Control Flow Checking or Not? (for Soft Errors)

ABHISHEK RHISHEEKESAN, Intel Technology India Pvt. Ltd., India
REILEY JEYAPAUL, ARM Research, UK
AVIRAL SHRIVASTAVA, Arizona State University, USA

Huge leaps in performance and power improvements of computing systems are driven by rapid technology scaling, but technology scaling has also rendered computing systems susceptible to soft errors. Among the soft error protection techniques, Control Flow Checking (CFC) based techniques have gained a reputation of being lightweight yet effective. The main idea behind CFCs is to check if the program is executing the instructions in the right order. In order to validate the protection claims of existing CFCs, we develop a systematic and quantitative method to evaluate the protection achieved by CFCs using the metric of vulnerability. Our quantitative analysis indicates that existing CFC techniques are not only ineffective in providing protection from soft faults, but incur additional performance and power overheads. Our results show that software-only CFC protection schemes increase system vulnerability by 18%–21% with 17%–38% performance overhead and hybrid CFC protection increases vulnerability by 5%. Although the vulnerability remains almost the same for hardware-only CFC protection, they incur overheads of design cost, area, and power due to the hardware modifications required for their implementations.

CCS Concepts: • **Computer systems organization** → *Reliability*; • **Software and its engineering** → *Data flow architectures*; *Control structures*;

Additional Key Words and Phrases: Soft error, reliability, vulnerability, transient fault, error correction code

ACM Reference format:
Abhishek Rhisheekesan, Reiley Jeyapaul, and Aviral Shrivastava. 2019. Control Flow Checking or Not? (for Soft Errors). *ACM Trans. Embed. Comput. Syst.* 18, 1, Article 11 (February 2019), 25 pages.
https://doi.org/10.1145/3301311

1 INTRODUCTION

The race for technology scaling has exposed the problem of fragile devices and the susceptibility of modern computing systems to environmental peturbances. Soft errors are transient faults caused due to multiple sources like static noise, external interference, and so forth, but the predominant cause of soft errors in modern processors is charge carrying radiation particles. In the era of many core systems the failure rate of our computing systems due to soft errors has exponentially increased, to reach critical levels [21].

Due to the threats posed by soft errors to reliable computing, reliability is rapidly emerging as a key design metric. Many soft error protection schemes have been built around space and/or

Authors' addresses: A. Rhisheekesan, Intel Technology India Pvt. Ltd., Sarjapur 2, 23-56P, Devarabeesanahalli, Varthur Hobli, Outer Ring Road, Bangalore, India; email: abhishek.rhisheekesan@intel.com; R. Jeyapaul, ARM Research, ARM Ltd Central Building (ARM3), 110 Fulbourn Road, Cambridgeshire, Cambridge, UK; email: reiley.jeyapaul@arm.com; A. Shrivastava, Arizona State University, 660 S. Mill Ave, Suite 203-15, Tempe, AZ; email: aviral.shrivastava@asu.edu.

1539-9087/2019/02-ART11 $15.00
https://doi.org/10.1145/3301311

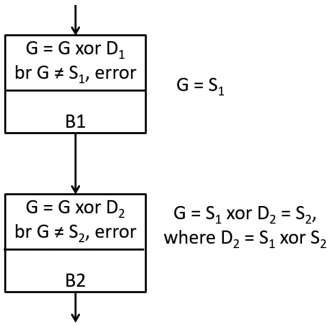


Fig. 1. CFCSS [28]: A software CFC scheme. CFCSS attaches signatures to each basic block. In basic block B2, the first instruction expects *Runtime Signature Register* G to hold the signature S_1 of basic block B1. It applies *xor* operation on contents of G with D_2 and stores the result in G, where $D_2 = S_1$ *xor* S_2. If the equality check for $G = S_2$ fails, e.g., if the control flow erroneously reached basic block B2 from some basic block other than B1 due to a soft fault, it indicates error and detects the CFE.

time redundancy. The simple redundancy based protection schemes, where same computation is executed twice and results checked for a mismatch, have been developed at various levels of system design abstraction—from transistor level [17] to gate level [14, 38] to system level [20, 29, 37]. While these redundancy based techniques can provide effective system reliability, they incur high overhead of 2× or more. Even though the performance overhead of redundancy based methods can be hidden or minimized, the power and area overheads cannot be hidden.

In pursuit of low overhead schemes, researchers found that the majority of soft faults affecting program behavior eventually manifest in the form of faults in the program execution sequence (33% of all transient errors result in control flow error on RISC processors and 77% on CISC processors [30]) and in general, results in erroneous execution. Therefore, by ensuring the control flow of the program is correct, significant protection can be achieved. Several protection schemes were proposed that protect computation by verifying the control flow of the program is correct and are called Control Flow Checking (CFC) techniques. For example, as shown in Figure 1, CFCSS adds some instructions to assign a variable to a unique value (signature) in each basic block, and also adds instructions in each basic block to check if the control flow is coming from a correct predecessor basic block. Several control flow based soft error protection techniques were developed over the last few decades and span across design layers from hardware [11, 23–25, 31, 32], software [2, 3, 9, 15, 28, 39, 40], and hardware-software hybrid techniques [4, 10, 12, 34–36, 42].

The CFC techniques claim to provide significant protection by detecting deviations from the correct execution flow of the program due to soft faults. Almost all the existing CFC techniques have performed targeted fault injection to evaluate the effectiveness of the techniques in detecting these deviations. In the later sections, we explain why these fault injection techniques are ineffective in comprehensively evaluating the effectiveness of the CFC techniques. Our systematic and quantitative methodology to evaluate the protection achieved by CFCs demonstrates that software-only CFC protection schemes increase system vulnerability by 18%–21% and hybrid CFC protection increases vulnerability by 5%. The vulnerability remains almost the same for hardware-only CFC protection, but the implementation of these techniques requires hardware modifications that incur additional overheads of area, power, and design cost. Thus, our studies refute the claims by various CFC schemes that they provide significant protection for execution of programs in processors against soft faults. In fact, our results prove that the latest CFC techniques tend to increase the system vulnerability of the programs compared to their predecessors and make the programs more susceptible to soft errors.

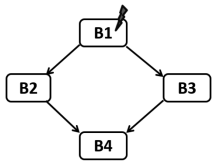


Fig. 2. Control Flow Error (CFE). The execution of instructions from basic block B1 to B2 or from B1 to B3 is correct, but a jump from B1 to B4 due to the occurrence of a soft fault is called a control flow error.

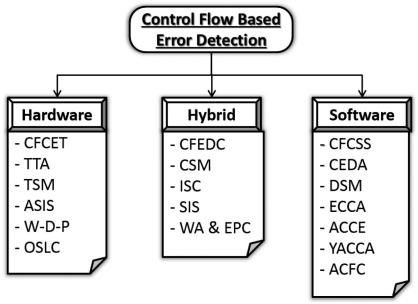


Fig. 3. Classification of CFCs into software based, hardware based, and hardware-software hybrid approaches.

2 CONTROL FLOW ERRORS AND CONTROL FLOW CHECKING

Control flow error (CFE) is the variant of soft errors which cause error in the control flow of applications. For any program, given a program input, the sequence of instructions executed is fixed. A control flow error is defined as the deviation from the correct execution flow of the program. For example, Figure 2 shows the control flow graph for a program with nodes representing basic blocks and edges depicting branches. A basic block is the section of the program code with a single entry point and a single exit point like a branch or a return instruction. In Figure 2, the execution is correct from basic block B1 to B2 or from B1 to B3, but a jump from B1 to B4 is considered incorrect. Due to the occurrence of a soft error strike during the execution in B1, if it jumps to some address location in B4, some of the instructions may have been skipped. Such a deviation in the control flow is termed as a control flow error. A CFE can result in incorrect output of the program like system crash or system hang or may go silent and cause output data errors. A CFC technique identifies deviation from the correct execution flow of the program. For example, it can detect the incorrect execution flow in the above case where the program took a jump from B1 to B4.

Existing CFC techniques can be broadly classified into software based, hardware based, and hardware-software hybrid approaches, as shown in Figure 3.

(1) **Software CFC techniques** [2, 3, 8, 9, 15, 28, 39–41] generally detect deviation in control flow by modification of the program by compilers or binary translators to insert signatures for each basic block in the program to represent the program's control flow, and by comparing these signatures with the signatures generated at runtime. For example, CFCSS [28] inserts a few instructions at the beginning of each basic block to set the outgoing signature (identifying the basic block in the program), and check the incoming signature register in the successor basic blocks to ensure the correctness of execution flow. CEDA [39], an advanced member of the software CFC category, adds signature verification at the start

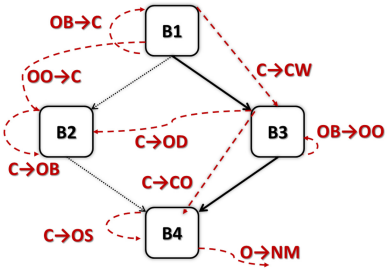
and end of each basis block, and detects even the aliasing errors by maintaining unique signatures for the aliased blocks. (The aliasing problem and its solution in CFCSS+NA [8] will be discussed in a subsequent section.) Other software CFC techniques like ECCA [2], ACCE [40], YACCA [15], and ACFC [41] use some variations of signature verification of basic blocks.

(2) **Hardware CFC techniques** [11, 23–25, 31, 32], in general, deploy dedicated monitoring hardware like watchdog processor or extra hardware within the processor to monitor the control flow by comparing the instruction addresses with stored expected addresses, or runtime signatures with reference signatures, or by verifying the integrity of signatures with error correction codes. For example, CFCET [32] uses execution tracing to transmit the runtime branch instruction address and branch target address to an external watchdog processor. The watchdog processor compares these addresses with the reference addresses stored in its associative memory to detect deviation in control flow. WDP [24] and OSLC [23] use the watchdog processor to compare with reference control flow. ASIS [11] uses a hardware signature generator and watchdog monitor to check the control flow of several processors.

