list all possible circular wait conditions. However, some conditions obviously do not end up in a deadlock, for example, "LT → traceQ → TT → SQ → LT." All the possible deadlock scenarios are listed as follows:

1. LT → chkQ → TT → issue resource → LT.

   Scenario: When chkQ is full, LT cannot retire its instructions. Then, those instructions which are ready to retire from LT are simply stalled in ROB. If this stalling results in the ROB being full of instructions from LT (note that the situation where the ROB is full of instructions from LT could be exacerbated by the fact that LT is favored by the dispatch thread scheduling policy for the stagger execution mode), the instruction dispatch operation will be blocked, thus, TT will be stalled in traceQ. Consequently, no corresponding instructions from TT can catch up to release the chkQ entries and then a deadlock can happen. In summary, the condition for this deadlock situation is the following:

   Observation 1 *When chkQ is full and ROB is full of instructions from LT, a deadlock happens.*

2. LT → LVQ → TT → issue resource →LT.

   Scenario: When LVQ is full, a load instruction from LT cannot proceed with its execution (an LT load instruction must guarantee to successfully buffer loaded data into LVQ to complete its execution). On the one hand, the stalled load instruction could result in the LQ being full of instructions from LT since LT is favored for dispatching. Further, a full LQ will block the instruction dispatch operation, thus, the corresponding load instructions from TT will be unable to catch up to release the LVQ entries and then a deadlock can happen. Hence, a deadlock observation follows:

   Observation 2 *When LVQ is full and LQ is full of instructions from LT, a deadlock happens.*

   On the other hand, the stalled load instruction could result in a full ROB, thus, the instruction dispatch operation will be blocked. Furthermore, if there is no load instruction from TT in the full ROB, *i.e.*, no load instruction from TT in LQ, no corresponding load instructions from TT will be able to catch up to release the LVQ entries and then a deadlock can happen.

   Observation 3 *When LVQ is full and ROB is full and there is no load instruction from TT in ROB, a deadlock situation takes place.*

3. LT → SQ →TT → issue resource → LT.

   Scenario: When SQ is full of instructions from LT, the instruction dispatch operation will be blocked (worse, the stalled store instructions would result in a full ROB which also blocks the dispatch operation). Consequently, no corresponding store

instructions from TT can catch up to release the SQ entries and then a deadlock could happen.[12]

Observation 4 *When SQ is full of instructions from LT, a deadlock happens.*

7.2 Deadlock Prevention

Based on the above systematic deadlock analysis, we propose two mechanisms to handle all possible deadlock situations: static hardware resource partitioning and dynamic deadlock monitoring.

7.2.1 Static Hardware Resource Partitioning

In static hardware resource partitioning, *i.e.*, each thread has itself allocated resources, the deadlock conditions identified can be broken such that we can prevent them altogether.[13] For example, we can partition the ROB in order to prevent the possible deadlock situation identified in Observation 1: if some entries of ROB are reserved for TT, TT dispatch operations could continue since when chkQ is full, the partitioned ROB cannot be full of instructions from LT. Subsequently, those dispatched TT instructions will be issued and their execution completed afterwards. After completion, they will trigger the result comparison and then free the corresponding chkQ entries if the operation was found to be fault-free. After some chkQ entries have been freed, LT is allowed to proceed.

Moreover, we find that only three hardware resources (ROB, LQ, and SQ) need to be partitioned in order to prevent all the deadlock situations we identified:

- Partitioning the ROB to break the deadlock situation identified in Observation 1: ROB will never be full of instructions from LT such that TT will be dispatched and then chkQ will be released.
- Similarly, partitioning LQ to break the deadlock situation identified in Observation 2;
- Partitioning SQ to break the deadlock situation identified in Observation 4.

