Software-Based Transparent and Comprehensive Control-Flow Error Detection

Edson Borin‡† Cheng Wang† Youfeng Wu† Guido Araujo‡
‡ IC-UNICAMP - Brazil †PSL-Intel Corporation – USA
{borin,guido}@ic.unicamp.br, {cheng.wang,youfeng.wu}@intel.com

Abstract

Shrinking microprocessor feature size and growing transistor density may increase the soft-error rates to unacceptable levels in the near future. While reliable systems typically employ hardware techniques to address soft-errors, software-based techniques can provide a less expensive and more flexible alternative. This paper presents a control-flow error classification and proposes two new software-based comprehensive control-flow error detection techniques. The new techniques are better than the previous ones in the sense that they detect errors in all the branch-error categories. We implemented the techniques in our dynamic binary translator so that the techniques can be applied to existing x86 binaries transparently. We compared our new techniques with the previous ones and we show that our methods cover more errors while has similar performance overhead.

1. Introduction

Transient faults, also called *soft-errors* or single-event upsets (SEUs), are intermittent faults that do not occur consistently. Generally, these faults are caused by external events such as neutron and alpha particles striking the chip or power supply and interconnect noise [5]. Although these faults do not cause permanent damage, it may result in incorrect program execution by altering signal transfers or stored values.

High-availability systems and safety-critical applications, such as spacecraft, airplanes and automotive control, are very sensitive to errors. Faulty behavior in these systems may lead to injury or damage to property. Therefore they must be reliable, and are generally designed to tolerate faults. Although soft-errors have been mainly addressed in this domain, new trends in general purpose microprocessor manufacturing have pushed these faults under the spot light.

Transistors are becoming increasingly smaller, faster, and with tighter noise margins, which make processors more susceptible to soft-errors [14]. In fact, soft-errors are already changing the way industry looks at processor design. Sun Microsystems lost a major customer to IBM due to server crashes caused by soft-errors [3]; and the fear of cosmic ray strikes led Fujitsu to use some form of error detection [2] to protect 80 percent of the 200,000 latches in its recent Sparc processor.

Most modern microprocessors already incorporate certain mechanisms for detecting soft-errors. Memory elements, particularly caches, are protected by mechanisms such as error-correcting codes (ECC) and parity. The protection is typically focused on memory because the techniques are well understood and do not require expensive extra circuitry. Moreover, caches take up a large part of the chip area in modern microprocessors.

Recent studies [14] show that in the near future the soft-error rate in combinational logic will be comparable to that of memory elements, and protecting the entire chip, instead of only the memory elements, will be in the top of the designers to do list.

Several works have investigated redundancy techniques to provide soft-errors reliability in combinational logic [9, 10, and 15]. Hardware based approaches generally rely on inserting redundant hardware such as duplicating functional units or even the entire processor. As an alternative to hardware approaches, software-based techniques change only the software, and the reliability comes *free of cost* (except for the performance loss). Moreover, the software-only approach can be applied to off-the-shelf processors.

In software-based techniques, the reliability is generally achieved by combining *data-flow* and *control-flow checking techniques*. Data-flow checking techniques rely on redundant computation by replicating instructions. On the other hand, control-flow checking is usually performed by comparing a run-time signature with a pre-computed one.

This paper focus on control-flow checking to detect a special class of errors called *control-flow errors*, which occur when a processor jumps to an incorrect next instruction due to a soft-error. We propose a control-flow error classification and two software-based comprehensive control-flow checking techniques that can detect all the errors in the classification. We also discuss variations of the techniques, which trade off performance and delay to report the fault. The techniques are implemented in our dynamic binary translator, which innovate by allowing legacy code to make transparent use of software-based reliability techniques.

Section 2 depicts the control-flow error classification. Section 3 describes related works and the new control-flow checking techniques. Section 4 formalizes the control flow checking problem and provides a formal proof that our control flow checking techniques can detect any single control-flow error. Section 5 presents the dynamic binary translator and the implementation of the control-flow checking techniques. Section 6 shows the experimental results, and Section 7 concludes the paper.

2. Control-Flow Errors

A control-flow error is a deviation from the program's normal instruction execution flow. This error can be a result of a fault in the target address, branch flags, or even a change in the instruction pointer (IP) register due to external interference [11]. We classify the control-flow errors into two main categories:

- *Branch-error*: when the error occurs in a branch instruction (mistaken branch, or branch to a random address, due to an error in the branch flag or in the target address). Although the error occurs at the branch instruction, it could be caused by instructions executed earlier than the branch instruction, such as instructions that generate the flags which affect the branch instruction.
- *Non-branch-error*: when the error occurs in a non-branch instruction due to a change in the instruction behavior or in the IP register.

Some non-branch-errors are very hard to cover with software-based control-flow reliability techniques. For example, take the instruction: a=a+c. If after executing this instruction the IP points to the same instruction and executes it again, the fault generates an error. Data-flow check techniques can detect this error by comparing the value of "a" with a redundantly computed value, but we cannot detect it by checking only the control-flow. Therefore, we will assume that these errors can be detected by data-flow checking or other means, and concentrate our efforts on branch-errors.

When a fault occurs in a branch instruction, the control-flow can be deviated to any point in memory. Most of modern processors have memory access protection mechanisms that can detect when the processor tries to execute instructions from non-code regions. The *execute disable bit* is an example of such feature in the new generations of Intel processors. Therefore, jumps to memory regions that do not contain code can be detected by the hardware and handled by the operation system. Self modifying code can also be an issue, but we will see that our dynamic binary translator can easily handle it.

On the other hand, control-flow deviations that reach the application code are not detected by these memory protection mechanisms; and these errors can produce silent data corruption (SDC) [8], which can cause the program to generate a wrong output.

In order to evaluate the error coverage of software-based control-flow error checking techniques, we propose a branch-error classification, where we divide the errors into six categories. Figure 1 depicts the branch-errors categories in a control-flow graph. The solid lines show the correct control-flows and the dashed lines represent the different categories of branch-errors.