(3) **Hybrid CFC techniques** [4, 10, 12, 34–36, 42], in general, involve modifications of program code using a compiler, and modifications in the processor hardware, to monitor the control flow. They apply a mix of software and hardware techniques to detect CFEs. CFEDC [12] inserts instructions identifying branches before the branches using a compiler, and modifies the fetch and decode stages in the processor pipeline with a hardware logic to correct any errors in the instruction preceded by a hamming code of the branch instruction. SIS [35] generates instruction streams of the program code applied with signatures and monitors signatures of the sequence of executed instructions using a watchdog processor. CSM [42] uses two-dimensional signatures where vertical signature identifies an interval of instructions that comprises multiple blocks and horizontal signature appends additional bits to every instruction word in the horizontal direction, and the dedicated signature monitoring hardware compares them with the runtime signatures.

We employ a widely used metric, *architectural vulnerability* [27], to estimate the unreliability of program execution on a microprocessor. A bit in a processor cycle, represented by a <bit, cycle> pair, is considered vulnerable if a fault in the bit can cause incorrect output and is considered not vulnerable, if the execution eventually results in correct output despite the fault in the bit. For example, in Figure 1, if a fault happens in PC during the fetch of the first instruction of B1, it will be caught by the CFC code of CFCSS. Therefore, the bits in the PC at that cycle are not vulnerable after CFCSS is implemented. The total vulnerability of the program execution is estimated as the count of <bit, cycle> pairs that are vulnerable during the execution in the processor.

Targeted fault injection is another methodology to evaluate the effectiveness of CFC techniques and they have been employed in evaluation of most of the CFC schemes. However, this is computationally prohibitive. Consider that on average, a MiBench benchmark [16] executes for 39 billion cycles on gem5 simulator [5], and it takes 1,121 seconds of execution time on our host processor (a 22-node Linux cluster with 22 Dual Quad-Core Intel Xeon E5620 2.4GHz processors with 24GB RAM). For exhaustive validation of the vulnerability of a 32-bit register, the total number of fault injection simulations required is equal to the number of bits in the register times the average execution cycles, which is equal to 1.25×10^9. So, the total host simulation time required for the fault injection campaign on the 32-bit register is 252 years. In contrast, performance simulation based *architectural vulnerability* calculation requires a single simulation run, which takes 1,121 seconds to execute on our host processor and is extremely faster compared to targeted fault injection.



Acronym	Portion of program basic blocks identified by acronym
O	Basic block in original source code before inserting CFC code
OS	Instructions in same original basic block
OB	Branch instruction in original basic block
OO	Instructions other than branch in original basic block
OD	Instructions in different original basic block
C	Instructions in CFC code of a basic block
CO	CFC instructions other than first instruction in the CFC code
CW	CFC code in wrong successor of original basic block
CA	CFC code in aliased successor of original basic block
NM	Non-code memory region

Fig. 4. Classification of Control Flow Errors (CFEs) that may occur in a generic control flow graph due to soft errors, mapped as transition of execution flow from different locations in basic blocks to locations in the same or different basic blocks or non-code memory regions (the rest of the possibilities are shown in Figure 8). For example, $C \rightarrow CW$ indicates a jump from an instruction in the CFC code of a basic block (C), to the first instruction in the CFC code of the wrong successor of the basic block (CW), which is one of the correct successors but execution should have ideally gone to the other successor of the basic block as per correct execution flow. Solid lines represent the original control flow of the program. Dotted lines indicate a deviation in control flow or CFE, between different portions of basic blocks in the original program code and the CFC code added by CFC techniques, due to soft errors.

This provides avenues for more sophisticated analysis of CFC schemes, as detailed in later sections. Also, architectural vulnerability analysis provides a more comprehensive determination of whether each and every <bit, cycle> pair in execution of a program is vulnerable or not after applying a CFC technique, whereas targeted fault injection selectively inserts faults in random <bit, cycle> pairs during execution of a program and tries to identify whether the fault results in a CFE.

3 QUANTITATIVE ANALYSIS OF CONTROL FLOW CHECKING

Our quantitative analysis of CFC strives to identify the <bit, cycle> pairs that were vulnerable, but are no longer vulnerable after applying the CFC scheme, thereby estimating the protection provided by the CFC technique. We approach the quantitative evaluation of CFC by breaking it down into two steps: (i) Which CFEs will be caused by a fault in a <bit, cycle>? (ii) Given the occurrence of that CFE, can the CFC scheme detect it?

3.1 Which CFEs are Caused by a Fault in a $< bit, cycle >$?

As shown in Figure 4, soft errors can cause a deviation in control flow from different basic blocks to different basic blocks or non-code memory regions. A CFE is specified by a $pc \rightarrow npc$ transition, where pc and npc are two consecutive PCs that are executed, such that in the correct execution npc should not have been executed after pc. For example, $O \rightarrow OS$ transition shows a CFE caused due to a jump from an instruction in the original source code of a basic block (O), to the original source code of the same basic block (OS). $O \rightarrow OD$ transition indicates a jump from an instruction in the original source code of a basic block (O) to the original source code of any other (different) basic block (OD). CW refers to the checking code of a wrong successor of the basic block (first instruction in the CFC code of one of the correct targets of a basic block, but the branch will be taken to the other target in correct execution flow) and CA refers to the first instruction in the CFC code of an aliased target of the basic block. (The aliasing problem and its solution in CFCSS+NA [8] will be discussed in a subsequent section.) CO stands for the rest of the CFC code other than the first instruction in the CFC code. NM indicates the non-code memory region. $OB \rightarrow OO$ indicates

PC		NPC	Impact	CFE type
O	OB	OS	Jump to same original basic block	Not successor CFE
	OO			
O	OB	OD	Jump to different original basic block	Not successor CFE
	OO			
O	OB	CW	Jump to wrong successor CFC code from original basic block	Wrong successor CFE (OB→CW due to branch condition error), Not successor CFE (OO→CW and other OB→CW cases)
	OO			
O	OB	CA	Jump to aliased successor CFC code from original basic block	Not successor CFE
	OO			
O	OB	CO	Jump to other CFC code from original basic block	Not successor CFE
	OO			
C	CW	OS	Jump to same original basic block from CFC code	Not successor CFE
	CA			
	CO			
C	CW	OD	Jump to different original basic block from CFC code	Not successor CFE
	CA			
	CO			
C	CW	CW	Jump to wrong successor CFC code from CFC code	Not successor CFE
	CA			
	CO			
C	CW	CA	Jump to aliased successor CFC code from CFC code	Not successor CFE
	CA			
	CO			
C	CW	CO	Jump to other CFC code from CFC code	Not successor CFE
	CA			
	CO			

Fig. 5. Classification of CFEs based on soft error source location PC and the corresponding destination NPC or next PC, its impact on the program control flow, and whether they fall into *(i) Not-successor CFE* or *(ii) Wrong-successor CFE* categories. *Wrong successor CFE* occurs if the execution flows from *OB* or a branch instruction in the original source code of a basic block to the first instruction in the wrong target basic block's CFC code (*CW*). The rest of the cases fall under *Not successor CFE* category.

a jump from a branch instruction in the original source code of a basic block (*OB*) to any other instruction in the original source code of a basic block (*OO*).

A vulnerable <bit, cycle> can result in two types of CFEs: *(i) Not-successor control flow errors*, i.e., when the execution does not flow to a correct descendant of the instruction. For example, if the execution jumps from the last instruction of a basic block to a basic block that is not a correct successor of the basic block, or to non-first instruction of a correct successor, or to an instruction in the same basic block, and *(ii) Wrong-successor control flow errors*, i.e., when the execution flows to a correct successor of the instruction, but it is incorrect, i.e., in a fault-free execution, the execution would not flow to that successor instruction. For example, a soft error alters the value of a register, which leads to the conditional branch decision to be taken in the alternate direction, but a correct branch target. As you can see from Figure 5, $pc \rightarrow npc$ transition from *OB* to *CW* due to a soft error in the branch condition leading to the branch being taken in the alternate direction is a case of wrong successor CFE and the rest of the cases fall under the not-successor CFE category. Identifying the CFEs caused by a bit flip is considerably hard if we have to follow the dependency chain of the faulted bit to see if it results in altering the control flow of the program. Fortunately, our analysis is simplified by an important observation. None of the CFC techniques can detect wrong-successor CFEs (except for YACCA [15] and CEDA [39], which try to detect some of them, but with little success as discussed later in the section) and hence we do not need to model the faults that cause wrong-successor CFEs. To model the not-successor type CFEs, we perform a microarchitectural component-wise analysis to identify the CFEs generated by a vulnerable <bit, cycle>. To simplify the discussions and due to lack of space, we will assume a five stage in order DLX pipeline [18], but our approach is not restricted to such architectures. Chapter 3 in [18] explains the five pipeline stages: Instruction Fetch (IF), Instruction Issue/Decode (ID), Execute (EX),

Write Result to CDB, Commit result to registers/memory (MEM to represent both Write Result and Commit), and other processor components like Reorder Buffers (ROBs), Instruction Queue (IQ), Register File (RF), Load/Store Queues/Buffers (LSQs), Branch Predictor, Common Data Bus (CDB), Reservation Stations (RSs) as part of the DLX pipeline, and caches like instruction cache and data cache as later additions to it.

3.1.1 Faults in PC. A flip in a PC bit can cause an erroneous value to be written into the PC if the instruction is a non-branch instruction. If it is a *branch* instruction, the erroneous PC value is overwritten in the next cycle with the value from the *branch target address* field in the *execute-memory* (EX/MEM) pipeline register. Since the erroneous PC value is not used to fetch instructions, the PC bits in the current cycle of a branch instruction can be considered not vulnerable. Let us assume a *non-branch* instruction in the current cycle. The bit flip in the PC causes the PC to change to a value that is at 1-bit hamming distance from the current value. Therefore, the PC in the next cycle will become $npc = H1(pc) + 4$, where the function $H1(value\ v)$ calculates all the possible values that are 1-bit hamming distance from v. We calculate *npc* values for every such 1-bit hamming distance value of *pc*, and then use those $pc \rightarrow npc$ transitions to find if the CFC technique will be able to catch the erroneous transition or not, as discussed in the next section.