Figure 10 illustrates how the static three-queue partitioning mechanism prevents the deadlock situation identified in Observation 3. When LVQ is full, the load instruction k in LQ of LT cannot be be issued. However, since ROB is now partitioned between LT and TT, the stalled load instruction k in ROB will only block LT from being dispatched. In other words, the TT dispatch operation will not be blocked by the stalled load instruction k, thus, for example, the load instruction i from TT will be dispatched which will then release the LVQ entry occupied by the counterpart load instruction i from LT. Once free LVQ entries are made available, the stalled LT load instruction k can be issued. In summary, we have the following:

[12] Since there is only one memory space for both LT and TT in our design, the store instructions can be released to the memory only after being found to be fault-free [4,27]. Hence, any LT store instruction ready to be retired needs to be continually buffered until TT compares the data stored and the reference address.

[13] The static hardware partitioning approach is similar to the deadlock avoidance designs in [27]. However, based on our systematic deadlock analysis, we clarify the implementation details of the basic approach.

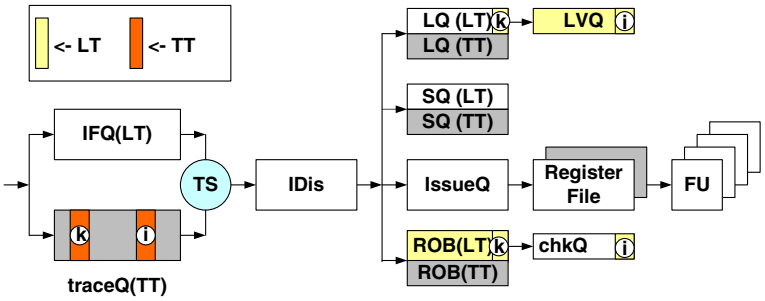
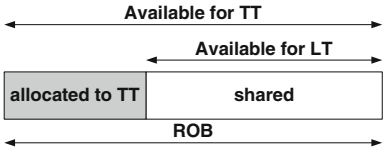


Fig. 10 Exemplified static hardware resource partitioning for deadlock prevention

Fig. 11 Static ROB partitioning mechanisms



Algorithm 2: Dispatch thread scheduling policy (with dynamic deadlock monitoring)

Apply ICOUNT.2.8.dispatch policy;
if *selected thread is LT* ***AND*** *IFQ not empty* ***AND*** *no precaution signal* **then**
    Dispatch from IFQ;
else if *distance between LT and TT meets predefined stagger execution mode* ***AND*** *traceQ is not empty* ***AND*** *not an unresolved branch instruction* **then**
    Dispatch from traceQ;
else
    nothing to be dispatched;
end

Observation 5 *For each ROB, LQ, and SQ, allocating some dedicated entries for TT will prevent the deadlock situations identified from occurring.*

It should be noted, however, that static hardware resource partitioning has some performance impact on an SMT processor, particularly when partitioning ROB, LQ, and SQ (instruction issue queues for load and store instructions) [38]. Figure 11 shows how the static partitioning is implemented: we allocate a minimum number of entries for TT to prevent deadlocks and the remainder of the queue is shared between LT and TT. Hence, the maximum available entry number for LT is the total number of queue entries minus the number of reserved entries whereas the maximum number of available entries for TT is the total number of queue entries.

7.2.2 Dynamic Deadlock Monitoring

From our deadlock analysis, we also conclude that if we can dynamically regulate the LT progress such that neither ROB nor LQ and SQ could be filled with instructions

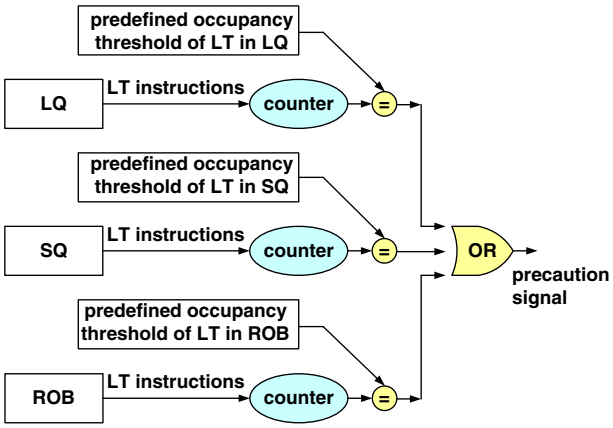


Fig. 12 Dynamic monitoring for deadlock prevention

from LT, the deadlock situations identified can be prevented. As illustrated in Fig. 12, instead of statically partitioning hardware resources, we dynamically count the instruction numbers from LT in ROB, LQ, and SQ, respectively, and then a caution signal is generated if at least one of the numbers of instructions counted exceeds the corresponding predefined occupancy threshold. Furthermore, if the caution signal is generated, the dispatch thread scheduling policy will pause LT from being dispatched.