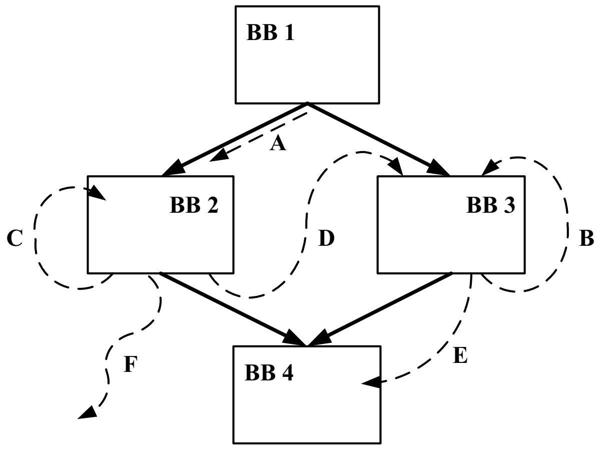


Figure 1 - Branch-error categories. A: mistaken branches. B: Jump to the beginning of the same basic block. C: Jump to the middle of the same basic block. D: Jump to the beginning of other basic block. E: Jump to the middle of other basic block. F: Jump to a non-code memory region.

The branch-error categories in Figure 1 are classified according to the target. Category A represents the mistaken branches, in other words, the errors occur when the branch was supposed to jump, but falls through (or vice-versa). Categories B and C are characterized by errors that change the control-flow to the same basic block of the branch instruction;

Branch-error Category	SPEC-Int 2000					SPEC-Fp 2000				
	Taken		*Not taken*		Total	*Taken*		*Not taken*		Total
	Addr.	*Flags*	*Addr.*	*Flags*		*Addr.*	*Flags*	*Addr.*	*Flags*	
A	0.11%	1.75%	0.00%	2.74%	4.60%	0.15%	3.65%	0.00%	1.83%	5.63%
B	0.09%	0.00%	0.00%	0.00%	0.09%	0.01%	0.00%	0.00%	0.00%	0.01%
C	0.49%	0.00%	0.00%	0.00%	0.49%	5.52%	0.00%	0.00%	0.00%	5.52%
D	0.90%	0.00%	0.00%	0.00%	0.90%	0.49%	0.00%	0.00%	0.00%	0.49%
E	16.13%	0.00%	0.00%	0.00%	16.13%	20.84%	0.00%	0.00%	0.00%	20.84%
F	16.23%	0.00%	0.00%	0.00%	16.23%	28.46%	0.00%	0.00%	0.00%	28.46%
No Error	0.00%	4.43%	50.95%	6.17%	61.56%	0.00%	5.44%	29.97%	3.64%	39.05%
Total	33.95%	6.19%	50.95%	8.91%	100.00%	55.47%	9.09%	29.97%	5.47%	100.00%
	40.14%		59.86%			64.56%		35.44%		

Figure 2 - Branch-error probabilities for the SPEC-Int and SPEC-Fp 2000 benchmarks.

the only difference is that category B represents jumps to the beginning of the basic block, and category C represents jumps to the middle (including the end) of the basic block. The same analysis used in categories B and C are applied to categories D and E, but in this case the errors change the control-flow to different basic blocks. Category F represents the control-flow errors that jump to a non-code memory region. As we stated before, errors in this category can be detected by memory access protection mechanisms.

In order to determine the importance of each branch-error category, we propose an error model to measure the probability for an error to occur at a branch-error category. We implemented the error model in our dynamic binary translator (described in Section 5) and evaluated the branch-error probabilities using the SPEC2000 benchmark.

The error model assumes a soft-error that results in 1 bit change in the address offset of the branch instruction or in the flags that determine the conditional branches direction. We consider that each bit in the address offset and in the flags has the same error probability.

Indirect branches generally have their target determined only at runtime. In order to compute the error probabilities in these instructions we have to modify the application code to analyze the bit errors in the branch target address every time the instruction is executed. Since, on the average, the execution frequency of indirect branches represents less than 5% of the total branches execution frequency, we simplify the analysis by not accounting the errors in these branches.

Given that soft-errors are temporal errors, we have to take into account the execution frequency of each instruction. The taken and not taken ratio is also important. When a conditional branch is not taken, a fault in the address offset does not change the normal control-flow, therefore it is not treated as an error. Considering this, we divide the error probabilities into *taken* and *not taken*. We also subdivide the errors into *addresses* and *flags*. Figure 2 shows the branch-error probabilities for the integer and the floating-point part of the SPEC2000 benchmark suite.

Accordingly to the error model, if we look at the SPEC-Int benchmark and assume a 1 bit fault at a branch instruction, the probability for an error in category A to occur is 4.6%. Faults in the flags when the branch is not taken are responsible for 2.74% of these errors.

Branch-error Category	Error Probability	
	SPEC-Int	SPEC-Fp
A	20.70%	17.33%
B	0.41%	0.03%
C	2.22%	16.98%
D	4.04%	1.52%
E	72.62%	64.14%
Total	100.00%	100.00%

Figure 3 - Branch-error probabilities for categories A, B, C, D, and E.

Most of the faults generate errors in category F or do not lead to errors. Since the memory protection systems can detect the errors in category F and we are not interested in faults that do not generate errors, we focus our attention to errors in categories A to E, in other words, the errors that may lead to silent data corruption. Figure 3 shows the branch-error probabilities when considering only these categories.

Notice that most of the errors are in category E, followed by category A. Categories C and D have different behavior in the SPEC-Int and the SPEC-Fp benchmarks. Given that the floating-point applications have big basic blocks, the probability of error in category C is higher than category D in the SPEC-Fp benchmark.

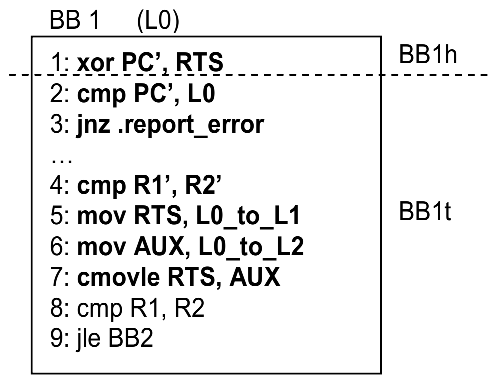
3. Control-flow Checking Techniques

Control-flow checking is usually performed through signature monitoring. The basic idea is to detect errors by comparing a run-time signature with a pre-computed one. Although some works have used hardware to assist the checking, we focus on the software-only techniques.

Alkhalifa *et al*. [1] proposed the Enhanced Control-flow Checking using Assertions (ECCA). This technique modifies the source code inserting a test assertion in the beginning of every basic block to check the signature, and a set assignment in the end to update the signature to the next basic blocks. The technique use expensive instructions (div and mul) to check and update the signature, and cannot detect control-flow errors in category A.

