Quantitative Analysis of Control Flow Checking Mechanisms for Soft Errors

Aviral Shrivastava, Abhishek Rhisheekesan, Reiley Jeyapaul, and Carole-Jean Wu
School of Computing, Informatics and Decision Systems Engineering
Arizona State University
Aviral.Shrivastava@asu.edu, Arhishee@asu.edu, Reiley.Jeyapaul@asu.edu, and carole-jean.wu@asu.edu

ABSTRACT

Control Flow Checking (CFC) based techniques have gained a reputation of providing effective, yet low-overhead protection from soft errors. The basic idea is that if the control flow – or the sequence of instructions that are executed – is correct, then most probably the execution of the program is correct. Although researchers claim the effectiveness of the proposed CFC techniques, we argue that their evaluation has been inadequate and can even be wrong! Recently, the metric of ***vulnerability*** *has been proposed to quantify the susceptibility of computation to soft errors. Laced with this comprehensive metric, we quantitatively evaluate the effectiveness of several existing CFC schemes, and obtain surprising results. Our results show that existing CFC techniques are not only ineffective in protecting computation from soft errors, but that they incur additional power and performance overheads. Software-only CFC protection schemes (CFCSS [14], CFCSS+NA [2], and CEDA [18])* ***increase*** *system vulnerability by* 18% *to* 21% *with* 17% *to* 38% *performance overhead; Hybrid CFC protection technique, CFEDC [4] also* ***increases*** *the vulnerability by* 5%*; While the vulnerability remains almost the same for hardware only CFC protection technique, CFCET [15], they cause overheads of design cost, area, and power due to the hardware modifications required for their implementations.*

1. BACKGROUND AND INTRODUCTION

Continuous and exponential technology scaling is responsible for the information revolution and the unprecedented integration of computing systems into our everyday lives. A negative consequence of technology scaling is that the transistors within modern processors have become increasingly susceptible to transient faults. Among the many sources of transient faults (e.g., electrical noise, external interference, cross-talk, etc.) the strike of sub-atomic particles, mainly low and high energy neutrons, cause majority of soft errors in electronic devices [9]. At the current technology node, everyday computing systems, e.g., laptops, smart-phones, tablets, etc., experience a Soft Error Rate (SER) of about once-a-year, but is expected to increase exponentially to about once-a-day in a decade [7]. Reliability is thus rapidly emerging as a primary design metric.

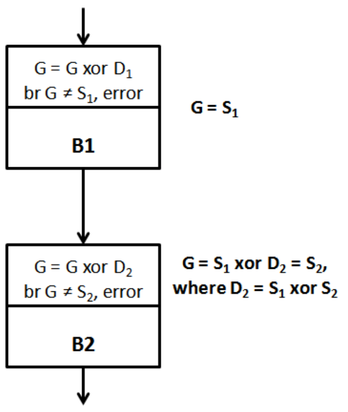


Figure 1: CFCSS[14]: A software CFC scheme. It inserts a few instructions at the beginning of each basic block. These instructions set the outgoing signature, and check the incoming signature register. Checking the incoming signature ensures the correctness of execution flow.

Most techniques to protect programs from soft errors are built around time and/or space redundancy (detailed in [11]). The idea is to perform the same computation twice and see if the results match; if not, then there is an error. Although redundancy based methods have been considered effective in providing system reliability, they are generally considered to have high overhead (around 2X). In particular, even though it may be possible to minimize/hide the performance overhead of redundancy based techniques (through parallelization), their power overheads cannot be hidden.

Control Flow Checking (CFC) techniques were proposed to provide efficient protection from soft errors. The main idea is that most soft errors will eventually manifest as errors in the sequence of instruction execution. Therefore, just by making sure that the sequence of instructions executed (or the control flow of the program) is correct, significant protection can be achieved. Figure 1 shows the basic CFC mechanism. Basically, each basic block in the program is assigned a signature, which is written in a register when the basic block is executed, and when execution flows to another basic block, the signature is checked to confirm if the execution is coming from the right basic block. Note that a CFC technique by itself does not provide any protection – it merely provides the capability of detecting errors. Combined with a scheme to recover from errors (e.g., restart from the beginning; or in the case of regularly created checkpoints, continue from the last checkpoint when an error is detected), CFC techniques can provide protection from soft errors. In this paper, since we are interested in estimating the protection achieved by CFC techniques, we assume that there is some (but it does not matter which one) scheme to recover from the control flow error detected.