More specifically, the comprehensive dispatch thread scheduling policy we developed is listed in Algorithm 2: first, we apply the ICOUNT.2.8. dispatch policy. If the selected thread is LT, we must then check whether IFQ is empty since no instruction can be dispatched in case of an empty IFQ. Furthermore, we need to make sure no precaution signal has been generated. If there is a such signal, we must stop dispatching from LT. On the other hand, if the selected thread is TT, we check the following conditions before dispatching TT:

- the stagger execution mode requirement;
- is traceQ not empty;
- are we not encountering an unresolved branch instruction.

It should be noted that compared with static resource partitioning, the dynamic deadlock monitoring approach comes with a higher design flexibility: by adjusting the predefined occupancy thresholds, we can manipulate the resource allocation between the cooperating threads. However, this flexibility comes at the cost of additional hardware: counters and additional logic, as well as a a more complicated thread scheduling policy.

8 Design Overhead Evaluation of Transient Fault-Tolerant SMT Processors

This section will present our fault-tolerant design overhead evaluation methodology and results. Our main purpose is to present performance numbers and sizing numbers of traceQ and chkQ for the schemes presented in Sect. 6 and Sect. 7.

Table 3 Key design differences between two simulated FT-SMT processors

	SRT-like design	Our design
Redundant thread generation	Fetch twice	Copy fetched instructions
Thread scheduling policy	ICOUNT.2.8	ICOUNT.2.8.dispatch
Deadlock handling	Static hardware partitioning	Dynamic deadlock monitoring
Eliminate TT mispredicted instructions	Using branch outcome queue to prevent being fetched	Using dispatch thread scheduling to prevent being dispatched
Stagger execution mode, chkQ, and LVQ	Implemented	Implemented

Table 4 Simulated FT-SMT processor configuration

Component	Configuration
Processor	2-way SMT, 8-way out-of-order issue, 32-entry IFQ, 256-entry trace queue (256-entry IFQ when simulating superscalar), 64-entry issue queue,128-entry ROB,64-entry load queue, 64-entry store queue, 64-entry LVQ, 256-entry chkQ
Functional unit	6 Int ALU, 2 Int Multiply, 4 FP Add, 2 FP Multiply
Branch predictor	Hybrid: 8K-entry bimodal, 8K-entry gshare, 8K-entry 2-bit selector, 16-entry RAS, 4-way 1K BTB, 10-cycle misprdiction penalty
L1 I- and D-cache	64KB, 32-byte block size, 4-way associative, 1-cycle hit latency
Unified L2 cache	1MB, 64-byte block size, 4-way associative, 6-cycle hit latency
Main memory	100-cycle latency

8.1 Performance Overhead Evaluation of Two Fault-Tolerant SMT Processor Designs

We now examine the performance overhead of a fault-tolerant SMT processor.[14] To this end, we have developed an SMT performance simulator based on the SimpleScalar toolset [39]. Moreover, two fault-tolerant SMT processor designs have been simulated. Table 3 shows the key design differences between the two designs: the first one is similar to that discussed by Reinhardt et al. in [4] whereas our design has been presented in Sects. 6 and 7. From a performance overhead viewpoint, the major differences are the way of generating redundant threads and the thread scheduling policy: an SRT-like design fetches a thread twice for redundant execution, adopts an ICOUNT policy [36], and applies the policy in the instruction fetch stage whereas our design copies the fetch instructions to generate the redundant thread, adopts a modified ICOUNT policy, and applies this in the instruction dispatch stage (details in Subsect. 6.3.1). Table 4 summarizes the key parameters of the processors simulated. The others microarchitectural configuration parameters such as fetch bandwidth, function units, L1 I- and D- caches, unified L2-cache, and main memory, are similar to the ones used in [4].

