Towards Low Overhead Control Flow Checking Using Regular Structured Control

Zhiqi Zhu Joseph Callenes-Sloan
Department of Electrical Engineering
The University of Texas at Dallas
Email: {zxz131630, jcallenes.sloan}@utdallas.edu

***Abstract*—With process scaling and the adoption of post-CMOS technologies, reliability has been brought to the forefront of modern computer system design. Among the different ways that hardware faults can manifest in a system, errors related to the control flow of a program tend to be the most difficult to handle when ensuring reliable computing. Errors in the sequencing of instructions executed are usually catastrophic, resulting in system hangs, crashes, and/or corrupted data. For this reason, conventional approaches rely on some form of general redundancy for detecting or recovering from a control flow error. Due to the power constraints of emerging systems however, these types of conservative approaches are quickly becoming infeasible. Control Flow Checking by Software Signatures (*CFCSS*) is a software-based technique for detecting control flow errors [1] that using assigned signatures rather than by using general redundancy. Unfortunately, the performance overhead for *CFCSS* can still be as high as 80%–90% for many applications. In this paper, we propose a novel method for reducing the overhead of control flow checking by exploiting the regular control structure found in many applications. Specifically, we observe that the alternating sequence of conditional and unconditional based control allows for the full control signatures to be computed at alternating basic blocks. Based on experimental results of the proposed approach, we observe that the overheads of the traditional methods are reduced on average by 25.9%.**

***Index Terms*— transient faults, error detection, control flow checking, software signatures, LLVM**

I. Introduction

In current computing systems, the handling of hardware faults has quickly become a first order design consideration. Moreover, with process scaling and the adoption of post-CMOS technologies, the occurrence of hardware faults is expected to increase exponentially in the next decade [2]. Hardware faults, including transient types of faults, are detrimental for building future low power and high performance machines. Transient hardware faults can manifest as different types of errors in the system. One common occurrence is the control flow error in the program (i.e. errors in the sequencing of instructions executed). This type of errors tends to be the most difficult to efficiently tolerate, as control errors in a system typically result in catastrophic events, such as crashes, hangs, or memory corruptions [3]. Therefore, detection of control errors is a major challenge for the design of reliable and efficient computing systems.

The traditional approach for error detection is based on redundancy [4], which can be based on either spatial or temporal redundancy. However, due to the power constrains and performance overheads, duplication quickly become infeasible for emerging highly power constrained systems. Control Flow Checking by Software Signature (*CFCSS*) [5] is a software-based technique developed specifically for detecting errors in the control flow of a program. Unfortunately, *CFCSS* also requires intensive instrumentation of the target code, and additional instructions for checking can incur costly runtime overheads, as high as 80% to 90%.

In this paper, we propose to reduce the overhead of providing control flow resilience by exploiting the regular control patterns found in many applications. Applications are commonly implemented with different types of loops and nested loops, that exhibit recurrent or regular control flow patterns. We observe that one such control flow pattern is the fact that many programs have alternating sequences of basic blocks terminating with conditional and unconditional branches. By leveraging this inherent structure in the control flow, we can calculate and track updates to the control signatures less frequently, so the runtime overhead of control flow checking can be significantly reduced without sacrifice of fault coverage. This paper makes the following contributions:

- We investigate techniques for reducing the runtime overhead of control flow checking by leveraging the regular control structure in many programs, eliminating the need to frequently update control signatures.
- We show an example implementation of regularized control flow checking algorithm (*RCFC*) by the *LLVM* compiler infrastructure.
- We perform an experimental study of a set of multimedia and scientific benchmarks to prove the efficiency of our proposed regularized control flow checking approach.

The rest of the paper is organized as follows. Background and related work are discussed in Section II. We introduce the proposed approach for reducing the overhead of control flow checking in Section III. Next, we explain the methodology and simulation results in Section IV and V. Section VI concludes.