The arsenal of control flow based soft error protection techniques span across design layers from hardware [3, 8, 10, 15], software [1, 5, 13, 14, 19, 18, 20], and hardware-software hybrid techniques [4, 16, 17, 21]. CFC techniques are attractive since they can often be implemented with much less overhead (as compared to full scale redundancy) and arguably provide a decent error coverage. Papers proposing CFC techniques perform fault injection tests and conclude that their techniques are quite effective in combating soft errors. However, we argue that the experimentation of these previous papers has been inadequate and even flawed!

Recently, the metric of *Vulnerability* has been proposed [12] to quantitatively estimate the susceptibility of program execution on a processor. A bit is vulnerable in a certain cycle of execution, if a fault in it may cause a wrong result, otherwise, it is not vulnerable. Adding up the number of vulnerable bits in each cycle of the execution of a program, gives us the total vulnerability of the program execution. Higher vulnerability of program execution implies that the program execution is more susceptible to soft errors.

The metric of vulnerability is more comprehensive, and does not require detailed compute-intensive fault injection experiments. In fact, the vulnerability of a program execution can be estimated in a single simulation run by tracking the events on each bit of the processor, and counting the number of vulnerable bits. In this work, we use this metric to evaluate the effectiveness of CFC techniques. While schemes to compute the vulnerability of a program without CFC are known, techniques to estimate the vulnerability of a program with CFC is not known. And that is the primary contribution of this paper. In this paper, we:
(i) expose the shortcomings in the evaluation of protection capabilities of previous CFC techniques,
(ii) propose a systematic methodology to quantitatively estimate the protection achieved by CFC schemes, and
(iii) explain why existing CFC techniques are not effective.

The basic idea is to analyze each vulnerable $< bit, cycle >$ in the original execution, and determine the control flow errors that it can cause. If any of the generated control flow error can be detected by the CFC, then the $< bit, cycle >$ is deemed to be *not vulnerable* in the presence of CFC. We estimate the vulnerability before and after applying CFC techniques. Our results reveal that existing CFC techniques not only do not protect execution from soft errors, but in fact incur additional power and performance overheads. In particular, software only CFC protection schemes (CFCSS [14], CFCSS+NA [2], CEDA [18]) increase system vulnerability by 18% to 21% with 17% to 38% performance overhead. Hybrid CFC protection (CFEDC [4]) increases vulnerability by 5%. Even though the vulnerability remains almost the same for hardware-only CFC protection scheme (CFCET [15]), they incur overheads of design cost, area, and power due to the hardware modifications required for their implementation.

2. WHAT WAS WRONG IN THE EVALUATION OF CFC TECHNIQUES?

Although there are several ways to evaluate the effectiveness of a CFC technique, most researchers have used some form of "targeted fault injection" to do this. This is because most other ways are either inaccessible, difficult to set up, or highly compute-intensive. Even in targeted fault injection, there is a question of how and where to insert faults. There have been 3 main approaches of targeted fault injection: i) Assembly code instrumentation (most popular), used in CFCSS[14], CFCSS-NA[2], CFCSS-IS[22], ii) gdb-based runtime fault injection used in ACCE[19], CEDA[18], ACFC[20], and iii) fault injection in memory bus, used in OSLC[8], SIS[17]. In addition researchers have attempted to use probability based expressions to increase the coverage of fault injections or even just use analytical subjective arguments to claim the superiority of their CFC technique over others. Due to space constraints, we will explain the problems with assembly code instrumentation approach. Similar problems exist with other evaluation schemes also.