We evaluated our designs using 13 SPEC benchmarks shown in Table 5. Those SPEC benchmarks characteristics and their IPCs in the superscalar execution mode

[14] Note that we are simulating the performance overhead in fault-free cases.

Table 5 SPEC benchmarks, input data sets and IPCs in superscalar execution mode

Benchmark	Version	Input	IPC
099.go	CINT95	test	1.11
132.ijpeg	CINT95	test	2.31
164.gzip	CINT2000	lgred.graphic	1.84
197.parser	CINT2000	lgred.in	1.51
171.swim	CFP2000	test	3.0
179.art	CFP2000	lgred.in	0.72
256.bzip2	CINT2000	lgred.source	2.16
130.li	CINT95	test	1.88
134.perl	CINT95	lgred.makerand.pl	1.87
175.vpr	CINT2000	lgred	1.65
101.tomcatv	CFP95	test	2.64
177.mesa	CFP2000	lgred.in	3.40
183.equake	CFP2000	lgred.in	1.70

are also shown. Furthermore, those input data sets that are not marked as "test" are from MinneSPEC [40], whereas those marked as "test" input data sets have been obtained directly from SPEC. In addition, we used the compiler optimization level "-O2" to compile the SPEC benchmarks executables. Moreover, when simulating, we fast-forwarded past the first 300 million instructions (during that time, the IPC is extremely unstable) for each thread and then simulated all the remaining instructions for each threads.

Moreover, we need to point out that using IPC to evaluate the fault-tolerant SMT performance is appropriate: in the fault-tolerant multi-threading execution mode, two copies of the same thread actually carry only a single computation task, *i.e.*, from the user's viewpoint, they function as if only one thread was executing. On the contrary, in the conventional multi-threading execution mode, two threads are typically independent, hence the executions carry out two tasks. In this sense, the fault-tolerant multi-threading which is running two copies of the same thread is functionally comparable to the superscalar where IPC is conventionally used to evaluate the performance. Furthermore, in our fault-tolerant multi-threading simulations, we computed IPCs as the *retired and then result-compared* instructions numbers from the leading thread divided by the total simulated clock cycle numbers.

The simulation results are shown in Fig. 13. Compared with the superscalar case (without fault-tolerance features), the performance overhead of our design is lowered by only 12% on average whereas an SRT-like design has 17% overhead. Further, if we choose the baseline as the two-way SMT without any fault-tolerance features (the baseline SMT processor is executing two identical threads), the performance overhead is 34% on average whereas the overhead of SRT-like design is 42%. Also, note that in Fig. 13, there are benchmarks that show that the IPCs of "superscalar" (without FT) is lower than those of "our design." [4] reported similar cases that for some benchmarks, the fault-tolerant SMT implementations displayed higher throughput than the conventional SMT without the fault-tolerant features. The authors suggested that the reason for this apparently odd behavior could be the branch target buffer prefetching that limited the TT branch misprediction penalties. In our designs, there are two reasons: first,