3.1.2 Faults in Pipeline Registers. In order to estimate the vulnerability of the pipeline registers, the effect of faults in the fields that can cause *not-successor* CFEs need to be modeled. We start with the branch target address field in the EX/MEM pipeline register. If the instruction in the current cycle is a branch instruction, a bit flip in the branch target address field can cause an erroneous value to be written in the next cycle. For this incorrect $pc \rightarrow npc$ transition, the corresponding *npc* value is $npc = H1(EX/MEM[Branch\ Target\ Address])$. Applying similar logic to that of PC, the bits in branch target address field in the pipeline register are deemed vulnerable or non-vulnerable based on the entry in the CFC protection table for the corresponding $pc \rightarrow npc$ transition. The bits can be considered non-vulnerable if the instruction in the current cycle is a non-branch instruction. To measure the vulnerability of the *branch/non-branch control bit* in the EX/MEM pipeline register (the bit that leads to the MUX select bit for the multiplexer that chooses between PC+4 and branch target), we consider two cases: (i) The bit indicates a branch instruction and is flipped to indicate a non-branch instruction in the current cycle, and (ii) vice versa. In the first case, the current PC value is supplied to the adder from the PC itself to increment by 4 considering the instruction in the current cycle to be non-branch instead of taking the value from the branch target address field in the EX/MEM pipeline register for a normal branch instruction. The corresponding *npc* values are $npc = pc + 4$. In the second case, the erroneous PC value is read from the branch target address field in the EX/MEM pipeline register considering the instruction in the current cycle to be a branch instead of supplying the value of PC to the adder to increment by 4 for the actual non-branch instruction. The corresponding *npc* value is $npc = EX/MEM[Branch\ Target\ Address]$.

Moving to the fields in the decode-execute (ID/EX) pipeline register, for the *branch offset* field in the ID/EX pipeline register, a bit flip can cause an erroneous value to calculate branch target address for the next cycle, provided the *branch/non-branch control bit* in the ID/EX pipeline register shows a *branch* instruction in the current cycle. The corresponding *npc* value is $npc = H1(ID/EX[Branch\ Offset]) << 2 + ID/EX[PC]$.

For the PC field in the ID/EX pipeline register, the same equations can be used except that the 1-bit hamming distance is applied in that pipeline PC field. For the PC and branch offset fields in the fetch-decode (IF/ID) pipeline register, the same equations apply. For the opcode field in the IF/ID pipeline register, the same logic applies for *npc* calculation as the *branch/non-branch control bit* in the ID/EX pipeline register, except that the *npc* values for every 1-bit hamming distance

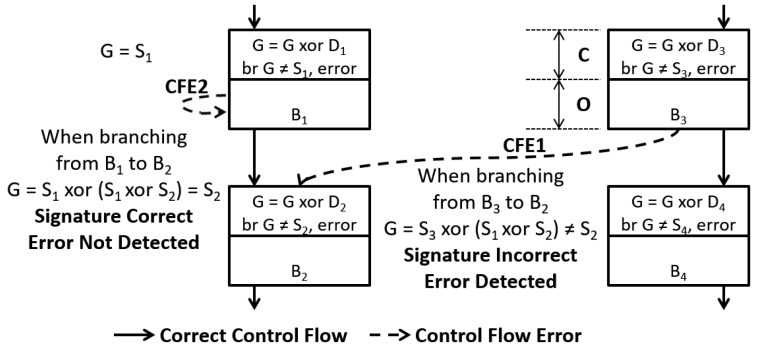


Fig. 6. In CFCSS [28], checking-instructions are added to each basic block to set a variable to the basic block signature; and to check if the execution is coming from a correct predecessor basic block. When execution flows to B2 from B1 (a correct predecessor of B2) the signature checks out, but if the execution flows from B3 (not a correct predecessor of B2), a CFE happens.

opcode that can transform a branch instruction to a non-branch instruction and vice-versa have to be considered. A detailed analysis of the faults in pipeline register fields is presented in [33].

3.1.3 Rest of the Components. We assume that both instruction cache and data cache have parity protection. For instruction cache, parity protection is enough, as it is not dirty, so whenever we detect a fault, a correct copy of instructions can be copied from the protected lower levels of memory. However, for data cache, parity protection is not enough and ECC protection has very high overheads. Faults in register file, data caches, ROB, IQ, and LSQ typically cannot cause *not-successor* CFEs; they can only cause *wrong-successor* CFEs. Therefore, their vulnerability remains the same before and after CFC. There is one exception though, and that is when the value in memory/register is used as target address to jump. This happens only in a few cases: (i) during function call, when the return address is saved in link register, (ii) switch case implementation where depending on the condition, the execution jumps to an address that is stored in an array, (iii) jump table where depending on the index of the function call, the execution jumps to a function address that is stored in an array, and (iv) faults in data in register/memory, where the data is consumed by instructions that may cause processor exceptions and exception handlers are called. We monitor the first three situations, and assume that the register and the memory location that contain the jump value for both the function call, and switch case and the associated instruction sequences are protected by the CFC techniques. The register/memory containing data for the fourth case is considered to be vulnerable in our control flow checking analysis. In addition, we find that none of the existing CFC techniques, to the best of our knowledge, detect a deviation in control flow to a processor exception handler due to fault in data in register/memory.

3.1.4 Wrong-Successor Control Flow Errors. There are more fields in the components, but they cause *wrong-successor* CFEs. Since existing CFC techniques are unable to detect *wrong-successor* CFEs, we do not need to model the protection offered by CFC techniques to those bits.

3.2 Given that Fault Results in a CFE, can a CFC Catch It?

This section describes the CFC schemes in detail and explains which CFEs can occur in the presence of these CFC techniques and will be detected by them, and maps CFEs to the protection model offered by the techniques.

PC	NPC	Detected or Not
O	**OS**	**Not Detected**
O	**OD**	Detected
O	**CW**	**Not Detected**
O	**CA**	**Not Detected**
O	**CO**	Detected
C	**OS**	Detected
C	**OD**	Detected
C	**CW**	Detected
C	**CA**	Detected
C	**CO**	Detected

Legend	Portion of program basic blocks identified by legend
O	Basic block in original source code before inserting CFC code
OS	Instructions in same original basic block
OD	Instructions in different original basic block
C	Instructions in CFC code of a basic block
CO	CFC instructions other than first instruction in the CFC code
CW	CFC code in wrong successor of original basic block
CA	CFC code in aliased successor of original basic block

Fig. 7. CFE Protection Table—For each CFC technique, we build this table, which shows whether the CFC technique can detect each kind of erroneous $pc \rightarrow npc$ transition. This table shows the protection model for the CFCSS technique.

3.2.1 Analysis of CFCSS. Consider the working of CFCSS shown in Figure 6. G or *Runtime Signature Register* holds the identifying signature of a basic block during runtime. At the start of the basic block B2, the signature of the predecessor basic block B1 is expected in G. The CFC code at the beginning of the B2 transforms this expected value in G to match the signature of B2 using *xor* operation. It compares G with the signature value of B2 and a mismatch will flag an error. If the execution flows from basic block B3 to the first instruction of B2 due to a fault in the PC during execution in B3, the CFCSS signature checking code will detect it. G will be holding the signature of B3 while executing instructions in B3. When it jumps to B2, G holds an incorrect signature value of B3, triggering a call to flag the error. Therefore, the bits in the PC of instructions in B3 (or any basic block other than the correct predecessor of B2) which can become the address of the first instruction in B2 are no longer vulnerable after CFCSS is implemented. Therefore, an incorrect $pc \rightarrow npc$ transition to a different basic block (or an incorrect successor) can be detected by CFCSS. On the other hand, CFCSS cannot detect the CFE when execution jumps to an instruction in the same basic block. If an incorrect branch happens within the basic block B1, G will still hold the signature of B1 while executing B1, and no error will be flagged during the signature match in B2.

In order to comprehensively model the protection achieved by a CFC scheme, we generate a table called a *CFC protection table*, which lists all the categories of $pc \rightarrow npc$ transitions, and indicates if they will be caught or not. The table in Figure 7 shows the protection model of CFCSS. The first column shows the location of *pc*, while the second column shows the location of *npc*. The third column lists whether the erroneous control flow that happens from $pc \rightarrow npc$ will be caught by the CFC or not. For example, the first row in the table shows that CFCSS cannot catch if the CFE causes a $pc \rightarrow npc$ transition from an instruction in the original source code of a basic block (*O*), to the original source code of the same basic block (*OS*). The second column states that if the CFE causes an erroneous $pc \rightarrow npc$ transition from an instruction in the original source code of a basic block (*O*) to the original source code of any other (different) basic block (*OD*), then CFCSS will catch it.

3.2.2 Analysis of CFCSS+NA. Figure 9 takes us through the working of the CFCSS+NA [8] scheme and how it tackles the problem of aliasing. In the case when multiple blocks share multiple successor nodes with the successor nodes having multiple predecessor nodes, owing to the mechanism of signature assignment and verification in CFCSS, aliasing of the signatures renders certain CFEs undetectable. For example, in this case where B4 and B5 share a common predecessor B2 and also have their independent predecessor B1 and B3, respectively, CFCSS selects the base for runtime adjusting signature D to be the signature of common predecessor B2 and the signature

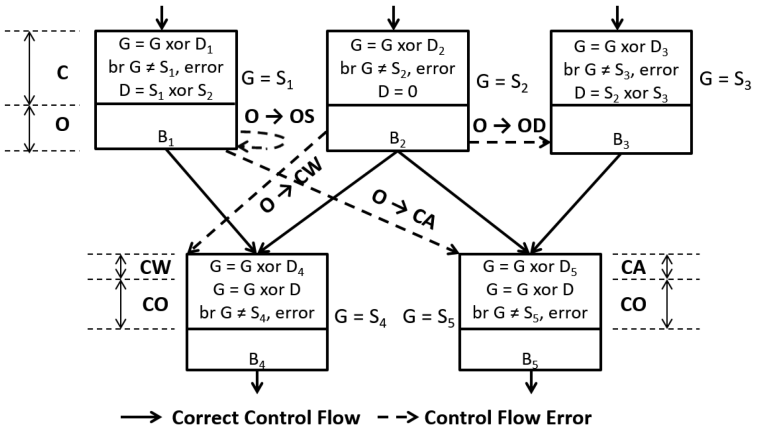


Fig. 8. When a basic block has more than one predecessor, then CFCSS requires a runtime adjusting signature (D) to uniquely identify where the execution is coming from. However, when there is a W-like control flow (formed by the control flow edges from B1 to B4, B2 to B4, B2 to B5, and B3 to B5), CFCSS is unable to uniquely identify the execution path. If a control flow fault causes the execution to jump from B1 to B5, CFCSS is unable to detect that, and it thinks that the execution is coming from B3 to B5. This is called aliasing, and B5 is an aliased block of B1.