In the assembly code instrumentation scheme, the binary or assembly code of the benchmark program is instrumented to corrupt bits in instructions, essentially simulating a soft error in the processor pipeline during program execution. Depending on which instruction the fault is injected into, and which bit in the instruction is toggled several control flow error types, e.g., a branch becoming a non-branch (branch deletion), a non-branch becoming a branch (branch creation), error in branch offset (target address error), branch predicate error (branching error) can be simulated. The effectiveness of a CFC technique is estimated by counting the number of these control flow errors caught by the CFC technique divided by the number of faults inserted.

There are at least three problems in this approach, because of which the number reported does not represent the protection afforded by the CFC scheme against soft errors:

i) Distribution of the faults in space (across microarchitectural components) is not correct: Inserting faults in the binary or assembly simulates the effect of fault injection in instruction cache, or the instruction register (or IF/ID pipeline stage). It does not model the effect of faults in other processor components, e.g., fault in the program counters, other pipeline stages, data cache, register file, or load store buffer. Control flow errors can occur due to faults in these microarchitectural components also. For example, if there is a soft fault in the link register (which contains the address to return from the function call), control flow becomes wrong.

ii) Distribution of faults in time (how many faults in a microarchitectural component) is not correct: In binary or assembly instrumentation scheme, researchers randomly choose a bit in the instruction at a randomly chosen address, and flip it. Such a scheme will result in even probabilities of fault insertion in any bit in the instructions. This does not accurately represent the distribution of transient faults. Since transient faults are evenly distributed in space and time, the probability of a transient fault in bits in a microarchitectural component is proprotional to the amount of time the bit is present in the component. For example, the probability of a fault being inserted in an instruction in cache is proportional to the time it is resident in the cache. And therefore instructions that are inside a loop have a much higher probability of being hit by a fault than an instruction that executes only once, and is quickly replaced by another in the cache.

iii) Only insert faults that cause not-successor control flow errors: Finally, and the most important point is that inserting faults in binary/assembly only results in soft errors that cause, what we term as, "not-successor control flow errors," and insert very few faults that cause "wrong-successor control flow errors."

Definition 1: *not-successor control flow errors* happen when execution flows from an instruction to one that is not a legitimate successor 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 legitimate successor, or to the non-first instruction of a legitimate successor basic block.

Definition 2: *wrong-successor control flow errors* are those that cause execution to go from an instruction to a legitimate, but incorrect successor. For example, if soft error happens in a register altering its value. If a branch decision could be made on wrong value of the register. The execution could still go to a legitimate, but incorrect target.

When a fault is inserted in binary/assembly, then most often it will cause a not-successor control flow error. Wrong-successor control flow errors are typically caused by faults in data. For example, if there is a fault in a value in data cache; it cannot cause a not-successor control flow error. The only way it can affect control flow is if it affects the outcome of a branch, then it has caused a wrong-successor control flow error. Since this methodology does not insert faults in data, such errors are not inserted, and the effectiveness of CFCs in catching these errors is not evaluated.

3. OUR APPROACH: ESTIMATING VULNERABILITY IN THE PRESENCE OF CFC

Vulnerability is a comprehensive metric for estimating the susceptibility of a the execution of a program on a processor. As opposed to fault injection based techniques, it does not depend on accurate distribution of fault injections – which is a very difficult thing to achieve. Vulnerability estimation systematically analyzes every $< bit, cycle >$, and declares it vulneable, if a fault in can cause the program output to go wrong.

CFC techniques are implemented as a set of control flow checks. A control flow check makes several $< bit, cycle >$ that were vulnerable before, not-vulnerable. For example, in figure 1, if there is a fault in PC during the fetch of the first instruction of B2, then it will be caught. Therefore the bits in the PC at that cycle are not vulnerable after CFCSS is implemented. In fact, the bits of PC of any instruction, which can become the address of an instruction in basic block B2 have now become non-vulnerable (because they will be caught by the CFC check). Plus bits in the branch offsets of the other jump instructions in the program that can become the address of an instruction in B2 through a 1-bit flip are also no longer vulnerable.