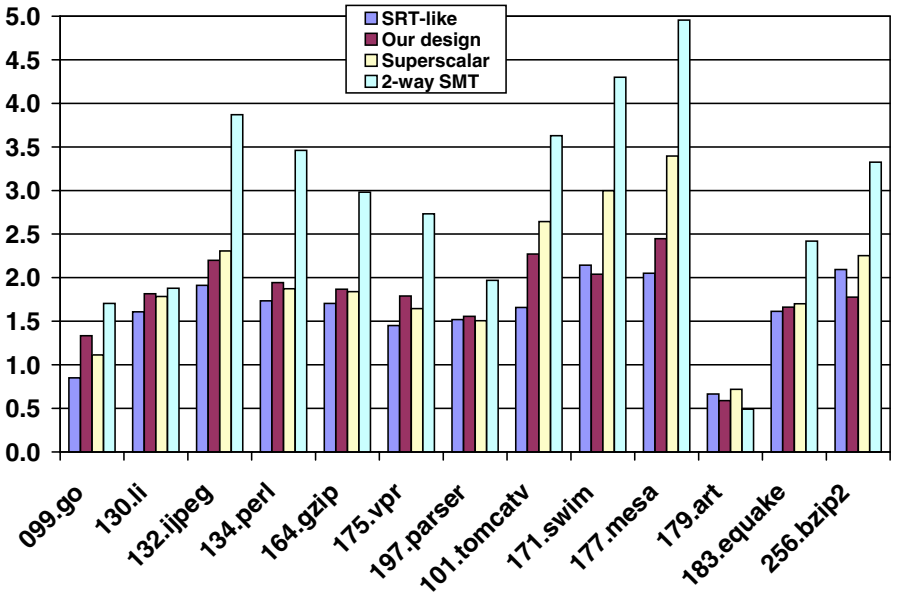


Fig. 13 IPC comparison of two FT-SMT processors, superscalar, and two-way SMT

because of the LVQ design, the LT prefetches the data for the TT, which improves TT execution. However, the data prefetching has not been supported in "superscalar" and "2-way SMT." Secondly, because of the copying of fetched instructions to generate the redundant thread, only LT needs to access the instruction caches and experience instruction cache misses. However, in conventional designs such as "superscalar" and "2-way SMT," all active threads fetch instructions on their own.

8.2 Hardware Overhead Evaluation

This subsection examines two queues which are introduced for the fault-tolerant schemes we have designed: traceQ and chkQ. We want to evaluate how their sizes affect the fault-tolerant performance overhead. To this end, using the performance simulators we developed, we varied the size of the queue and then observed the pipeline throughput to determine the optimal design points.

8.2.1 Sizing TraceQ

TraceQ functions as a buffer holding the TT instructions to support the staggered execution mode. However, from Eq. 12, we know that to support the staggered execution mode, the minimum size of traceQ must be larger than the size of IFQ plus the predefined instruction distance. Hence when choosing the size of traceQ as 32-entry, the staggered execution mode is not supported whereas when traceQ is a 64-entry or 128-entry unit, the predefined instruction distance is 32-instruction. Likewise, when

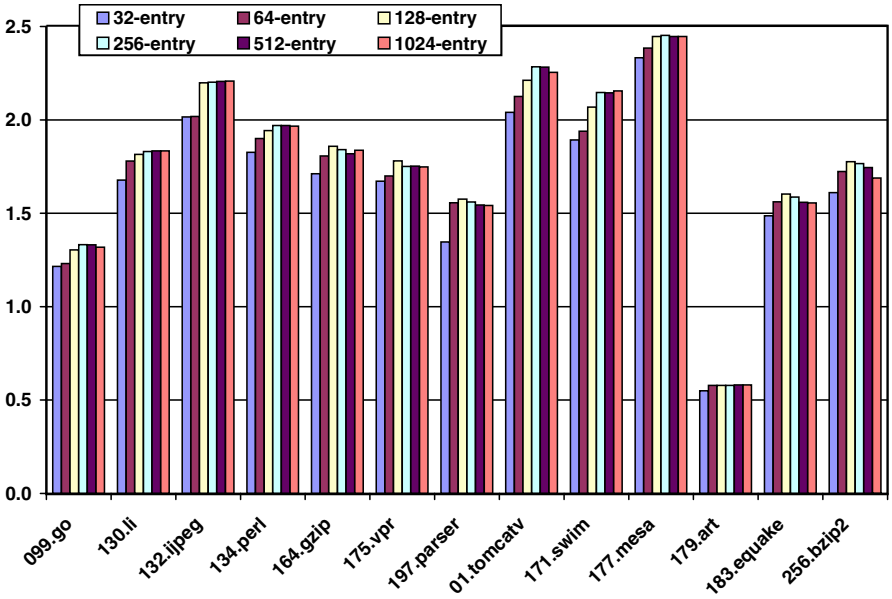


Fig. 14 IPC comparison of varying size of traceQ (IFQ: 32-entry, chkQ: 256-entry)