The control-flow checking by software signatures [12] (CFCSS) assigns a signature to each basic block, and uses a shadow PC register (PC') to track these signatures and to detect the errors in the control flow. PC' is implemented using a general purpose register that always contains the signature for the currently executing block. Upon entry to any block, the PC' is xor'ed with a statically determined constant to transform the previous block's signature into the current block's signature. After the transformation, the PC' can be compared to the basic block signature. The CFCSS technique requires that common predecessor blocks have the same signature. Due to this restriction, the technique cannot detect errors in categories D and E if the correct and the wrong target have the same signature. Since the signature is updated only at the beginning of the basic block, errors in category C also cannot be detected. Finally, the CFCSS technique updates the current basic block signature using the predecessor basic block signature; therefore, the successors of a branch basic block cannot distinguish if the last branch was mistaken. Consequently, errors in category A are not detected.

The ECF (enhanced control-flow checking), proposed by Reis *et al*. [13], extends the CFCSS technique with a run-time adjusting signature (RTS). They innovate by introducing a conditional signature update, so that predecessor or successor basic blocks are not required to have the same signature. Figure 4 shows an example of the ECF technique using the "cmovle" instruction to update the RTS accordingly to the next basic block.



```
1: xor PC', RTS
2: cmp PC', L0
3: jnz .report_error
…
4: cmp R1', R2'
5: mov RTS, L0_to_L1
6: mov AUX, L0_to_L2
7: cmovle RTS, AUX
8: cmp R1, R2
9: jle BB2
```

Figure 4 - ECF technique applied to a basic block with signature L0.

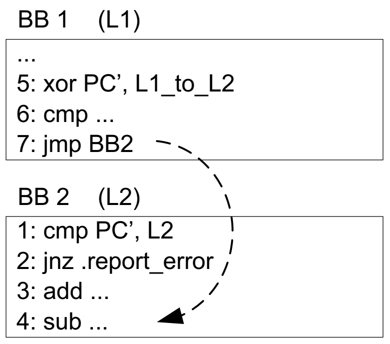
The conditional signature update in ECF fixes limitation of the CFCSS technique of not being able to detect errors in categories A, D and E. To the best of our knowledge, this is the first technique to cover all the branch-errors in categories A, B, D, and E, which represents the best error coverage when considering the experimental results with our error model. Although the technique covers most of the categories, it still cannot detect errors in category C, which represents 16.98% of the potential errors in the SPEC-Fp 2000 benchmark.

In order to detect the errors in all the branch-error categories we propose two new comprehensive control-flow checking techniques: the first is the Edge control-flow checking technique (EdgCF) and the second is the Region based control-flow checking technique (RCF).

3.1. The Edge Control-Flow Checking Technique

The main idea behind the Edge Control-Flow (EdgCF) Checking Technique is to update the PC' register with the next basic block signature in the end of the current basic block and check it in the beginning of the next one. Figure 5 shows an example of PC' being updated and checked. Instruction 5 updates the PC' with the next basic block signature. L1_to_L2 is a constant that when xor'ed with L1 generates L2. Instructions 1 and 2 check PC'.

Notice that the example in Figure 5 still does not detect errors that jump to the middle of the correct target basic block. If, due to a fault, instruction 7 branches directly to instruction 4, the execution skip instructions 1 to 3, then the code will not detect the error.

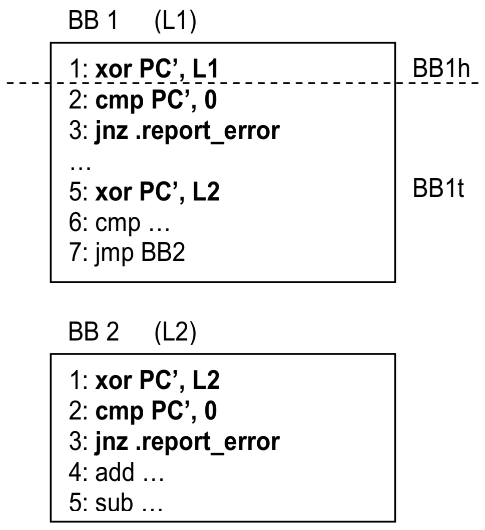


```
BB 1    (L1)
...
5: xor PC', L1_to_L2
6: cmp ...
7: jmp BB2

BB 2    (L2)
1: cmp PC', L2
2: jnz .report_error
3: add ...
4: sub ...
```

Figure 5 - PC' update and branch error example.

The undetected error in Figure 5 occurs because the control-flow jumps between two points that have the same signature (L2). In order to detect this kind of error we also update the signature in the beginning of the basic block. Figure 6 shows the EdgCF technique updating PC' in the beginning of the basic blocks.



```
BB 1    (L1)
1: xor PC', L1                 BB1h
------------------------------------
2: cmp PC', 0
3: jnz .report_error
…
5: xor PC', L2                 BB1t
6: cmp …
7: jmp BB2

BB 2    (L2)
1: xor PC', L2
2: cmp PC', 0
3: jnz .report_error
4: add …
5: sub …
```

Figure 6 - PC' updated in the beginning of the basic blocks.

The EdgCF technique modifies PC' so that between two basic blocks (in the control-flow edges), the PC' contains the correct next basic block signature, and in the middle of the basic blocks it contains zero. The technique is able to detect the fault that makes instruction 7 in BB1 jumps to the middle of BB2 in Figure 6 (similar to the jump in Figure 5). Although this fault skips the checking code in BB2, the PC' value will also be wrong in the next basic block, and the next checking code will detect the error.

To update PC' at the end of the basic block, we need to consider the following cases:

- If the basic block has only one successor: we insert an instruction to transform the current value of PC' (zero) to the new value (the successor basic block signature).
- If the basic block has a conditional branch: we use conditional instructions (such as predicated instructions, or conditional branches) to update the signature accordingly to the next basic blocks.
- If the basic block has a dynamic branch, such as an indirect jump, a call, or a return instruction: we generate code to get the dynamic target address and map it to the target basic block signature. To avoid the cost of mapping the address to the signature, we use the address of the first instruction in a basic block as the basic block signature. This is very convenient, since in this way we always have unique signatures, and the address to signature mapping has no cost.

In the IA-32 architecture, the return ("ret") instruction is an implicit dynamic branch. This instruction pops the target address from the stack and branches to it. Figure 7 shows the EdgCF technique code to update PC' in the end of a basic block when it has a "ret" instruction. Instruction 5 copies the target address to register R1, and instruction 6 changes PC' accordingly to the next basic block using R1.

```
BB 1
...
5 :  mov R1, [RSP]
6 :  xor PC', R1
7 :  ret
```

Figure 7 – PC' updated in the end of a basic block with a dynamic branch ("ret" instruction).