The objective of vulnerability modeling in the presence of CFC technique is to find the $< bit, cycle >$s that were vulnerable, but are no longer vulnerable after the implementation of CFC – essentially calculating the protection afforded by the CFC technique. We do this by breaking it into two steps: i) for each vulnerable $< bit, cycle >$, find out which control flow error it causes, and ii) find out if that control flow error can be caught by the CFC that we are evaluating. The first step is relatively CFC independent, and captures the impact of soft errors in architectural bits on the control flow of the program, while the second step is relatively architecture independent and captures the capabilities of the CFC technique. We first start with the second step.

3.1 Which control flow errors can the CFC catch?

Although CFC techniques attempt to detect control flow errors, most CFC schemes can only detect a subset of control flow errors. A control flow error is identified 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, if the execution flows from a basic block to a basic block that is not a legitimate successor, then CFCSS can detect it, but it cannot detect when the execution jumps to an instruction in the same basic block.

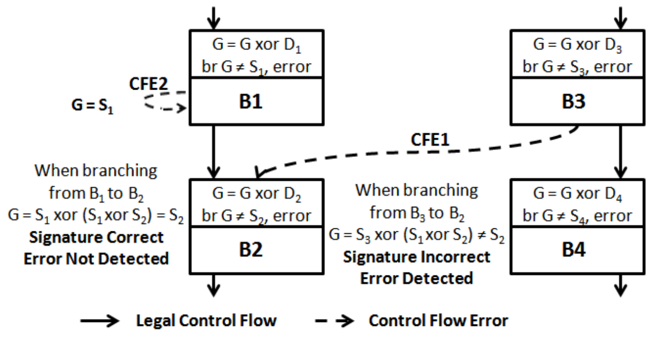
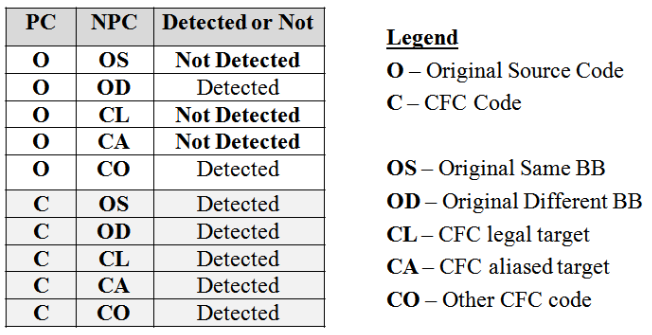


Figure 2: In CFCSS[14], checking-instructions are added to each basic block to set a variable to the basic block signature; and to check if the execution is coming from a legitimate parent basic block. When execution flows to B2 from B1 (a legal parent of B2) the signature checks out, but if the execution flows from B3 (not a legal parent of B2), a control flow error happens.

To comprehensively model the protection achieved by a CFC technique implementation, we build a CFC Protection Table – a table of all the categories of $pc \rightarrow npc$ transitions, that may lead to control flow errors in the system. The CFC Protection Table for the CFCSS protection technique is shown in figure 3. 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 occurs represented by the $pc \rightarrow npc$ transition, where pc is at the location specified in the first column, and npc is at the location specified in the second column, will be detected by the CFC or not. For protection modeling for CFCSS, we label the code in the program with a software based CFC implementation as: "*original source code*" – that was part of original the unprotected program; and "*CFC code*" – that is part of the additional code inserted during the CFC implementation (respective sections of a basic block are labeled in figure 2). The first row in figure 3 shows that CFCSS cannot detect if the control flow error causes a $pc \rightarrow npc$ transition from an instruction in original source code of a basic block (O), to the original source code of the same basic block (OS). On the other hand, the second row shows that if the control flow error causes an erroneous $pc \rightarrow npc$ transition from an instruction in 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. This is because, such a control flow error will essentially bypass the signature update in the other



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

Legend

O – Original Source Code
C – CFC Code

OS – Original Same BB
OD – Original Different BB
CL – CFC legal target
CA – CFC aliased target
CO – Other CFC code

Figure 3: 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 of CFCSS.

basic block, and then the signature will not match in the next basic block. CL refers to the checking code of a legal target of the basic block, and CA refers to the checking code of an aliased target of the basic block.