traceQ has more than 256 entries, the predefined instruction distance is 128 instructions. Figure 14 shows the simulation results. Obviously, when traceQ is small, the benefit of the staggered execution mode cannot be fully exploited, and the IPC is degraded: compared with a 256-entry traceQ, IPC in those cases is lowered by 9% on average. On the other hand, if we continued increasing traceQ, the IPC improvement would be limited: on average, there would be no significant IPC improvement. Therefore, we choose a 256-entry traceQ as our design point. Furthermore, each entry of traceQ holds an instruction word (32-bit as typical size), a thread ID (1-bit), and a branch resolve status bit (1-bit). Hence, the storage core of traceQ is : $256 \times (32+1+1) = 1088$ bytes.

8.2.2 Sizing chkQ

Figure 15 shows the simulation results when we vary the size of chkQ, given the size of traceQ is 256-entry and the predefined instruction distance is 128-instruction. We can see that increasing the size of chkQ has diminishing throughput return: beyond a 64-entry chkQ, no significant improvement can be observed. The reason for that is because chkQ is buffering those retired (but waiting for comparison) LT instructions and because TT is reduced such that TT can catch up with LT more after being dispatched. Hence, chkQ is not frequently full. In other words, chkQ can be made to be smaller than traceQ. Hence, we choose a 64-entry chkQ. In addition, it should be noted that each entry of chkQ holds sequential numbers (10-bit, for instance), instruction

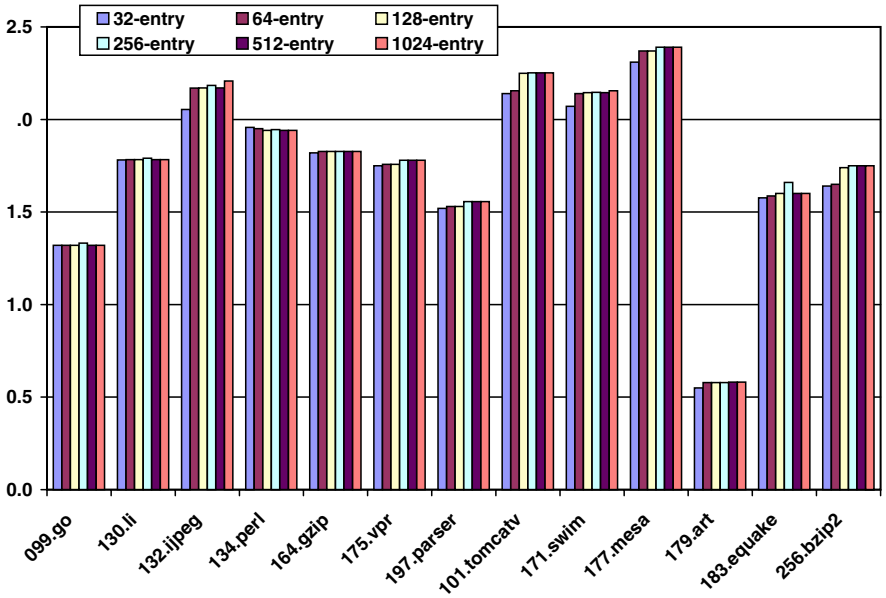


Fig. 15 IPC comparison of varying size of chkQ (traceQ: 256-entry, predefined instruction distance: 128-instruction)

word (32-bit), execution results (32-bit, for instance), and status bit (1-bit). Hence, the storage core of chkQ is: $64 \times (10 + 32 + 32 + 1) = 600$ bytes.

9 Conclusions

We have presented two architecture mechanisms to protect microprocessors against the effect of radiation. First, we have observed that many faults manifest themselves as control flow errors that cause the processor to violate the correct sequencing of instructions. To detect such control flow errors, we started from a basic compile-time signature assignment algorithm. Then, approaches to improving the fault detection coverage of the basic algorithm were discussed. Moreover, to allow the processor to efficiently check the run-time sequence and detect control flow errors, an on-chip assigned-signature checker was designed: this checker is capable of executing three additional instructions (SIC, SIJ, SIJC). The algorithm of embedding three additional instructions into programs were also described.