Figure 8 shows an example of a basic block with a conditional branch. Instructions 6 to 10 update the PC' (using the conditional move instruction-"cmov") to the next basic block signature (L1 or L2) accordingly to the branch condition (note that we assume that the xor instruction won't change the flag set by the cmp instruction at 6. We will address this issue late).

The example in Figure 8 uses a branch instruction to report the detected error. This instruction is a new potential source of branch-errors, but the EdgCF and the previous techniques do not handle the potential errors properly. Notice that if, due to a soft-error, instruction 3 jumps to another basic block, the technique still detect the error, but if the control-flow deviation is to the middle of the same block, the techniques will not detect it. since the PC' will remain zero according to our single error model. The ECCA technique [1] uses the "div" instruction to check the signature. The technique generates two numbers and divides both so that if the signature is wrong, a division by zero exception occurs. The divide by zero exception handler is modified to detect if the exception is a control-flow error or a pure divide by zero exception. Although the EdgCF technique can use the same approach to check the signature, the "div" instruction is very expensive, and the performance overhead would be prohibitive. To avoid this performance overhead and detect the potential branch-error introduced by the new branch instructions, we proposed the Region Based Control-Flow Checking technique.

```
BB 1    (L1)
1 :  xor PC', L1
2 :  cmp PC', 0
3 :  jnz .report_error
...
6 :  cmp R1', R2'
7 :  mov AUX, PC'
8 :  xor PC', L2
9 :  xor AUX, L3
10:  cmovle PC', AUX
11:  cmp R1, R2
12:  jle BB3
```

Figure 8 - Basic block with a conditional branch.

3.2. The Region Based Control-Flow Checking Technique

In order to protect the execution from control-flow errors on the new branch instructions introduced in the basic block, we create a different region for each branch instruction, and a region for the original basic block instructions. We will call this approach Region based Control-Flow (RCF) Checking. Figure 9 shows the RCF technique applied to the code in Figure 8. The region R1E represents the entrance of the basic block BB1. The signature checking code is placed in this region. Region R1 is assigned to the original basic block instructions and the regions that appear from instruction 9 to 12 (R2E and R2E/R3E) are result of the signature update code in the conditional basic block. The region R2E/R3E means that both R2E and R3E are valid signatures.

Notice that the branch instruction introduced to check the signature is protected by region R1E. In other words, if a branch-error occurs in this branch instruction, and the instruction jumps to code out of this region, the signature will be wrong, and the technique will detect the error. We could assign a region for each instruction, and so increase the protection from non-branch-errors, but the performance cost and code footprint size would be prohibitive.

We update PC' in the end of basic blocks using the same approach in EdgCF technique.

```
              BB 1
R1E           1 : cmp PC', R1E
              2 : jnz .report_error
              3 : xor PC', R1E_to_R1
-----------------------------------------
              ...
R1            6 : cmp R1', R2'
              7 : mov AUX, PC'
              8 : xor PC', R1_to_R2E
-----------------------------------------
R2E           9 : xor AUX, R2_to_R3E
              10: cmovle PC', AUX
-----------------------------------------
R2E/R3E       11: cmp R1, R2
              12: jle BB2
```

Figure 9 - Regions attributed to a basic block. Region R2E/R3E means that R2E and R3E are valid signatures.

4. Correctness

In this section, we first formalize the control flow checking problem. All the signature-based control flow checking techniques can be formalized in the same way. The only differences between them are different signature generation function GEN_SIG and signature checking function CHECK_SIG. Based on that, we give the sufficient and necessary conditions for the GEN_SIG and CHECK_SIG function in order to detect any single control-flow error without false positive. The previous control flow checking techniques do not satisfy the sufficient condition. Therefore, none of them can detect all possible single control-flow errors. We show that our EdgCF technique can detect any single control-flow error by proving that it satisfies both the sufficient and necessary condition. At last, we also show that the GEN_SIG function in our EdgCF technique is among the simplest functions (hence the lowest overhead) that can satisfy both the sufficient and necessary condition.

4.1. Control Flow Checking Problem

To detect the control-flow errors on branch instructions, we must distinguish the following two different branch targets:

Definition 1: The *logic branch target* is the branch target in the program semantic.

Definition 2: The *physical branch target* is the branch target in the program execution.

Without control-flow error, the logic branch target and the physical branch target are the same. The control-flow errors lead to the mismatch between the logic branch target and the physical branch target. So *the goal of control-flow checking is to detect the mismatch between the logic branch target and the physical branch target.*

To formalize the control-flow checking problem, we divide the program into basic blocks so that control-flow errors happen only at the end of a block. With basic blocks, we can represent each logic branch target as a basic block (e.g. using the beginning address of basic block). However, we can not represent the physical branch targets as basic blocks because a control flow error may jump to the middle of a basic block.

To handle the jump-to-the-middle problem, we further split each basic block B into two basic blocks: the head block Bh and the tail block Bt, as shown in Figure 10. The head block Bh contains no instruction in block B. (Although Bh contains no instruction in original block B, it may be instrumented with instructions for control flow checking, see section 4.2.) It only acts as the entry point of block B and falls through to the tail block (Bt). The tail block Bt contains all the instructions in block B. We should note that, *the control flow error never happens on the fall-through edge Bh → Bt.* So the correct control flow B1 → B2 becomes B1h → B1t → B2h → B2t, and the control flow error that B1 jumps to the middle of B2 can be modeled as B1h → B1t → B2t. In this way, we can ignore the jump-to-the-middle problem and simply represent each physical branch target also as a basic block.

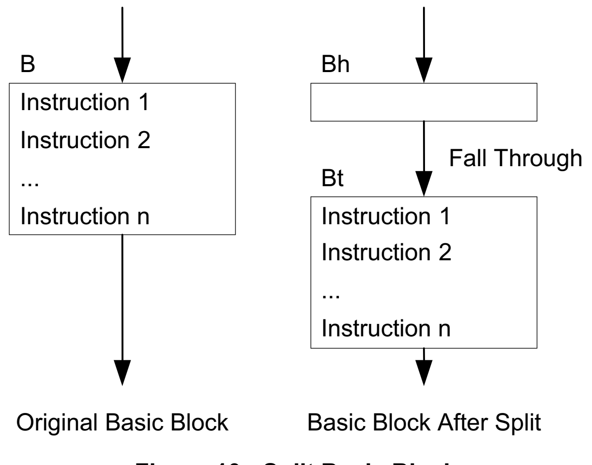


Figure 10 - Split Basic Block.