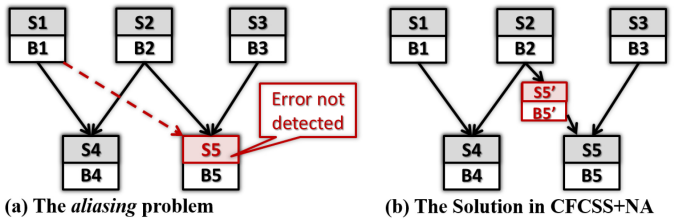


Fig. 9. CFCSS+NA [8]: Demonstrating the aliasing problem in CFCSS implementation, with an example CFG.

checking code is generated as shown in Figure 8. Here, B5 can be considered as an aliased target of B1, or the independent successor of the predecessor B2 with which B1 shares the common successor B4. If a CFE causes a *pc* in B1 to erroneously transition to the *npc* of first instruction in B5, the aliasing problem renders such CFEs undetectable since the signature checking code in B5 will pass. The solution to the aliasing problem is provided by CFCSS+NA as shown in Figure 9 by splitting of the critical edge [7]. Here, between the blocks that form a W-shaped structure, one of the transition edges is interrupted by a dummy block $B5'$ with a unique signature $S5'$ of its own. This fix ensures that no two basic blocks in a given program share a common predecessor. The only difference in the protection model for CFCSS+NA compared to CFCSS is the erroneous transition of $pc \rightarrow npc$ from O to CA can be detected in CFCSS+NA, as shown in the table in Figure 10, where O represents the original source code and CA represents the CFC aliased target as shown in Figure 8.

3.2.3 Analysis of CEDA. CEDA [39] is one of the state-of-the-art techniques in the field of software-only CFC. As illustrated in Figure 11, this CFC detection mechanism adds signature verification code at the start and end of each basic block. It detects aliasing errors by maintaining unique signatures even for the aliased blocks and is one of the few techniques to detect incorrect conditional branches or *wrong successor CFEs* by employing jump check instructions (although

PC	NPC	Detected or Not	PC	NPC	Detected or Not
O	OS	Not Detected	C	OS	Detected
O	OD	Detected	C	OD	Detected
O	CW	Not Detected	C	CW	Detected
O	CA	Detected	C	CA	Detected
O	CO	Detected	C	CO	Detected

Fig. 10. This table shows the protection model for the CFCSS+NA [8] technique.

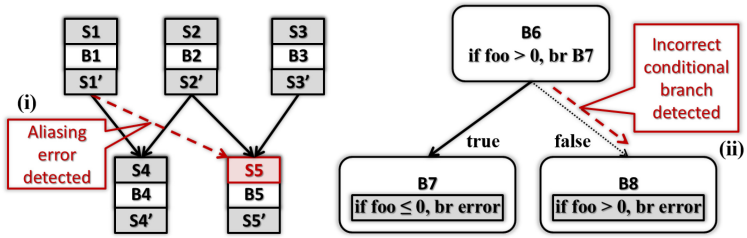


Fig. 11. CEDA [39]: The original program with five basic blocks ($B1$–$B5$) is shown with the CEDA implementation of signature-checking code headers and footers ($S1$–$S5$ and $S1'$–$S5'$) to detect CFEs during its execution. The detection of (i) aliasing errors and (ii) incorrect conditional executions is shown.

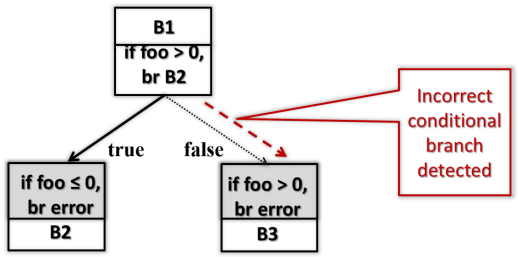


Fig. 12. The jump check instructions in CEDA provide protection to only the condition flags in the processor status register. It cannot detect indirect errors in the variables that propagate to the conditional variable *foo* that decides the branch outcome.

PC	NPC	Detected or Not	PC	NPC	Detected or Not
O	OS	Not Detected	C	OS	Detected
O	OD	Detected	C	OD	Detected
O	CW	Detected	C	CW	Detected
O	CA	Detected	C	CA	Detected
O	CO	Detected	C	CO	Detected

Fig. 13. This table shows the protection model for the CEDA [39] technique.

with limited success as detailed in the later part of this section). Figure 12 shows an instance of the conditional branch in B6 which is supposed to direct the branch execution to B7, but due to a soft fault, the B6 to B8 branch is taken. The jump check instruction, *iffoo* $>$ 0, *br error*, in the B8 header which rechecks the condition will detect such an incorrect conditional branch and flag a CFE. The CEDA protection model, as shown in the table in Figure 13, captures the solutions to aliasing and

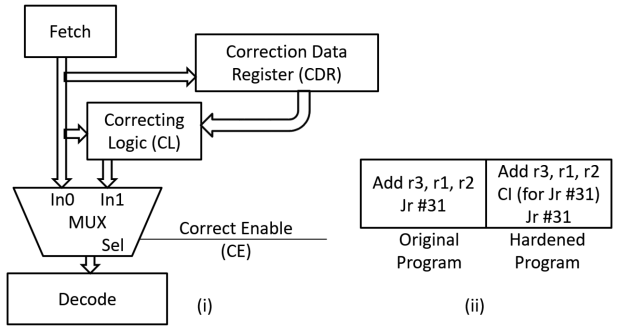


Fig. 14. CFEDC [12]: (i) Working of CFEDC. (ii) Program hardening.

PC	NPC	Detected or Not
OB	OB	Detected
OB	OO	Detected
OB	C	Detected
OO	OB	Not Detected
OO	OO	Not Detected
OO	C	Not Detected
C	OB	Detected
C	OO	Not Detected
C	C	Not Detected

Legend	Portion of program basic blocks identified by legend
OB	Branch instruction in original basic block
OO	Instructions other than branch in original basic block
C	Instructions in CFC code of a basic block

Fig. 15. This table shows the protection model for the CFEDC [12] technique.

to some limited extent, for wrong-successor CFEs in the erroneous transition of $pc \rightarrow npc$ from O to CA and from O to CL as detected.

CEDA is one among the few CFC techniques that attempt to protect one case of *wrong-successor* CFEs through *jump-check* instructions. This scheme can protect the processor status register (that holds the condition flags of the comparison) for the short duration (between the two comparisons). However, if there was a fault in the variable that was used in the comparison, then these *jump-check* instructions cannot detect it. We monitor this in the evaluation of the CEDA scheme and assume that it protects the whole processor status register.

3.2.4 Analysis of CFEDC. CFEDC [12] is a hybrid CFC technique that detects and corrects CFEs. The methodology involves two stages: (i) *Program hardening*: The *control instruction CI* is a hamming code of the subsequent branch instruction as shown in Figure 14. During compilation, a correction code for the control instruction CI, called *correction data CD*, is inserted before each branch instruction in the program. CD from the program is introduced into the pipeline before the decoding of the control instruction. (ii) *Correcting hardware*: The CD from the program is processed by specialized hardware within the pipeline to detect any errors during the $fetch$ stage of the control instruction. In the case of a detected error, the *Correcting Logic CL* together with the *Correction Data Register CDR* is used to introduce the correct control instruction into the pipeline register of the *decode* stage.

The CFEDC protection model is shown in the table in Figure 15. CFEDC can detect only CFEs occurring in two types of erroneous $pc \rightarrow npc$ transitions: (i) from the branch instructions (represented by OB or original source code branch instructions) to any other instructions like OB, OO

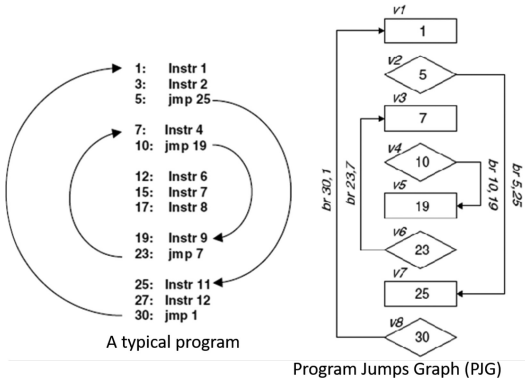


Fig. 16. CFCET [32]: A typical program with its Program Jumps Graph (PJG) is shown. PJG is used as a reference graph by the watchdog processor in CFCET.

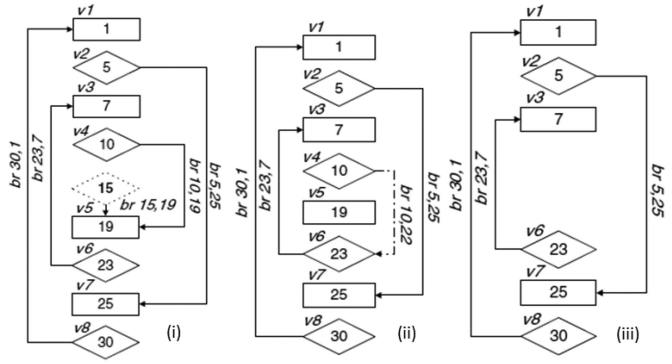


Fig. 17. CFCET [32]: Different CFEs and their effects on PJG are shown here; for example, (i) branch insertion at address 15, (ii) branch target modification at address 10, and (iii) branch deletion at address 10.

(which represent other source code in the original basic blocks), or *C* (which stands for the CFC code or the control instruction CI), and (ii) from the CFC code (*C*) to branch instructions (*OB*), since the correction data generated from the correction logic CL during the fetch of the branch instruction and the value in the correction data register CDR from the CI instruction will cause a mismatch.