Second, we realized that since the very concept of SMT inherently provides redundancy, we could run two copies of the same thread on top of SMT platforms in order to detect and correct soft errors. This allows, upon detection of an error, the rolling back of the processor state to a known safe point, and then a retry of the instructions, thereby effecting a completely error-free execution. We have discussed two crucial implementation issues introduced by this basic scheme: (1) the design trade-off between the fault detection coverage versus design costs; (2) the possible occurrence of deadlock situations.

Algorithm 3: Embedding three additional instructions into programs

```
Derive CFG of the given program;
Assuming we have nodes: V_i (i=1,2,...,N) where N is the total number of nodes in the CFG;
Assign a unique state code D(i) to every node V_i;
Apply Two-Pass primary node selection algorithm;
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) ⊕ 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
        /*V_i is an MBI node                                                    */
        (assume V_j is selected as its primary node);
        Assign the reference signature of V_i as: S(i) = D(i) ⊕ 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 ∈ 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 → V_i is a taken path then
                imm1=X, imm2=D(k) ⊕ D(j);
            else if V_k → V_i is a not-taken path then
                imm1=D(k) ⊕ D(j), imm2=X;
            else
                /*V_k → V_i is not a conditional branch path                    */
                imm1=D(k) ⊕ D(j), imm2=X;
            end
        end
    end
end
```

To achieve the largest possible fault detection coverage, we replicate the instructions fetched so as to generate the redundant thread copies. Further, we apply the SMT thread scheduling at the level of the instruction dispatch stage to lower the performance overhead. As a result, when compared to the baseline processor, our simulation results show that by using our two new schemes, the performance overhead can be reduced down to 34% on average, down from 42%. Finally, in the fault-tolerant execution mode, since the two copied threads are cooperating with each other, deadlock situations could be quite common. We thus presented a detailed deadlock analysis and then concluded that allocating some entries of ROB, LQ, and SQ for the trailing thread would be sufficient to prevent such deadlocks.

In terms of future research, we would like to design a simulation tool which would allow the random injection of faults into the microprocessor model to evaluate the effectiveness of our proposed protection schemes.