With the above treatment, we now can formalize any program execution path (including both the correct program execution paths and the program execution paths with control-flow errors) as follows:

Definition 3: The *program execution path* is a sequence of basic blocks B_i, $0 \le i \le n$, where n is the number of blocks executed in the program, such that B_{i+1} is the physical branch target of the last branch instruction of B_i. Also for each basic block B_i, $0 \le i < n$, there is a basic block T_{i+1}, which is the logic branch target of the last branch instruction of B_i. In other words, B_{i+1} is the basic block executed after B_i, and T_{i+1} is the basic block that was supposed to be executed after Bi.

With definition 3, the control-flow checking problem is to answer the following question:

$$(T_{i+1} == B_{i+1})\ ?, \quad 0 \le i < n$$

4.2. Control Flow Checking Technique

To solve the control flow checking problem, the signature-based control flow checking techniques instrument a signature generation function GEN_SIG at the exit of each (head and/or tail) block to generate a signature for the program execution path. They also instrument a signature checking functions CHECK_SIG at the entry of each (head and/or tail) block to check the signature for control flow error detection.

Different control flow checking techniques use different GEN_SIG and CHECK_SIG functions. For example, in the EdgCF technique shown in Figure 6, the signature is computed and placed in PC'. In block BB1, Instruction 1 belongs to the head block (block BB1h) and the rest instructions belong to the tail block (block BB1t). At the exit of block BB1h, instruction 1 generates the signature with:

$$PC' \quad \leftarrow \quad PC' \oplus L1$$

At the exit of block BB1t, instruction 5 generates the signature with:

$$PC' \quad \leftarrow \quad PC' \oplus L2$$

At the entry of block BB1t, instruction 2-3 checks the signature with:

$$(PC' == 0)\ ?$$

As another example, in the ECF techniques shown in Figure 4, the signature is computed and placed in a pair <PC', RTS>. In block BB1, Instruction 1 belongs to the head block (block BB1h) and the rest instructions belong to the tail block (block BB1t). At the exit of block BB1h, instruction 1 generates the signature with:

$$< PC', RTS > \quad \leftarrow \quad < PC' \oplus RTS, RTS >$$

At the exit of block BB1t, instruction 4-7 generates the signature with:

$$< PC', RTS > \quad \leftarrow \quad < PC', L0\_to\_L1 >$$

or

$$< PC', RTS > \quad \leftarrow \quad < PC', L0\_to\_L2 >$$

At the entry of block BB1t, instruction 2-3 checks the signature with:

$$(PC' == L0)\ ?$$

4.3. Assumptions

To prove the correctness of the control-flow checking techniques, we have to make some basic assumptions on the following instrumentation-dependent issues:

Issue 1: The instrumented code itself is also susceptible to control-flow errors.

Issue 2: A soft-error may change the control-flow so that it skips the instrumented code, thus escaping the checking.

With Issues 1, a control flow error may jump to the middle of the instrumented code, whose effect is implementation-dependent. Moreover, the instrumented code needs to be implemented with self-checking if itself has control-flow error. Here we do not consider those implementation details (although our RCF technique can handle the error in the instrumented branch instruction). So we make the following assumption:

Assumption 1: The instrumented function (GEN_SIG or CHECK_SIG) is an atomic unit such that either all or none of it is executed.

Assumption 1 simplifies our later discussion. Based on our experience in IA32, the control flow errors that break the atomicity of GEN_SIG and CHECK_SIG function usually lead to the program fails (e.g. illegal instruction trap) or checking fails. So the non-atomicity of these functions does not affect much to the control flow checking. Moreover, the GEN_SIG function and CHECK_SIG function are typically implemented as simple as possible to reduce the checking overhead. So the probability that a control flow error breaks the atomicity of these functions is small.

With issue 2, if a control-flow error escapes all the instrumented checking functions, there is no way to detect the error. So we have to make the following assumption:

Assumption 2: A control flow error may skip CHECK_SIG functions. However, any control-flow error must finally reach at least one CHECK_SIG function.

4.4. Correctness Condition

We denote the signature generated at the exit of block B_i and checked at entry of block B_{i+1} as S_{i+1}. The signature checking function CHECK_SIG at the entry of block B_{i+1} need to check S_{i+1} for control flow error. Therefore, the signature checking function at the entry of block B_{i+1} should be in the form:

$$CHECK\_SIG(S_{i+1}, B_{i+1})\ ?$$

The signature S_{i+1} generated at the exit of block B_i must depend on T_{i+1} in order to detect the control flow error happens from B_i to T_{i+1}. Moreover, with assumption 2, in order to detect all the control flow errors, one single CHECK_SIG function must be able to detect all the control flow errors along the whole program execution path. So the signature S_{i+1} also need to recursively depend on the previous signature S_i to detect the control flow error happened in previous blocks. Therefore, the signature generation function at the exit of block B_i should be in the form:

$$S_{i+1} \quad \leftarrow \quad GEN\_SIG(S_i, B_i, T_{i+1})$$

If we only consider the *single-error* along the program execution path, we can get the following sufficient condition for detecting any single control-flow error:

Sufficient Condition:

$$\forall j, 0 \le j < n, T_{j+1} \ne B_{j+1}, T_{i+1} = B_{i+1}, i \ne j, 0 \le i < n$$
$$\Rightarrow \quad \neg CHECK\_SIG(S_n, B_n)$$

The sufficient condition only guarantees that there is no false negative in the checking. However, it may produce false positives. To avoid the false positives, we need the following necessary condition:

Necessary Condition:

$$T_{i+1} = B_{i+1}, 0 \le i < n \quad \Rightarrow \quad CHECK\_SIG(S_n, B_n)$$

All the previous control flow techniques [1][12][13] satisfy the necessary condition. Unfortunately, none of them satisfy the sufficient condition. Therefore, none of them can detect all possible single control-flow errors. Next we show that our EdgCF technique can detect any single control-flow error by proving:

Claim 1: The *GEN_SIG* function and *CHECK_SIG* function in EdgCF technique satisfies both the sufficient and necessary condition:

Proof: First, if we represent each head block by the unique block address and each tail block by 0, the GEN_SIG and CHECK_SIG function in EdgCF techniques can be expressed as

$$\begin{aligned} GEN\_SIG(x,y,z) &\equiv x \oplus y \oplus z \\ CHECK\_SIG(x,y) &\equiv (x == y) \end{aligned} \quad (4)$$

To check (4), with block BB1 in Figure 6, we can derive:

$$GEN\_SIG(PC', BB1h, BB1t)$$
$$= GEN\_SIG(PC', L1, 0) = PC' \oplus L1$$

$$GEN\_SIG(PC', BB1t, BB2h)$$
$$= GEN\_SIG(PC', 0, L2) = PC' \oplus L2$$

$$CHECK\_SIG(PC', BB1t)$$
$$= CHECK\_SIG(PC', 0) = (PC' == 0)$$

For simplicity, here we represent all the tail blocks by 0, which is not unique for each tail block. This does not affect the correctness of our control flow checking technique because the control flow error never happens on the fall-through edge from a head block to a tail block (see section 4.1).