3.2.5 Analysis of CFCET. CFCET [32] employs a watchdog processor along with the internal execution tracing feature available in most COTS processors (e.g., Branch Trace Messaging or BTM available in the Intel Pentium family [19]) to verify the control flow execution of a program on the main processor. The CFC scheme traces the *program jumps graph (PJG)* at runtime and compares with a reference jumps graph generated at compile time to detect a possible alteration in control flow. As shown in Figure 16, PJG represents the control flow of the program where the rectangular nodes represent basic blocks excluding the branch instructions at the end of the basic blocks, the diamond nodes represent the branch instructions, and the branches represent the jump between the branch instructions and its correct branch targets. The reference PJG is loaded in the associative memory of the watchdog processor. A branch insertion or a branch target modification causes a mismatch when compared to this reference PJG, as shown in Figure 17, thereby leading to the detection of the CFE. However, a branch deletion cannot be detected using CFCET since it

PC	NPC	Detected or Not
OB	O	Detected

Legend	Portion of program basic blocks identified by legend
OB	Branch instruction in original basic block
O	Basic block in original source code before inserting CFC code

Fig. 18. This table shows the protection model for the CFCET [32] technique.

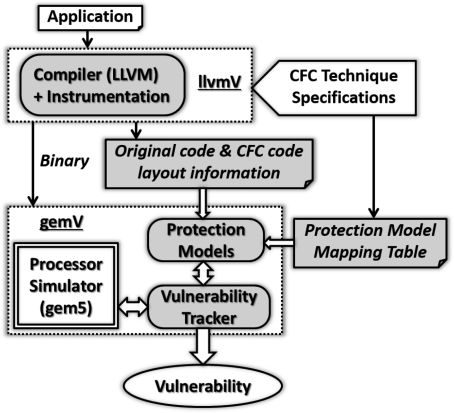


Fig. 19. Overview of the *gemV+llvmV* framework highlighting (i) the *LLVM* compiler setup with methods for instrumentation for software based control flow protection techniques and (ii) the *gem5* simulator setup including the *vulnerability tracker* and *protection model*.

causes a non-branch instruction to execute instead of a branch instruction, but BTM cycles will not be generated as BTM is enabled only for branch instructions.

The CFCET protection model, as shown in the table in Figure 18, has a relatively simplified erroneous $pc \rightarrow npc$ transition case since it can detect only branch insertions and branch target modifications. Branch insertions and branch target modifications are represented by transitions from *OB* or branches in the original source code to *O* or any original source code instructions. The rest of the cases like branch deletion (as explained earlier in this section), erroneous $pc \rightarrow npc$ transitions due to soft faults in PC (same explanation as branch deletion), and wrong-successor CFEs (since both the branches will be present in the reference PJG) go undetected.

4 GEMV-CFC IMPLEMENTATION

In order to model the protection offered by CFC techniques, we built a framework called *gemV+llvmV* (V stands for vulnerability), as shown in Figure 19, based on the gem5 simulator [5] and LLVM compiler [22]. gemV models the architectural vulnerability of all microarchitectural components, including caches, register files, pipeline registers, reorder buffers, load store queues, and so forth. We have implemented various state-of-the-art software CFC technqiues like CFCSS, CFCSS+NA, and CEDA, and the software portion of the hybrid technique CFEDC in the llvmV compiler.

4.1 Vulnerability Modeling in gemV

To calculate the vulnerability of microarchitectural components in a processor, we implemented two software modules in gem5, *vulnerability tracker* (VT) and *protection model* (PM), and plugged the modules to each microarchitecture component in the gem5 simulator, as shown in Figure 20. The *Vulnerability Tracker* functions as a wrapper around each processor component's

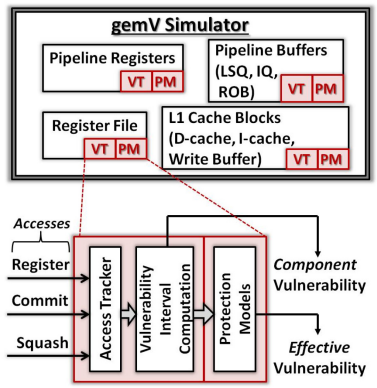


Fig. 20. Detailed description of the gemV simulator setup, in which the *Vulnerability Tracker* (VT) and *Protection Model* (PM) implementations are highlighted for each of the architecture components. One such VT+PM block is expanded, and its interface with the component accesses is described in detail, and the flow toward deriving the *vulnerability* outputs are indicated.

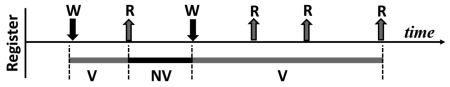


Fig. 21. The vulnerability tracking for a register in the register file.

functional model to monitor its accesses, and computes the vulnerability of the attached processor component. Vulnerability of a component begins from the time data is written into the component until the time it is last used by the processor or removed from the component. An access can be considered final and binding iff the instruction that initiated the access is committed. In the event of a branch misprediction, the speculatively executed instructions are *squashed* in the processor pipeline, and the associated accesses should not be counted. Once the accesses that effectively contribute to the vulnerability of the component are committed, the vulnerability intervals are computed. The vulnerability intervals of a register in a register file during a sequence of reads and writes to the register are shown in Figure 21. These vulnerability intervals for each unit (entry) of the component accessed during execution are accumulated to derive the total component vulnerability. The vulnerabilities of individual processor components are then accumulated to obtain the total system vulnerability.

4.2 Pipeline Vulnerability Modeling

No publicly available tool models the vulnerability of pipeline registers. This is because even though performance simulators are cycle-accurate, they are not bit-accurate. They do not model all the bits in the pipeline to achieve speedup over RTL, and as a result, they cannot model the reads and writes to the bits, and therefore, cannot model their vulnerabilities. We need to model the pipeline vulnerability more accurately, since protecting some of the fields of pipeline registers is one of the main goals of CFC techniques. Our accurate pipeline vulnerability modeling is based on detailed RTL analysis of the pipeline of ARM Amber core [1]. We divide the bits in the pipeline register into logical fields, and a field is considered vulnerable if a fault in that field may cause incorrect execution, which includes wrong results, generating an exception, or a CFE. We further classify the vulnerability of the fields as per instruction as their vulnerability is most closely associated with the opcode of the instruction. To find the vulnerable fields of a pipeline register, we

perform detailed fault injection experiments. We find that even though the pipeline stages have more than 1,646 bits, only about 390 of them are vulnerable for the *adc* instruction, for example. A detailed analysis of the pipeline register fields, and which fields are vulnerable in each pipeline register for each opcode, while omitted here for lack of space is presented in [33].

4.3 Validation of Vulnerability Estimation

None of the currently available vulnerability estimation tools [6, 13, 27] have presented validation results. Validating architectural vulnerability modeling and its implementation is quite difficult as it may require correlation against fault injection. However, this is impractical as it is computationally prohibitive, as explained earlier.

We have partially validated our vulnerability models and implementation through fault injection experiments. We compare $\frac{v}{bc}$ from gemV, and $\frac{fs}{as}$ from fault injection experiments. Correlation coefficient (CoC) is defined as $CoC = (\frac{v}{bc} \times 100)/(\frac{fs}{as})$. From gemV, v is the estimated vulnerability of a processor component over the execution of a program (v stands for vulnerable <bit, cycle> pairs, defined in Section 2), b is the number of bits in that processor component, and c is the number of cycles for which the program executes. From the fault injection experiments, fs is the number of simulations that failed, and as represents the total number of simulations. Since fault injection is computationally prohibitive, we perform exhaustive fault injection for some random locations in each microarchitectural processor components, e.g., register 2 of the RF, entry 1 of the ROB, and so forth.[1]

For processor components with array-like structures (e.g., PC, register file, caches), we inject faults and let the execution continue to see if the fault will cause an error. For processor components like pipeline registers, we use the results of RTL analysis. If a pipeline field has been identified as vulnerable for an opcode, then we assume that fault in that field when the pipeline register carries the instruction with the given opcode will definitely cause an error. For other buffers in the architecture (e.g., ROB, LSQ), we assume that the whole entry is vulnerable if it is live.

Since we are only modeling architectural vulnerability, these fractions may not match. This is because of *software masking*. Software masking is the effect where, even though a variable is used, its value does not affect the result. For example, if one operand of a multiplication operation is zero, then fault injection experiments will not fail on an error in the second operand, while the vulnerability simulator will estimate the vulnerability of the second operand. To avoid this mismatch in our validation experiments, we execute programs that have minimal software masking. We get pretty high correlation (see Table 1) of the vulnerability against fault injection results. Since we are using normalized effective vulnerability as the ratio of system vulnerability after and before control flow checking, software masking should not affect the ratio as long as software masking is not changing the overall system vulnerability by a significant margin, although the absolute system vulnerability numbers in the experiments are affected by the overestimation due to software masking.

4.4 Protection Modeling in llvmV and gemV

The protection offered by the CFC techniques cannot be modeled solely in the simulator, since information about the implementation of the software techniques is required in the simulator. This information must be generated in the compiler, and provided to the simulator, separately. We implement the software CFC protection techniques in LLVM, and generate the binary of the application with and without the CFC protection, as well as the extra information needed by the simulator, as shown in Figure 19. For example, for the CFC technique CFCSS, where basic block

[1] In this sense, our validation is incomplete.

Table 1. Correlation Coefficient between $\frac{v}{bc}$ (SER from GemV-CFC) and $\frac{fs}{as}$ (SER from Fault Injection Experiments), Where Soft Error Rate (SER) is Defined as System Failure Rate Due to Soft Errors

Benchmark	Correlation coefficient between $\frac{v}{bc}$ and $\frac{fs}{as}$
Matrix Multiplication	100.26
Vector Dot Product	100.11
Vector Addition	99.71
Matrix Determinant	99.99
Fibonacci	100.34
Factorial	99.97
Vector Cross Product	99.98
Permuations and Combinations	99.98

Correlation coefficient (CoC) is defined as $CoC = (\frac{v}{bc} \times 100)/(\frac{fs}{as})$.