3.2 What control flow errors are caused by a fault in <bit,cycle>?

In general, modeling the effects of a fault, i.e., which control flow errors will a flip in $< bit, cycle >$ will cause is hard. This is not only because the number of bits (and cycles) is very large, but also because to find the effect of each fault, we essentially have to follow the dependencies of the fault (analyze error propagation), and and see if it can affect the sequence of instructions executed. Fortunately, our analysis is simplified by the observation that a fault in a $< bit, cycle >$ can be of two types: i) not-successor control flow error, and ii) wrong-successor control flow error, and existing CFC techniques cannot detect wrong-successor control flow errors; they can only detect not-successor control flow errors.

We perform a component-wise analysis. For each $< bit, cycle >$ tuple in each component, we find out, which control flow error (in the last section) will it cause, and then we see, if the control flow error is can be detected by the CFC technique or not. For lack of space, we will discuss the fault analysis on PC for a 5-stage in-order DLX pipeline [?], but our approach and analysis is not restricted to the architecture. In fact, our experiments are on the ARM v7-a architecture.

A bit flip in PC can cause an erroneous value to be written into the PC (updated next-PC); if the select bit for the multiplexer (MUX) that chooses between PC+4, and branch target indicates a non-branch instruction. For instance, let us assume a non-branch instruction in the current cycle. The bit flip in the PC causes it 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 values of pc, and then use those $pc \rightarrow npc$ transition to find if the CFC technique will be able to catch the erroneous transition or not. If the MUX select bit indicates a branch instruction, the erroneous PC values are overwritten in the next cycle with values from the branch target address field in the execute-memory pipeline register. Since the erroneous PC value is not used to fetch instructions, the PC bits in the current cycle can be considered not vulnerable.

In summary, most bit flips in the PC will most probably cause a not-successor control flow error. However, some bit flips in the PC can cause a wrong-successor control flow error. Since not-successor control flow errors are typically caught by existing CFC schemes, we conclude that PC is relatively well protected by CFC schemes. However a bit flip in the register file will in most cases result in a wrong-successor control flow error by changing a branch outcome. As a result, it will be detected by existing CFCs. There are exceptions though – for example if the program jumps to an address specified in a register. Then a bit-flip will most probably cause a not-successor control flow error. As a result, most bit flips in the register file will not be caught by CFC schemes. Similar is the story with other components like caches, load store queues etc. We perform detailed analysis of the impact of bit-flip in each bit of the microarchitectural component. It is worth mentioning that bit flips in several bits of the pipeline registers, e.g., the branch target address can cause not-successor control flow errors.

4. EXPERIMENTS AND ANALYSIS

Simulation Environment	
Architecture	ARM v7-a
Pipeline	5-stages (Out-Of-Order)
L1 D-Cache L1 I-Cache	64KB (2-way) 32KB (2-way)
D-TLB / I-TLB	64 entries
Physical Reg (INT/FP) Architecture Reg (INT/FP)	128/128 16/32

Table 1: Experimental setup for the case-study application of the gemV+llVm framework.

4.1 Experimental Setup

We perform our experiments and quantitatively analyze the effectiveness of CFC techniques using our gemV+llVm framework configured for the ARM v7-a architecture on MiBench benchmarks [6] (more details in table 1). We have implemented various state-of-the-art software CFC techniques, i.e., CFCSS, CFCSS+NA, and CEDA, and the software part of the hybrid technique CFEDC in the llVm compiler. The llVm compiler also generates the information about the location of CFC instructions within each basic block, and the location of the original source code, as shown in figure 2. This information is needed by the gemV simulator to model the protection achieved by the CFC techniques. For each CFC technique, we construct the CFE Protection Table (like that in figure 3 for CFCSS), for each of the implemented CFC techniques, and provide it as input to the gemV+llVm framework.