Next, with formula (4) and $S_0 = B_0$, we can derive:

$$T_{i+1} = B_{i+1}, 0 \le i < n$$
$$\Rightarrow \quad S_n = S_{n-1} \oplus B_{n-1} \oplus T_n = \ldots = S_0 \oplus B_0 \oplus T_n = B_n$$
$$\Rightarrow \quad CHECK\_SIG(S_n, B_n)$$

And

$$\forall j, 0 \le j < n, T_{j+1} \ne B_{j+1}, T_{i+1} = B_{i+1}, i \ne j, 0 \le i < n$$
$$\Rightarrow \quad S_n = S_{n-1} \oplus B_{n-1} \oplus T_n = \ldots = B_{j+1} \oplus T_{j+1} \oplus T_n \ne B_n$$
$$\Rightarrow \quad \neg CHECK\_SIG(S_n, B_n)$$

□

For previous discussion, we know that the signature generation function GEN_SIG must change with S_i, B_i and T_{i+1}. So the GEN_SIG function in (4) is among the simplest functions (hence the lowest overhead) that can satisfy both the sufficient and necessary condition. Another similar choice is GEN_SIG (x, y, z) = x - y + z, which also satisfies both the sufficient and necessary condition. In real implementation, we actually use this function to avoid the EFLAGS problem in IA32 (see Section 5.1).

5. Implementation

We implemented the control-flow checking techniques in our dynamic binary translator (DBT).

The overall structure of the DBT is shown in Figure 11. The DBT runs on top of OS as a user-level run-time system. The program binary code is dynamically translated and stored into the code cache. Then the translated code can be executed under the control of the DBT, which allows us to apply different dynamic binary translation techniques to the code, such as compatibility support, security checking, reliability enforcement, performance improvement, etc.

The DBT consists of three individual modules: the *Runtime* module, the *Frontend* module and the *Backend* module.

The *Runtime* module provides the system supports for the DBT. It automatically loads the original program code into memory and initializes the program execution context at program startup. To facilitate the program execution, it provides all OS interfaces such as I/O and system calls. The *Runtime* module also handles all the system events such as OS call-backs, exception, dynamic library load, and code self-modification, etc.

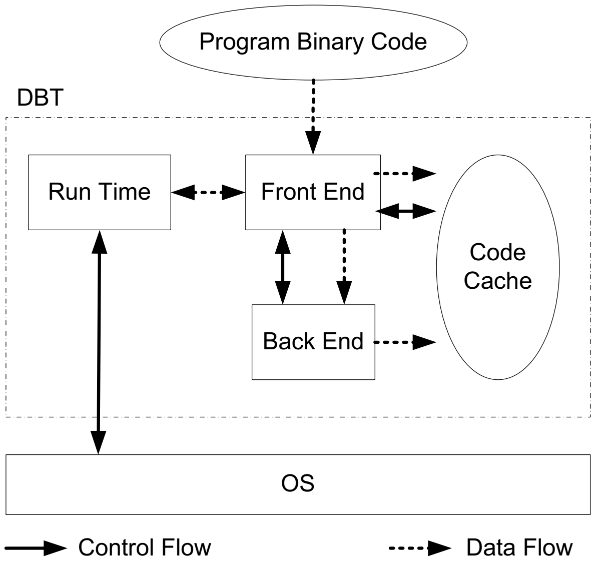


Figure 11 - The dynamic binary translator overall structure.

The *Frontend* module manages the whole program execution for dynamic binary translation. It dynamically recognizes the original program instructions, translates them into instructions in the code cache using different dynamic binary translation techniques, and controls the code execution in the code cache. For the system related features in the program, it interacts with the *Runtime* module to get system support. To provide optimization to the dynamic binary translation, the *Frontend* module also collects program profiling information during the code execution and selects hot traces based on the profiling information for run-time optimization by the *Backend*.

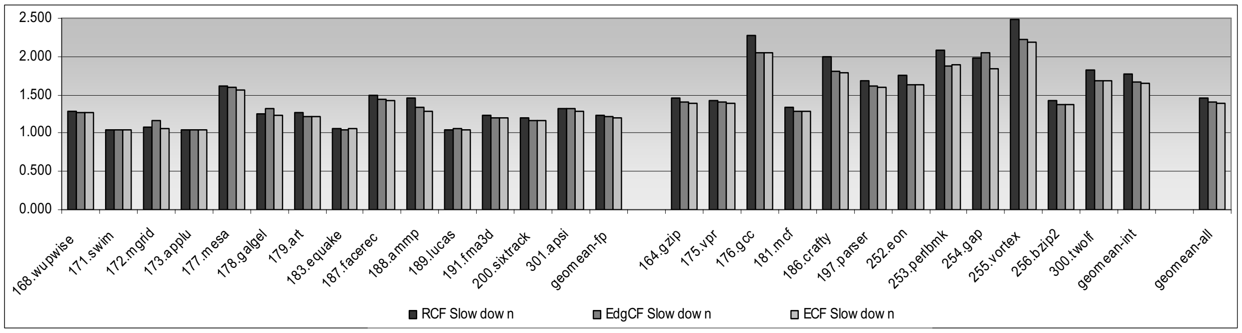


Figure 12 - Performance slowdown for the RCF, EdgCF and ECF techniques.

The *Backend* module performs run-time optimization for the dynamic binary translations. It builds an intermediate representation (IR) from the hot traces selected by the *Frontend* module. It then performs optimizations on the IR, and finally generates optimized code into the code cache to improve performance.

In our experiments, we configured the DBT to translate IA32 [7] to EM64T [6] binary code.

The binary translation is done on demand, which means that every time a non-translated basic block has to be executed, the DBT takes control of the execution and translate the basic block. Therefore, only executed blocks are translated.

The code cache and the DBT code are placed in memory pages with the *execute disable bit* set to allow execution. This allows us to detect branch-errors in category F.