Table 2. Experimental Setup for the Case Study Demonstration of the GemV-CFC Framework

Compilation Environment	
Compiler	LLVM (ARM v7-a)
Cross-compiler	CodeBench gcc (ARM v7-a)
Simulation Environment	
Mode	System Emulation mode
Architecture	ARM v7-a
Pipeline	5-stages (Out-Of-Order)
L1 D-Cache L1 I-Cache	64KB (Two-way) 32KB (Two-way)
D-TLB / I-TLB	64 entries
Physical Reg (INT/FP) Architecture Reg (INT/FP)	128/128 16/32

boundaries are required to determine whether it can detect the CFE or not, we develop compiler APIs to identify and label the boundaries. The instrumented binaries are then interpreted by the *protection models* in the simulator to compute effective system vulnerability in the presence of CFC techniques.

The *Protection Model* software module attached to the simulated model of each processor component, as shown in Figure 20, evaluates the protection offered to the component. PM implements the systematic methodology to determine the cycles during which soft faults in the microarchitectural component bits can cause CFEs that can be detected by CFC schemes, by looking up the respective CFC scheme's protection tables. Such `bit-cycles` are accumulated and "negated" from the vulnerability derived from the corresponding *Vulnerability Tracker*, thus computing its *effective vulnerability* after protection.

5 HOW EFFECTIVE ARE CFCS?

We perform our experiments and quantitative estimation of the effectiveness of CFC schemes on our gemV+llvmV framework for the ARM v7-a architecture on MiBench benchmarks [16]. More details of the experimental setup are provided in Table 2. We have implemented state-of-the-art

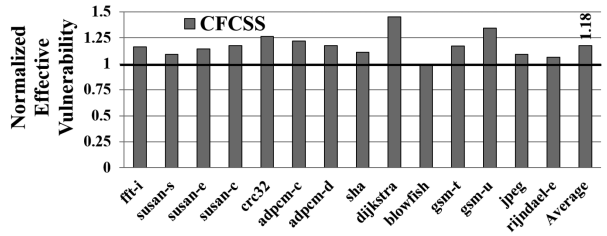


Fig. 22. The effective system vulnerability (normalized over original program vulnerability) of CFCSS, for a ECC protected L1 data cache.

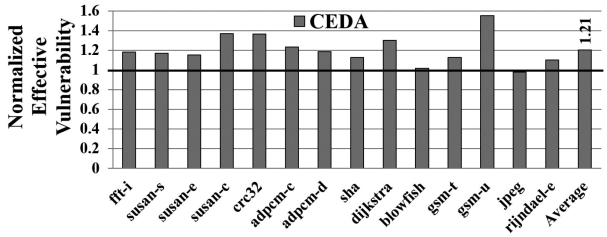


Fig. 23. The effective system vulnerability (normalized over original program vulnerability) of CEDA, for a ECC protected L1 data cache.

software CFC techniques like CFCSS, CFCSS+NA, and CEDA, and the software part of the hybrid technique CFEDC in llvmV compiler. The compiler also generates the information about the location of CFC instructions within each basic block, and the location of the original source code to detect the category of erroneous $pc \rightarrow npc$ transitions in *CFC protection tables*, in the assembly file. Program binary and the object dump files are generated from the assembly file by cross compiling with the gcc cross-compiler for ARM. The object dump file is parsed to obtain the instruction addresses corresponding to the CFC and the original basic block locations in the assembly file. This is provided as an input to the gemV simulator to model the protection achieved by the CFC techniques. For each CFC technique, we construct the *CFC protection table* and provide it as an input to gemV.

5.1 Vulnerability Increases on Applying CFC, Making It Ineffective!

Figure 22 plots the vulnerability of benchmarks on implementing CFCSS, normalized to the vulnerability of the original benchmarks without applying CFCSS. On average, CFCSS increase the vulnerability of the MiBench benchmarks by 18%, although the intent of the CFC technique is to reduce the vulnerability of the programs. In fact, for *dijkstra* benchmark, applying CFCSS increases its vulnerability by more than 40%.

Similarly, on applying state-of-the-art software CFC techniques like CEDA and CFCSS+NA increases the vulnerability of these benchmarks by 21% and 18%, respectively (as shown in Figure 23 and Figure 24), while the hybrid CFEDC scheme increases their vulnerability by 5% on an average, as shown in Figure 25. Although the vulnerability on applying CFCET remains the same as shown in Figure 26, the additional design, area, power, and cost overheads cannot be ignored.

The CFC code added to the original program by the software-only CFC techniques adds to the execution time of the original program. Compared to the original program, the CFC code in the program code generated by software-only CFC techniques reduces the performance of the program execution on the processor and also increases the executed instructions. In effect, it increases

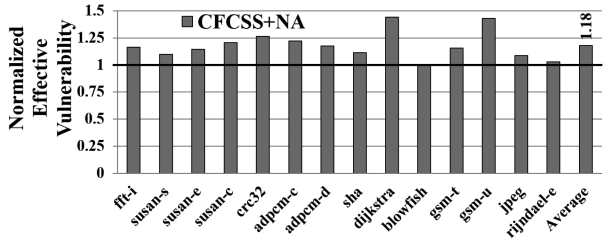


Fig. 24. The effective system vulnerability (normalized over original program vulnerability) of CFCSS+NA, for a ECC protected L1 data cache.

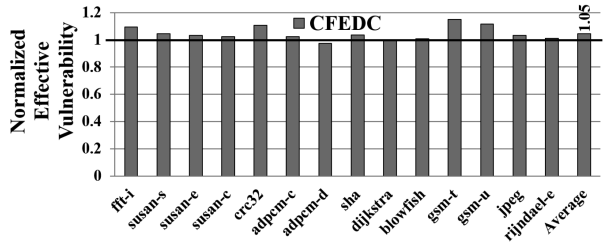


Fig. 25. The effective system vulnerability (normalized over original program vulnerability) of CFEDC, for a ECC protected L1 data cache.

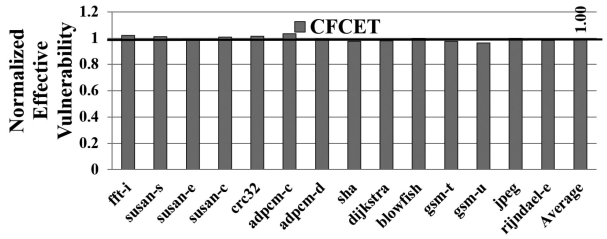


Fig. 26. The effective system vulnerability (normalized over original program vulnerability) of CFCET, for a ECC protected L1 data cache.

the vulnerability of the program and negates even the small amount of protection achieved by these CFC techniques. The overhead in vulnerability and execution time added by the CFCSS code inserted by the compiler is plotted in Figure 27. We observe that in most benchmarks, the added CFC code contributes significantly to the total effective system vulnerability, thereby adding (32.5%) instead of reducing system vulnerability. The reason for this is an average 19.5% increase in simulation runtime added on by an additional 32.5% executed instructions. We can also see that in the cases where the CFC code contributes a larger fraction of the system vulnerability, it also contributes a larger fraction toward the runtime. In other words, runtime increase and vulnerability increase go hand-in-hand.

To expose the impact of performance on vulnerability increase by CFC techniques, Figure 28 plots the normalized vulnerability per-instruction and per-cycle of the program execution. The normalized vulnerability per execution cycle is almost 1, asserting the fact that the increase in vulnerability follows the trend of the increase in execution time. Owing to the increased number of CFC instructions executed in software based protection techniques (five to seven instructions per basic block), the increase in vulnerability is proportionately less compared to the increase in the

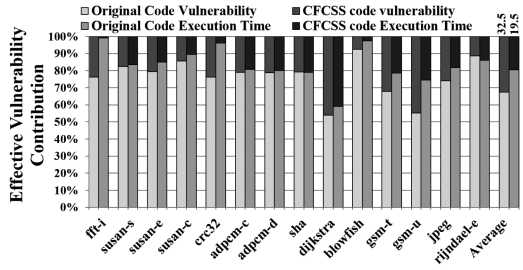


Fig. 27. The percentage contribution of original code and the CFC code toward the effective system vulnerability and the execution time with CFCSS protection.

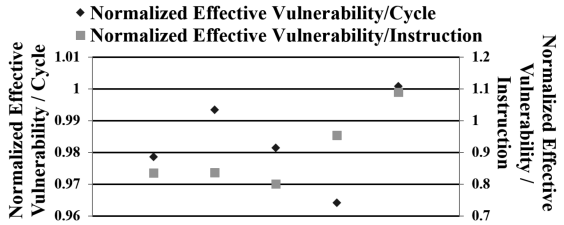


Fig. 28. The normalized effective system vulnerability per cycle and per instruction upon implementation of (a) CFCSS, (b) CFCSS+NA, (c) CEDA, (d) CFEDC, and (e) CFCET.

number of dynamic instructions executed. The normalized vulnerability per instruction in hybrid technique, CFEDC, is slightly higher (0.95) compared to SW based CFC techniques (0.8), since it uses only one instruction per basic block for CFC. In the case of CFCET, although the executed instructions remain the same, the vulnerability increases due to the extra execution cycles required for execution tracing, thereby increasing the residency of vulnerable data in processor components and reflecting in an increase of 9% in normalized vulnerability per instruction.

6 WHY CFCS ARE NOT EFFECTIVE?

The ineffectiveness of existing CFC techniques lies in their inability to provide reasonable protection against *wrong-successor CFEs*. Existing CFC techniques only attempt to detect a subset of soft faults that cause *not-successor* CFEs. The catch lies in the fact that only a small fraction of the soft faults cause *not-successor* CFEs, while most of them cause *wrong-successor* CFEs. For example, in our experimental setup, even without considering cache, the average vulnerability over all MiBench applications is 1.72×10^{14} bit-cycles.[2] Even if only 33% of these can cause CFEs [35], it implies that faults in $1.72 \times 10^{14} \times 0.33 = 5.7 \times 10^{13}$ bit-cycles can cause CFEs. Out of these, only 2.3×10^{12} bit-cycles are protected by CFCSS, according to our runs using gemV-CFC. This implies that CFCSS can detect only 4% of all the soft faults that cause CFEs. On average, *not-successor* CFEs can happen in CFCSS protected MiBench benchmarks in 2.64×10^{12} bit-cycles, as per our estimates from the simulation runs. Therefore, the rest of the bit-cycles can cause *wrong-successor* CFEs (5.44×10^{13} bit-cycles), which constitutes a large fraction (95.4%) of possible CFEs, and are not protected by any of the existing CFC techniques. This is the main reason why existing CFC techniques are not effective in protecting from soft faults. This is true even for several other CFC techniques that we have not evaluated, since none of them attempt to provide protection from *wrong-successor* CFEs.