4.2 Vulnerability increases after software CFC!

Figure 4 plots the vulnerability of benchmarks when CFC is applied, normalized to their vulnerability without CFC. For this plot, the data cache is ECC protected. The plot shows this vulnerability ratio for 5 CFC protection schemes, CFCSS, CFCSS+NA, and CEDA (software techniques), CFCET (hardware technique), and CFEDC (hybrid technique). The last set of bars show the vulnerability ratio averaged over all the benchmarks. The most important observation to be made from this plot is that the vulnerability does not reduce after applying

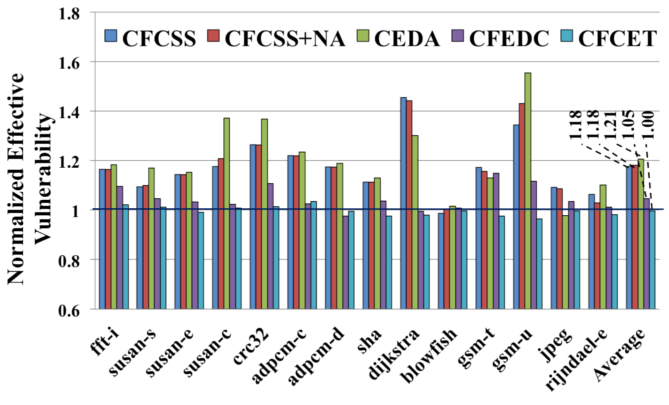


Figure 4: The effective system vulnerability (normalized over original program vulnerability) of the various CFC techniques, for a ECC protected L1 data cache.

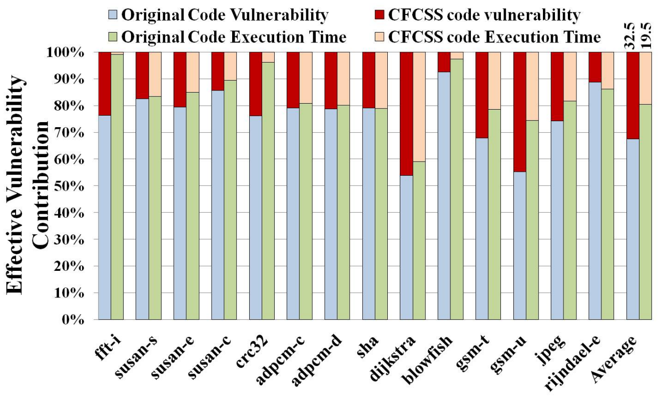


Figure 5: The proportion of vulnerability and execution time contributed by the added CFC code and original program code in the CFCSS implementation is represented for the benchmarks.

the CFC protection. Software based CFC protection (CFCSS, CFCSS+NA, and CEDA) increases system vulnerability for most benchmarks (13 out of 14), and on an average increase it by 18%-21%. For the *gsm-u*, the application of CEDA increases the vulnerability of the benchmarks by more than 50%. Hybrid CFC technique, CFEDC, also results in an increase in the vulnerability of most of the benchmarks, and on an average results in 5% increase of system vulnerability. Although the vulnerability after applying the hardware scheme CFCET does not increase, note that hardware techniques will have additional design, area, power and cost overheads.

Figure 5 plots the overhead in vulnerability and runtime added by the CFCSS code inserted by the compiler. If the vulnerability of the application after the application of CFCSS is 1, then the dark portion of the left bar for each application shows the vulnerability of the extra code added by CFCSS, while the light portion of the bar shows the vulnerability of the original code after the application of CFCSS (this is less than the vulnerability of the original code before the application of CFCSS). We can see that CFCSS code has a quite significant vulnerability (on avaerage 32.5%). The right bar is the performance bar, and also has light and dark portions. If the runtime of the application after applying CFCSS is 1, then the dark portion is the fraction of runtime spent in executing CFCSS code, while the light portion is fraction of runtime spent in executing the original code (which is approximately the same as the runtime of the original code before the application of CFCSS). The reason we plot both the bars together is to show the correlation in the increase in vulnerability and the increase in the runtime. We can see that in the cases where the CFC code contributes a larger fraction of the system vulnerability, it also contributes a larger fraction towards the runtime. The correlation coefficient between the vulnerability increase and runtime increase is 0.75.