Self-modifying code is handled using the *write protection* mechanism. Whenever the program modifies the original code, the processor raises an exception and the DBT takes control of the execution. After this, the DBT identifies and removes the outdated code that was previously translated from the modified code. As soon as the control flows to the modified code, DBT naturally translates it into the code cache as if it was never translated before.

In order to implement the control-flow checking techniques, we insert instructions to check and update the signature in every translated basic block. Due to the translation on demand scheme, the CFG is changed during the execution; therefore, we do not implement the techniques that need the CFG to attribute signatures to the basic blocks, such as CFCSS and ECCA. The ECF, EdgCF and RCF techniques were implemented using the address of the first instruction in each basic block (region) as the signature number.

5.1. EM64T Architecture Issues

The control-flow checking techniques require dedicated registers for PC', and for the runtime signature register (RTS) in the ECF technique. Since the EM64T architecture has more registers than the IA32 one, we do not need to spill registers to provide PC' and RTS during the translation of IA32 programs to EM64 code.

When modifying the binary, we have to make sure that the inserted instructions do not change the program behavior. Although the instructions to update the signatures only modify registers not used by the original program, the "xor" instruction implementation in the EM64T architecture modifies the EFLAGS register. Therefore, instead of using the "xor" instruction, we use *load effective address* (lea). The "lea" instruction does not have side-effects and has performance similar to the "xor" instruction. Moreover, it is a three address instruction, which allows us to save some instructions in the implementation. Figure 13 shows the usage of the "lea" instruction to update the signature.

We implemented the signature checking using both the "cmov" and the branch instructions. The approach insert branches in the ECF and EdgCF techniques is unsafe, however, it provides a fair performance comparison with the RCF technique.

The signature checking code must not generate side-effects as well. In order to check the signature without changing the EFLAGS, we use the *jump if CX is zero* (jcxz) instruction. Figure 13 shows an example of the RCF technique checking the signature. Instruction 1 updates PC' to R1C region. Instructions 2 and 6 save and restore CX, and instructions 3 to 5 check PC'.

```
      BB 1
      1: mov AUX, ecx
      2: lea ecx, [PC'-R1E]
      3: jcxz BB1.check
R1E   4: jmp fault_detected
      BB1.check:
      5: mov ecx, AUX
```

Figure 13 - RCF technique checking the signature.

6. Experimental Results

The results were generated using an Intel Xeon machine with 3.6 GHz and 4GB of RAM. We executed the SPEC 2000 benchmark with the reference data set. The baseline results are the applications running on the DBT with no instrumentation. The average slow down from the native code to running on DBT is about 12%.

Figure 12 shows the performance slowdown for the SPEC 2000 applications when applying the RCF, EdgCF and ECF techniques in the DBT. The left part of Figure 12 shows the SPEC floating point benchmarks and the right part shows the integer ones. The figure also shows the geometric mean for the floating point, the integer, and the entire benchmark. The techniques RCF, EdgCF, and ECF presented an average slowdown of 1.46, 1.41, and 1.39 times, respectively.

Notice that the performance slowdown is less dramatic in the floating point benchmarks. It happens because these benchmarks have large basic blocks and/or more time-consuming instructions (like floating point instructions).

Since the RCF technique requires the signature to be updated more than twice in each basic block, it inserts more instructions per basic block than the other techniques. Therefore, this technique shows worse performance than the other two. The EdgCF and the ECF techniques insert the same amount of instructions per basic block. However, in the current ISA, the ECF uses cheaper instructions to update the signature, which leads to a slight performance difference.

Although the RCF technique presented the worst performance, it has the best error coverage, since it can detect the errors in the branch instructions inserted to update and check the signatures. The EdgCF and ECF techniques can use safe instructions (like "cmov" and "div") to update and check the signatures, but these instructions may lead to performance loss. We also implemented the control-flow techniques using the "CMOVcc" (conditional move) instruction to update the signature. Figure 14 shows the average performance slowdown for the SPEC 2000 benchmark when using the "Jcc" (conditional jump) and the "CMOVcc" (conditional move) instruction to update the signature. The shadowed area shows the unsafe configurations.

Notice that the RCF technique using "Jcc" instructions, which is safe, almost beats the ECF technique when using "CMOVcc" instructions.

Update instruction	Performance Slowdown		
	RCF	EdgCF	ECF
Jcc	1.46	1.41	1.39
CMOVcc	1.57	1.54	1.44

Figure 14 - Performance slowdown when using the "Jcc" and "CMOVcc". The shadowed cells indicate the techniques that are unsafe when implemented using the "Jcc" instruction to update the signature.

The fail-stop model relies on the *halt-on-failure* property [4]. Accordingly to this property, the error should be detected and the process stopped before writing permanent data or communicating with other processes. The existing software-based control-flow checking techniques, however, are not able to implement the halt-on-failure property, even if it checks the signature after each instruction. It happens because a branch-error can change the control-flow directly to an instruction that store data in the memory, and even if the error is detected immediately after the instruction, the stored data may be a communication with other processes or a permanent data write. For that reason, we assume a relaxed fail report model, where the error must be reported, but not necessarily before any data write or communication.

As long as we are not required to report the error immediately, the control-flow checking techniques can be optimized for performance by not checking the signature at every basic block. Notice that the signature may not be checked, but it still has to be updated in every basic block. This is valid because if an error occurs, and the signature becomes wrong, each update to PC' will also generate a wrong signature, therefore, if we consider only single errors, once the signature becomes wrong, it will always be wrong, and the signature can only be checked for correctness in the end of the program (or functions). As an example of optimization, Reis *et al.* [13] proposed checking the signature only in basic blocks that have store instructions.

In order to evaluate the impact that signature checking has on performance, we implemented four signature checking policies in the control-flow checking techniques:

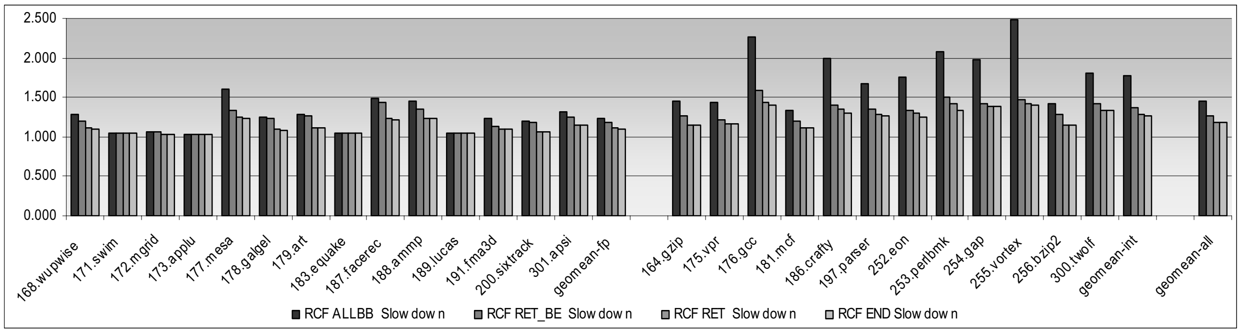


Figure 15 - SPEC2000 performance for the RCF technique implementing the signature checking policies.