[2]If you consider cache vulnerabilities also, then the number is 5.4×10^{17}.

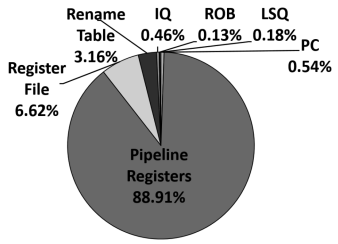


Fig. 29. Vulnerability distribution in the processor components.

Table 3. Protection Achieved in Processor Components in the gemV+CFC Framework for the Case Study Experiments

	Protection Achieved			
Protection Technique	**PC**	**Pipeline Register**	**Link Register**	**Processor Status Register**
CFCSS	91.03%	0.72%	100%	0%
CFCSS+NA	91.03%	0.73%	100%	0%
CEDA	93.19%	1.07%	100%	100%
CFEDC	12.8%	0.21%	0%	0%
CFCET	N/A	0.4%	0%	0%

We analyze the distribution of vulnerability among the processor components, as shown in Figure 29. We find that the pipeline registers (89%) constitute the major vulnerability proportion followed by the register file (6.62%), rename table (3.16%), and then the PC (0.54%). The rest (0.77%) is contributed by ROB, IQ, and LSQ. Since the primary targets of protection by CFC techniques are the PC, pipeline registers, link register, and processor status register, CFC techniques do stand a good chance of providing protection. However, Table 3 shows that although CFC techniques are able to achieve good protection of PC, they are not so effective in protecting the pipeline registers from soft errors. The protection achieved on pipeline registers, which contribute the most to system vulnerability, is grossly inadequate in all these CFC techniques.

7 WHY PREVIOUS EVALUATIONS WERE WRONG?

We further investigate to find out the reasons for the stark difference in our results, and that obtained by the previous researchers, and narrow it down to flaws in the experimental methodology of the previous works. Almost all the previous CFC papers have performed "targeted fault injection." They essentially directly inject faults that will cause *not-successor* CFEs and then perform assumption/analysis/simulation/execution to see if their technique can detect the injected fault or not. Three main approaches have been taken by researchers in targeted fault injection when it comes to how and where to insert faults: (i) Assembly code instrumentation used in CFCSS [28] and CFCSS-NA [8], (ii) gdb-based runtime fault injection used in ACCE [40] and CEDA[39], and (iii) fault injection in memory bus used in OSLC [23] and SIS [36]. Researchers have also employed probability based expressions to increase coverage of fault injections and have even used analytic arguments in many instances to explain the effectiveness of their CFC techniques, while comparing with other schemes. The gdb-based runtime fault injection, fault injection in memory bus, and targeted fault injection methods deployed by CFC techniques, in general, have several shortcomings: (i) The targeted fault injections, directed to analyze the error detection capabilities

of the specific CFC technique implementation, are not representative of the error coverage provided to the processor as a whole. (ii) The targeted nature of the fault injections does not give an accurate measure for comparative analysis among different proposed techniques. (iii) The exhaustive amount of simulation runs, required for reasonable fault injection coverage to get an estimate of SER (Soft Error Rate) (as explained in Section 2), make this methodology unsuitable for quick and extensive design space explorations. (iv) The targeted fault injection methods cannot be easily ported across different processor architectures or system configurations. We will attempt to explain the problems in detail with an assembly code instrumentation approach. Other evaluation schemes also exhibit similar shortcomings.

The assembly instrumentation scheme involves toggling bits in assembly instructions in the benchmark program's assembly/binary code and simulates a soft fault during the execution in the processor pipeline. Several CFE types like a non-branch instruction becoming a branch instruction (branch insertion), a branch turning into a non-branch (branch deletion), branch offset error, and branch predicate error can be simulated based on the instruction the fault is injected into and which instruction bit is corrupted. By counting the number of CFEs detected by the CFC scheme and dividing it by the total number of inserted faults, the effectiveness of a CFC technique can be estimated. This approach can be ineffective in catching CFEs due to at least three problems:

(1) **Coverage of faults in space or across microarchitectural components is not sufficient**: Instrumentation of assembly code instructions maps to simulation of fault injection in the instruction register or the IF/ID pipeline stage or the instruction cache, but it ignores the CFEs caused by the rest of the processor components like program counter, other pipeline stages, register file, or data cache.
(2) **Coverage of faults in time is not enough (how many faults or duration in which faults are injected in a microarchitectural component)**: A random instruction bit at a random address is selected and toggled in the assembly code instrumentation scheme to achieve an even probability of fault injection. This is not an accurate representation of the distribution of transient faults. As per the definition of the metric *Vulnerability*, a bit in a microarchitectural component is vulnerable in a cycle of execution, if a soft fault in the bit leads to an invalid result, and hence the probability of a transient fault is proportional to the amount of time the bit is resident in the component. For example, the instructions inside a loop have a higher probability of exposure to a soft fault, since they are resident in instruction cache for a longer time than non-loop instructions which get quickly replaced in the cache by the next set of instructions.
(3) **This fault injection technique only inserts faults that cause not-successor CFEs**: Lastly, and most importantly, the simulation of fault injection in assembly code instructions only results in what we term as *not-successor CFEs*. They insert very few faults that lead to *wrong-successor CFEs*. Wrong-successor CFEs are typically the result of data corruption, due to soft faults, that lead to the branch condition or branch decision variables. Since fault injection based on assembly code instrumentation does not toggle data bits, this CFC evaluation scheme does not evaluate the effectiveness of CFC techniques in detecting such errors.

8 ADVANCED CFCS

Advanced CFC techniques assume they provide more protection by detecting more types of CFEs. On the contrary, our experiments show that some of the later CFC techniques in pursuit of advancing the effectiveness of CFC tend to increase the effective system vulnerability of the program compared to its predecessors. CEDA is supposedly one of the more advanced software CFC

schemes that tries to plug the gaps in earlier CFC implementations, e.g., by detecting even one variant of wrong-successor CFEs via jump-check instructions and also by detecting aliased errors by maintaining unique signatures even for aliased blocks, but the effective system vulnerability normalized over original program vulnerability increases to 21% compared to 18% in the case of CFCSS and CFCSS+NA. For 10 out of 14 MiBench benchmarks, CEDA has increased the effective system vulnerability of the programs compared to other CFC techniques. In fact, CEDA has an effective system vulnerability normalized to the original program of 1.55, compared to 1.43 for CFCSS+NA, 1.34 for CFCSS, and 1.12 for CFEDC. The additional code overhead in implementing the advanced CFC techniques renders them worse compared to earlier software CFC techniques, as the residency of additional CFC code instructions in different microarchitectural processor components tends to increase vulnerability.

9 SUMMARY AND CONCLUSIONS

Our studies contradict claims by various CFC schemes that they provide significant protection for program execution in processors against soft errors and proves that these schemes make matters worse as they make the programs more susceptible to soft errors. Using our *gemV-CFC* framework, we estimate the *architectural vulnerability* of the program "without CFC" and "with CFC." Our experimental results show that the vulnerability of the program "with CFC" is higher than "without CFC" for software -only and hybrid CFC techniques. This can be attributed to the additional vulnerability due to the extra instructions that implement the CFC scheme offsets the small reduction in vulnerability achieved by the CFC technique. Even though the vulnerability remains almost the same for hardware-only CFC techniques, the overheads incurred in design cost, area, and power due to the hardware modifications required for their implementation cannot be hidden. Also the contribution of reduction in vulnerability by the existing CFC techniques is small, since the total number of bits protected by these CFC schemes is only a small fraction of the total number of vulnerable bits in the processor.

Some other protection techniques targeted at protecting the components like pipeline registers that are not effectively protected by CFC techniques. For example, C-element latch scheme [14] can provide effective protection for pipeline registers and Shield [26] for register files. C-element latch scheme can provide almost 100% protection by duplication of pipeline latches, while Shield can selectively protect the most vulnerable registers using ECC and thus protect integer RF by up to 84%, with reasonable hardware overheads and no performance overhead. Together, they could provide effective protection to the system along with ECC protection for L1 data cache and parity protection for L1 instruction cache.

REFERENCES

[1] 2010. Amber ARM-compatible core :: Overview. http://opencores.org/project,amber.

[2] Z. Alkhalifa, V. S. S. Nair, N. Krishnamurthy, and J. A. Abraham. 1999. Design and evaluation of system-level checks for on-line control flow error detection. *IEEE Trans. Parallel Distrib. Syst.* 10, 6 (June 1999), 627–641. DOI: https://doi.org/10.1109/71.774911

[3] S. A. Asghari, H. Taheri, H. Pedram, and O. Kaynak. 2014. Software-based control flow checking against transient faults in industrial environments. *IEEE Trans. Indust. Inf.* 10, 1 (Feb. 2014), 481–490. DOI: https://doi.org/10.1109/TII.2013.2248373

[4] J. R. Azambuja, M. Altieri, J. Becker, and F. L. Kastensmidt. 2013. HETA: Hybrid error-detection technique using assertions. *IEEE Trans. Nucl. Sci.* 60, 4 (Aug. 2013), 2805–2812. DOI: https://doi.org/10.1109/TNS.2013.2246798

[5] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The gem5 simulator. *SIGARCH Comput. Archit. News* 39, 2 (Aug. 2011), 1–7. DOI: https://doi.org/10.1145/2024716.2024718

[6] A. Biswas, P. Racunas, R. Cheveresan, J. Emer, S. S. Mukherjee, and R. Rangan. 2005. Computing architectural vulnerability factors for address-based structures. In *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA'05)*. 532–543. DOI: https://doi.org/10.1109/ISCA.2005.18

[7] Preston Briggs, Keith D. Cooper, Timothy J. Harvey, and L. Taylor Simpson. 1998. Practical improvements to the construction and destruction of static single assignment form. *Softw. Pract. Exper.* 28, 8 (July 1998), 859–881. DOI: https://doi.org/10.1002/(SICI)1097-024X(19980710)28:8⟨859::AID-SPE188⟩3.0.CO;2-8