4.3 Why are CFCs ineffective?

The reason exisinting CFCs do not protect computation from soft errors is because they only protect from faults in some microarchitectural components, e.g., PC, some bits of some pipeline registers (ones before the branch address generation), and branch target buffer, the link register, and the processor status register. Faults in all other microarchitectural components, e.g., the data cache, all other pipeline register bits, register file, and all the buffers and queues in the processor are not protected by existing CFC schemes. This is because faults in these components rarely cause "not-successor control flow errors." Most often they cause "wrong-successor control flow errors." For example, a fault in the data cache will cause a control flow error if the data value affects the outcome of the branch, and that results in a "wrong successor control flow error", and not a "not successor control flow error."

The reason why vulnerability actually increases is because the extra code added to implement CFC adds to the execution time, this in turn increases the time variables spend in microarchitectural components, and that increases the system vulnerability. In fact that increase more than outshines any reduction achieved by CFC.

To evaluate the potential of protection possible through a classicial CFC scheme, we assume a hypothetical CFC scheme that can detect all the illegitimate control flow errors. Even such a scheme can reduce the vulnerability of the execution by only 4.04% – assuming no extra instructions to achieve such a scheme.

We have modeled and experimented upon several CFC schemes, and observed that they do not reduce the vulnerability. However, there are more CFC schemes that we have seen but not implemented and tested. We believe that all of those schemes will also result in an increase in the vulnerability – and are therefore not useful. This is because existing CFC techniques only compete with each other to detect more kinds and larger fractions of "not-successor control flow errors.". For example, CFCSS cannot detect control flow erros when there is an aliased block. CFCSS+NA fixes this problem. These two did not detect the error in the conditional execution. CEDA fixes that loophole. The problem is that, as the techniques try to cover more and more cases, they add additional instructions – which backfires, and actually increases the vulnerability even more. In fact, we observe that the more sophisticated the technique is, higher is the increase in vulnerability.

Finally, there are some schemes (YACCA[5] and CEDA[18]), that attempt to detect errors in branch direction. They use additional instructions to re-evaluate the branch condition; but such schemes can only detect faults that (happened in the branch operands) between the two computations. Therefore they cannot have a significant impact.

5. SUMMARY

This paper has a very surprising conclusion. We show that existing CFC techniques (that have been developed to protect programs from soft errors), actually make matters worse âĂŞ they make programs more susceptible to soft errors. We show this by estimating the vulnerability of the program "without CFC", and "with CFC". We find that the vulnerability of the program "with CFC" is higher than "without CFC" for software and hybrid CFC schemes. This is because the small reduction in program vulnerability achieved by CFC is overwhelmed by the additional vulnerability due to the extra instructions that implement the CFC. Also the reduction in CFC is quite small, because the total number of bits even partially protected by CFC are a small fraction of the total number of bits in the processor.

6. ACKNOWLEDGEMENTS

This work was partially supported by NSF grant CCF 1055094 (CAREER).

7. REFERENCES

[1] Alkhalifa, Z., Nair, V. S. S., Krishnamurthy, N., and Abraham, J. A. 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.

[2] Chao, W., Zhongchuan, F., Hongsong, C., Wei, B., Bin, L., Lin, C., Zexu, Z., Yuying, W., and Gang, C. CFCSS without Aliasing for SPARC Architecture. In *International Conference on Computer and Information Technology (CIT)* (29 2010-july 1 2010), pp. 2094 –2100.

[3] Eifert, J., and Shen, J. Processor Monitoring Using Asynchronous Signatured Instruction Streams. In *Twenty-Fifth International Symposium onFault-Tolerant Computing* (jun 1995), p. 106.

[4] Farazmand, N., Fazeli, M., and Miremadi, S. FEDC: Control Flow Error Detection and Correction for Embedded Systems without Program Interruption. In *Third International Conference on Availability, Reliability and Security* (march 2008), pp. 33 –38.