- ALLBB: the signature is checked in every basic block.
- RET-BE: the signature is checked in basic blocks with back edges, and in basic blocks with return instructions.
- RET: the signature is checked in basic blocks with return instructions.
- END: the signature is checked only in the end of the application.

The signature checking policies presentation is sorted by the signature checking frequency. Notice that the less frequently we check the signature, the more delay it can take to report the error. Moreover, a branch-error may lead the program to an infinite loop, and in the case of the RET and END policies, where the signature is not checked inside loops, the error may not be reported. The RET-BE method places checks in the blocks that have back edges to help preventing infinite loops. Figure 15 compares the RCF technique performance when implementing the four signature checking policies.

The performance improvement is higher in the integer benchmark than in the floating-point one. On the average, the performance slowdown dropped from 77% to 37% when comparing the ALLBB to the RET-BE policy in the integer benchmark, from 23% to 18% in the floating-point benchmark, and from 46% to 26% in the entire benchmark. Again, the difference in the performance improvement is because the floating-point benchmarks have large basic blocks and/or more time-consuming instructions (like floating point instructions).

The average performance slowdown is 46% for the ALLBB policy, and 16% for the END one. Since the END policy only checks the signature once, and the ALLBB policy check it in every basic block, we can see that the signature checking is responsible for a big share in the performance slowdown. Even though it is an impressive performance improvement, the END policy may not report branch-errors that lead the program to infinite loops.

Although the RET policy checks the signature more frequently than the END one, both policies have similar performance. It happens because the programs spent most of the executing time in inner loops rather than calling and returning from functions.

In our tests, the DBT itself was not modified to provide reliability. The binary translation time is very small compared to the application execution time. Therefore, we estimate that a reliable version of the DBT translator will not change significantly the total execution time.

7. Conclusion and Future Work

In this paper we study a special class of errors called control-flow errors. We proposed a new control-flow error classification, a model to measure the branch-error probabilities, and two new control-flow checking techniques: the Edge Control-Flow and the Region Based Control-Flow checking techniques. We also formalized the control-flow checking problem and provided a proof that our control flow checking technique can detect any single control-flow error.

We implemented the techniques in our dynamic binary translator and evaluate them using the SPEC2000 benchmark. The results show that the RCF technique can cover all the branch-errors, including those that occur at the conditional branch instructions inserted to update/check the signature, and the performance cost is very close or even better than the other techniques.

In the future we will add data flow checking into our implementation and measure the overall performance impact. We will also work on soft-error injection to measure the actual effectiveness of our techniques in detecting both control and data flow errors.

8. Acknowledgments

We would like to thank Shubu Mukherjee, George Reis, Sri Nair, Ho-seop Kim, Mauricio Breternitz, Jr, Brian Deitrich, and the anonymous reviewers for their valuable comments and discussions. We also appreciate the support provided by Jesse Fang at the Programming System Laboratory at Intel.

9. References

[1] Z. Alkhalifa, V. S. S. Nair, N. Krishnamurthy, and J. A. Abraham, Design and evaluation of system-level checks for on-line control-flow error detection, *IEEE Trans. Parallel Distrib. Syst.*, vol. 10, pp. 627-641, June 1999.

[2] H. Ando et al., A 1.3GHz Fifth Generation SPARC64 Microprocessors, *Proc. IEEE International Solid-State Circuits Conference*. (ISSCC 03), IEEE Press, 2003, pp. 246-247.

[3] R. Baumann, Soft Errors in Commercial Semiconductor Technology: Overview and Scaling Trends, *IEEE 2002 Reliability Physics Symp*. Tutorial Notes, Reliability Fundamentals, IEEE Press, 2002, pp.121-01.1-121-01.14.

[4] S. Chandra, P. M. Chen, How Fail-Stop are Faulty Programs?, *Proceedings of the 1998 Symposium on Fault-Tolerant Computing* (FTCS) , June 1998.

[5] C. Constantinescu. Trends and challenges in VLSI circuit reliability. *IEEE Micro*, vol. 23, pp. 14-19, Jul.-Aug. 2003.

[6] Intel@ Extended Memory 64 Technology Software Developer's Guide

[7] IA-32 Intel@ Architecture Software Developer's Manual

[8] S. S. Mukherjee, J. Emer, and S. K. Reinhardt. The Soft Error Problem: An Architectural Perspective, proceeding of the 11th Int'l Symposium on High-Performance Computer Architecture (HPCA-11), pages 243-247, 12-16 Feb. 2005.

[9] T. Michel, R. Leveugle and G. Saucier. A New Approach to Control-flow Checking without Program Modification. *Proc. FTCS-21*, 1991, pp. 334-341.

[10] M. Namjoo. CERBERUS-16: An Architecture for a General Purpose Watchdog Processor. *Proc. Symposium on Fault-Tolerant Computing*. 1983, pp.216-219.

[11] T. J. O'Gorman, J. M. Ross, A. H. Taber, J. F. Ziegler, H. P. Muhlfeld, I. C. J. Montrose, H. W. Curtis, and J. L. Walsh. Field testing for cosmic ray soft errors in semiconductor memories. In *IBM Journal of Research and Development*, pages 41-49, January 1996.

[12] N. Oh, P. P. Shirvani, and E. J. McCluskey. Control-flow checking by software signatures. IEEE Transactions on Reliability, vo. 51, No 2, pp. 111-122, March 2002.

[13] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. SWIFT: Software Implemented Fault Tolerance. *Proceedings of the Third International Symposium on Code Generation and Optimization* (CGO), March 2005.

[14] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 389-399, June 2002.

[15] N. R. Saxena, E. J. McCluskey. Control-flow Checking Using Watchdog Assists and Extended-Precision Checksums. *IEEE Transactions on Computers*, Vol. 39, No. 4, Apr. 1990, pp. 554-559.