[8] Wang Chao, Fu Zhongchuan, Chen Hongsong, Ba Wei, Li Bin, Chen Lin, Zhang Zexu, Wang Yuying, and Cui Gang. 2010. CFCSS without aliasing for SPARC architecture. In *Proceedings of the IEEE 10th International Conference on Computer and Information Technology (CIT'10)*. 2094–2100. DOI: https://doi.org/10.1109/CIT.2010.356

[9] E. Chielle, G. S. Rodrigues, F. L. Kastensmidt, S. Cuenca-Asensi, L. A. Tambara, P. Rech, and H. Quinn. 2015. S-SETA: Selective software-only error-detection technique using assertions. *IEEE Trans. Nucl. Sci.* 62, 6 (Dec. 2015), 3088–3095. DOI: https://doi.org/10.1109/TNS.2015.2484842

[10] M. Duricek and T. Krajcovic. 2014. Interactive hybrid control-flow checking method. In *Proceedings of the 2014 International Conference on Applied Electronics*. 79–82. DOI: https://doi.org/10.1109/AE.2014.7011673

[11] J. B. Eifert and J. P. Shen. 1995. Processor monitoring using asynchronous signatured instruction streams. In *Proceedings of the 25th International Symposium on Fault-Tolerant Computing, 1995, "Highlights from Twenty-Five Years'."* 106. DOI: https://doi.org/10.1109/FTCSH.1995.532620

[12] N. Farazmand, M. Fazeli, and S. G. Miremadi. 2008. FEDC: Control flow error detection and correction for embedded systems without program interruption. In *Proceedings of the 3rd International Conference on Availability, Reliability and Security (ARES'08)*. 33–38. DOI: https://doi.org/10.1109/ARES.2008.199

[13] Xin Fu, Tao Li, and José A. B. Fortes. 2006. Sim-SODA: A unified framework for architectural level software reliability analysis. In *Proceedings of the Workshop on Modeling, Benchmarking and Simulation (held in conjunction with International Symposium on Computer Architecture)*.

[14] K. T. Gardiner, A. Yakovlev, and A. Bystrov. 2007. A C-element latch scheme with increased transient fault tolerance for asynchronous circuits. In *Proceedings of the13th IEEE International On-Line Testing Symposium (IOLTS'07)*. 223–230. DOI: https://doi.org/10.1109/IOLTS.2007.5

[15] O. Goloubeva, M. Rebaudengo, M. Sonza Reorda, and M. Violante. 2003. Soft-error detection using control flow assertions. In *Proceedings of the 18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*. 581–588. DOI: https://doi.org/10.1109/DFTVS.2003.1250158

[16] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the 2001 IEEE International Workshop on Workload Characterization, WWC-4. (WWC'01)*. IEEE Computer Society, Washington, DC, 3–14. DOI: https://doi.org/10.1109/WWC.2001.15

[17] P. Hazucha, T. Karnik, S. Walstra, B. Bloechel, J. Tschanz, J. Maiz, K. Soumyanath, G. Dermer, S. Narendra, V. De, and S. Borkar. 2003. Measurements and analysis of SER tolerant latch in a 90 nm dual-Vt CMOS process. In *Proceedings of the IEEE 2003 Custom Integrated Circuits Conference*. 617–620. DOI: https://doi.org/10.1109/CICC.2003.1249472

[18] J. Hennessy and D. Patterson. 2012. *Computer Architecture: A Quantitative Approach* (5th ed.). Morgan Kaufmann.

[19] Intel Corporation. 1997. *Pentium Processor Family Developer's Manual*. Intel Corporation.

[20] R. Jeyapaul, Fei Hong, A. Rhisheekesan, A. Shrivastava, and Kyoungwoo Lee. 2011. UnSync: A soft error resilient redundant multicore architecture. In *Proceedings of the International Conference on Parallel Processing (ICPP'11)*. 632–641. DOI: https://doi.org/10.1109/ICPP.2011.76

[21] Sammy Kayali. 2000. Reliability considerations for advanced microelectronics. In *Proceedings of the 2000 Pacific Rim International Symposium on Dependable Computing (PRDC'00)*. IEEE Computer Society, Washington, DC, 99. http://portal.acm.org/citation.cfm?id=826038.826937

[22] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO'04)*. IEEE Computer Society, Washington, DC, 75–. http://dl.acm.org/citation.cfm?id=977395.977673

[23] H. Madeira and J. G. Silva. 1991. On-line signature learning and checking: Experimental evaluation. In *CompEuro'91. Proceedings of the 5th Annual European Computer Conference on Advanced Computer Technology, Reliable Systems and Applications*.642–646. DOI: https://doi.org/10.1109/CMPEUR.1991.257464

[24] T. Michel, R. Leveugle, and G. Saucier. 1991. A new approach to control flow checking without program modification. In *Proceedings of the 21st International Symposium on Fault-Tolerant Computing, 1991. FTCS-21. Digest of Papers*. 334–341. DOI: https://doi.org/10.1109/FTCS.1991.146682

[25] G. Miremadi, J. Ohlsson, M. Rimen, and J. Karlsson. 1998. Use of time, location and instruction signatures for control flow checking. In *Proceedings of the DCCA-5 International Conference*.

[26] P. Montesinos, W. Liu, and J. Torrellas. 2006. Shield: Cost-effective soft-error protection for register files. In *Proceedings of the 3rd IBM TJ Watson Conference on Interaction between Architecture, Circuits and Compilers (PAC'06)*.

[27] Shubhendu S. Mukherjee, Christopher Weaver, Joel Emer, Steven K. Reinhardt, and Todd Austin. 2003. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. *IEEE/ACM International Symposium on Microarchitecture* 0 (2003), 29. DOI: https://doi.org/10.1109/MICRO.2003.1253181

[28] N. Oh, P. P. Shirvani, and E. J. McCluskey. 2002. Control-flow checking by software signatures. *IEEE Trans. Reliab.* 51, 1 (March 2002), 111–122. DOI: https://doi.org/10.1109/24.994926

[29] N. Oh, P. P. Shirvani, and E. J. McCluskey. 2002. Error detection by duplicated instructions in super-scalar processors. *IEEE Trans. Reliab.* 51, 1 (March 2002), 63–75. DOI: https://doi.org/10.1109/24.994913

[30] J. Ohlsson, M. Rimen, and U. Gunneflo. 1992. A study of the effects of transient fault injection into a 32-bit RISC with built-in watchdog. In *Proceedings of the 22nd International Symposium on Fault-Tolerant Computing, 1992. FTCS-22. Digest of Papers.* 316–325. DOI: https://doi.org/10.1109/FTCS.1992.243569

[31] L. Parra, A. Lindoso, M. Portela, L. Entrena, F. Restrepo-Calle, S. Cuenca-Asensi, and A. Marínez-Álvarez. 2013. Efficient mitigation of data and control flow errors in microprocessors. In *Proceedings of the 2013 14th European Conference on Radiation and Its Effects on Components and Systems (RADECS'13).* 1–4. DOI: https://doi.org/10.1109/RADECS.2013.6937381

[32] A. Rajabzadeh and S. G. Miremadi. 2006. CFCET: A hardware-based control flow checking technique in COTS processors using execution tracing. *Microelectron. Reliab.* 46, 5 (2006), 959–972.

[33] Abhishek Rhisheekesan. 2012. *Quantitative Evaluation of Control Flow based Soft Error Protection Mechanisms.* Master's thesis. School of Computing, Informatics and Decision Systems Engineering, Arizona State University.

[34] N. R. Saxena and W. K. McCluskey. 1990. Control-flow checking using watchdog assists and extended-precision checksums. *IEEE Trans. Comput.* 39, 4 (April 1990), 554–559. DOI: https://doi.org/10.1109/12.54849

[35] Michael A. Schuette and John Paul Shen. 1983. On-line monitoring using signatured instruction streams. In *Proceedings of the 13th International Test Conference.* 275–282.

[36] Michael A. Schuette and John Paul Shen. 1987. Processor control flow monitoring using signatured instruction streams. *IEEE Trans. Comput.* 36, 3 (March 1987), 264–276. DOI: https://doi.org/10.1109/TC.1987.1676899

[37] Jared C. Smolens, Brian T. Gold, Babak Falsafi, and James C. Hoe. 2006. Reunion: Complexity-effective multicore redundancy. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 39).* IEEE Computer Society, Washington, DC, 223–234. DOI: https://doi.org/10.1109/MICRO.2006.42

[38] Darshan D. Thaker, Francois Impens, Isaac L. Chuang, Rajeevan Amirtharajah, and Frederic T. Chong. 2008. On Using Recursive TMR as a Soft Error Mitigation Technique. http://citeseerx.ist.psu.edu/viewdoc/download?rep=rep1&type=pdf&doi=10.1.1.131.523

[39] Ramtilak Vemu and Jacob Abraham. 2011. CEDA: Control-flow error detection using assertions. *IEEE Trans. Comput.* 60, 9 (Sept. 2011), 1233–1245. DOI: https://doi.org/10.1109/TC.2011.101

[40] R. Vemu, S. Gurumurthy, and J. A. Abraham. 2007. ACCE: Automatic correction of control-flow errors. In *Proceedings of the IEEE International Test Conference (ITC'07).* 1–10. DOI: https://doi.org/10.1109/TEST.2007.4437639

[41] Rajesh Venkatasubramanian, J. P. Hayes, and B. T. Murray. 2003. Low-cost on-line fault detection using control flow assertions. In *Proceedings of the 9th IEEE On-Line Testing Symposium (IOLTS'03).* 137–143. DOI: https://doi.org/10.1109/OLT.2003.1214380

[42] Kent Wilken and John Paul Shen. 1988. Continuous signature monitoring: Efficient concurrent-detection of processor control errors. In *Proceedings of the 1988 International Conference on Test: New Frontiers in Testing (ITC'88).* IEEE Computer Society, Washington, DC, 914–925. http://dl.acm.org/citation.cfm?id=1896122.1896279

Received March 2018; revised November 2018; accepted November 2018