[5] Goloubeva, O., Rebaudengo, M., Sonza Reorda, M., and Violante, M. Soft-error detection using control flow assertions. In *IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems* (nov. 2003), pp. 581 – 588.

[6] Guthaus, M. R., Ringenberg, J. S., Ernst, D., Austin, T. M., Mudge, T., and Brown, R. B. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the IEEE International Workshop on Workload Characterization* (Washington, DC, USA, 2001), IEEE Computer Society, pp. 3–14.

[7] Kayali, S. Reliability Considerations for Advanced Microelectronics. In *Proceedings of the 2000 Pacific Rim International Symposium on Dependable Computing* (Washington, DC, USA, 2000), PRDC '00, IEEE Computer Society, p. 99.

[8] Madeira, H., and Silva, J. On-line signature learning and checking: experimental evaluation. In *Proceedings of 5th Annual European Computer Conference* (may 1991), pp. 642 –646.

[9] May, T. Soft Errors in VLSI: Present and Future. *IEEE Transactions on Components, Hybrids, and Manufacturing Technology 2*, 4 (dec 1979), 377 – 387.

[10] Michel, T., Leveugle, R., and Saucier, G. A New Approach to Control Flow Checking without Program Modification. In *Twenty-First International Symposium on Fault-Tolerant Computing* (jun 1991), pp. 334 –341.

[11] Mukherjee, S. *Architecture Design for Soft Errors.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[12] Mukherjee, S. S., Weaver, C., Emer, J., Reinhardt, S. K., and Austin, T. A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor. *IEEE/ACM International Symposium on Microarchitecture 0* (2003), 29.

[13] Nicolescu, B., Savaria, Y., and Velazco, R. Software detection mechanisms providing full coverage against single bit-flip faults. *IEEE Transactions on Nuclear Science 51*, 6 (dec. 2004), 3510 – 3518.

[14] Oh, N., Shirvani, P., and McCluskey, E. Control-flow checking by software signatures. *IEEE Transactions on Reliability 51*, 1 (mar 2002), 111 –122.

[15] Rajabzadeh, A., and Miremadi, S. CFCET: A Hardware-based Control Flow Checking Technique in COTS Processors using Execution Tracing. *Microelectronics Reliability 46*, 5 (2006), 959–972.

[16] Saxena, N. R., and McCluskey, W. K. Control-Flow Checking Using Watchdog Assists and Extended-Precision Checksums. *IEEE Transactions on Computing 39*, 4 (Apr. 1990), 554–559.

[17] Schuette, M. A., and Shen, J. P. Processor Control Flow Monitoring Using Signatured Instruction Streams. *IEEE Trans. Comput. 36*, 3 (Mar. 1987), 264–276.

[18] Vemu, R., and Abraham, J. CEDA: Control-Flow Error Detection Using Assertions. *IEEE Transactions on Computers 60*, 9 (Sept. 2011), 1233–1245.

[19] Vemu, R., Gurumurthy, S., and Abraham, J. ACCE: Automatic correction of control-flow errors. In *Test Conference, 2007. ITC 2007. IEEE International* (oct. 2007), pp. 1 –10.

[20] Venkatasubramanian, R., Hayes, J., and Murray, B. Low-cost on-line fault detection using control flow assertions. In *On-Line Testing Symposium, 2003. IOLTS 2003. 9th IEEE* (july 2003), pp. 137 – 143.

[21] Wilken, K., and Shen, J. P. Continuous Signature Monitoring: Efficient Concurrent-Detection of Processor Control Errors. In *Proceedings of the 1988 International Conference on Test* (Washington, DC, USA, 1988), ITC'88, IEEE Computer Society, pp. 914–925.

[22] Wu, Y., Gu, G., Huang, S., and Ni, J. Control Flow Checking Algorithm using Soft-based Intra-/Inter-block Assigned-Signature. In *Computer and Computational Sciences, 2007. IMSCCS 2007. Second International Multi-Symposiums on* (Aug.), pp. 412–415.