References

1. Hennessy, J.L., Patterson, D.A.: Computer architecture: a quantitative approach. 3rd 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 (1993)
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 Comput. (1999)
6. Stackhouse, B., Bhimji, S., et al.: A 65 nm 2-billion transistor quad-core itanium processor. IEEE Trans. Solid-State Circuits (2009)
7. Quach, N.: High Availability and reliability in the itanium processor. IEEE Micro. (2000)
8. Sanda, P.N., Kellington, J.W., Kudva, P., Kalla, R., McBeth, R.B., Ackaret, J., Lockwood, R., Schumann, J., Jones, C.R.: Soft-error resilience of the IBM POWER6 processor. IBM J. Res. Dev. (2008)
9. Clarke, W.J., Alves, L.C., Dell, T.J., Elfering, H., Kubala, J.P., Lin, C., Mueller, M.J., Werner, K.: IBM System z10 design for RAS. IBM J. Res. Dev. (2009)
10. 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 J. Solid-State Circuits (2003)
11. Intel Corporation, (Santa Clara): IA-32 intel architecture software developer's manuals (2006)
12. Wilken, K., Shen, J.P.: Continuous signature monitoring: low-cost concurrent-detection of processor control errors. IEEE Trans. Comput. Aided Des. (1990)
13. Ohlsson, J., Rimén, 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)
14. Schuette, M.A., Shen, J.P.: Processor control flow monitoring using signatured instruction streams. IEEE Trans. Comput. (1987)
15. Mohmood, A., McCluskey, E.J.: Concurrent error detection using watchdog processors—a survey. IEEE Trans. Comput. (1988)
16. Schuette, M.A., Shen, J.P.: Exploiting instruction-level parallelism for integrated control-flow checking. IEEE Trans. Comput. (1994)
17. Warter, N.J., Hwu, W.-M.W.: A software based approach to achieving optimal performance for signature control flow checking. 20th international symposium on fault-tolerant computing (1990)
18. 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)
19. Alkhalifa, Z., Nair, S., Krishnamurthy, N., Abraham, J.A.: Design and evaluation of system-level checks for on-line control flow error detection. IEEE Trans. Parallel Distrib. Syst. (1999)
20. Shirvani, P.P., McCluskey, E.J.: Fault-tolerant systems in a space environment: The CRC ARGOS Project. Tech. Rep. CRC-TR 98-2, Stanford University (1998)
21. Bagchi, S., Srinivasan, B., Whisnant, K., Kalbarczyk, Z., Iyer, R.K.: Hierarchical error detection in a software implemented fault tolerance (SIFT) environment. IEEE Trans. Knowl. Data Eng. (2000)
22. Oh, N., Shirvani, P.P., McCluskey, E.J.: Control-flow checking by software signatures. IEEE Trans. Reliab. (2002)
23. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, techniques, and tools. Addison-Wesley Publishing Company, Wokingham, UK (1986)
24. Borin, E., Wang, C., Wu, Y., Araujo, G.: Dynamic binary control-flow errors detecttion. ACM SIGARCH Computer Architecture News (2005)
25. Saxena, N.R., McCluskey, E.J. Dependable adaptive computing systems- the ROAR project. In: 1998 IEEE international conference on systems, man and cybernetics (1998)
26. Rotenberg, E.: AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. In: 29th international symposium on fault-tolerant computing (1999)
27. Mukherjee, S.S., Kontz, M., Reinhardt, S.K.: Detailed design and evaluation of redundant multithreading alternatives. In: 29th international symposium on computer architecture (2002)
28. Vijaykumar, T.N., Pomeranz, I., Cheng, K.: Transient-fault recovery using simultaneous multithreading. In: 29th international symposium on computer architecture (2002)
29. Ray, J., Hoe, J.C., Falsafi, B.: Dual use of superscalar datapath for transient-fault detection and recovery. In: 34th international symposium on microarchitecture (2001)

30. Smolens, J.C., Kim, J., Hoe, J.C., Falsafi, B.: Efficient resource sharing in concurrent error detecting superscalar microarchitectures. In: 37th international symposium on microarchitecture (2004)
31. Bossen, D.C., Tendler, J.M., Reick, K.: Power4 system design for high reliability. IEEE Micro. (2002)
32. Mukherjee, S.S., Weaver, C., Emer, J., Reinhardt, S.K., Austin, T.: A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In: 36th international symposium on microarchitecture (2003)
33. Mendelson, A., Suri, N.: Designing high-performance & reliable superscalar architectures the out of order reliable superscalar (O3RS) approach. In: International conference on dependable systems and networks (2000)
34. Kang, D., Gaudiot, J.-L.: Speculation control for simultaneous multithreading. In: 18th international parallel and distributed processing symposium (2004)
35. Compaq Computer Co., Massachusetts: Alpha 21264/EV68CB and 21264/EV68DC Hardware Reference Manual, 1.1 ed. (2001)
36. Tullsen, D.M., Eggers, S.J., Levy, H.M.: Simultaneous multithreading: maximizing on-chip parallelism. In: 22nd international symposium on computer architecture (1995)
37. Silberschatz, A., Galvin, P.B., Gagne, G.: Applied operating system concepts. 1st edn. John Wiley & Sons, Inc. (2000)
38. Raasch, S.E., Reinhardt, S.K.: The impact of resource partitioning on SMT processors. In: 12th international conference on parallel architectures and compilation techniques (2003)
39. Burger, D., Austin, T.M.: The SimpleScalar Tool Set, Version 2.0. Tech. Rep. 1342, University of Wisconsin-Madison Computer Sciences Department (1997)
40. KleinOsowski, A., Lilja D.J.: MinneSPEC: a new SPEC benchmark workload for simulation-based computer architecture research. Tech. Rep. ARCTiC Lab No. 02–08, University of Minnesota, Minneapolis (2002)