II. Background and related work

CFCSS Example Control flow checking based on software signature (*CFCSS*) is the most representative method [5] for transient control error detection. The algorithm of *CFCSS* could be summarized by following steps, and example details are shown in the Fig. 1.

1. Assign a static signature s for each basic block and calculate the signature difference d compared to predecessor.
2. At run time, calculate dynamic signature G.
3. For *fan-in* node, the dynamic signature should be calibrated by adjusting signature D: $G = G \bigoplus D$.
4. Check the equivalence between dynamic signature G and static signature s. If $G = s$, no error; otherwise the program will jump to error recovery routines.

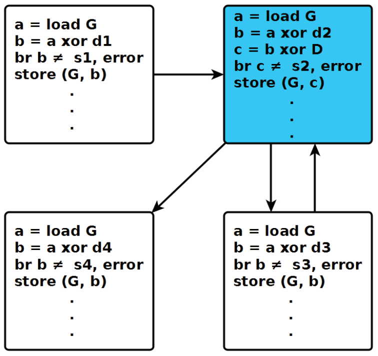


Fig. 1: Traditional *CFCSS*. White box is the basic block terminated with the *unconditional branch*, and blue one is with *conditional branch*.

By using *CFCSS*, enormous amount of instructions should be inserted dynamically for error detection. The prohibition for practical implementation of *CFCSS* is huge performance overheads caused by the complicated signature calculation.

III. Proposed Approach

In this paper we will propose a novel method which could detect the control flow errors efficiently, and making software-based approach much more practical for real applications. Our proposed algorithm will simplify the signature checking by taking advantage of regular control flow structures in the most computing workload. For convenience, in the later sections we will refer to the basic blocks with unconditional *branch* as ***unconditional blocks***, and those have conditional *branch* as ***conditional blocks***.

A. Regular Control Flow Structure

By study of the scientific applications, we know the most computing rely on the *for-loop* or *while-loop* operation. Through observation of the CFG, the *loop* operations have the similar control flow structure as shown in Fig. 2, which have the following special characteristics:

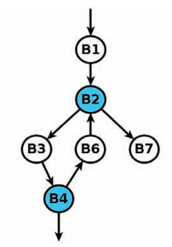


Fig. 2: Control flow graph with regular structure.

1. The *conditional* and *unconditional* blocks are executed in the alternating sequence.
2. Two sequential executed *unconditional* blocks are always connected by an identical *conditional* block for different dynamic instances at run time.

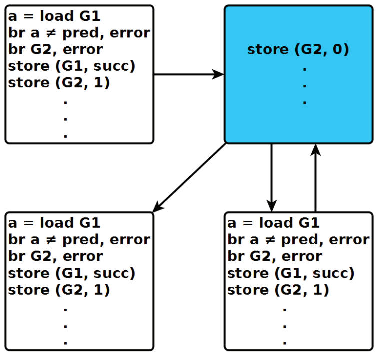
These regular control flow structures allow us to implement checking only at the unconditional blocks instead of each block which traditional *CFCSS* applied. Specifically, if an unconditional block has been performed without control flow error, its predecessor should be exactly same as the successor of last executed unconditional block. For example, the successor of *B1* is *B2* in Fig. 2. If we assume no control flow error, the predecessor of next unconditional block should be *B2* (e.g. *pred*(*B3*/*B7*) = *succ*(*B1*) = *B2*). With the help of this pervasive alternating execution pattern, we could reduce the performance overheads due to the efficient and simplified checking algorithm.

B. Algorithm

For our approach, *Regularized Control Flow Checking (RCFC)*,we could supervise the dynamic control flow behaviors by specific characteristics which discussed above. First, we assign a unique signature for each basic block statically. Then, at run time, detection mechanism will be implemented to check that the predecessor of executing unconditional block is conditional and same as the successor of last executed unconditional one. Here, we use global register *G1* to keep track of the successor signature, and label block types by *G2* (e.g. *unconditional* = *1* and *conditional* = *0*). For typical transitions between *BB*s which shown in Fig. 2, the details of our algorithm are shown as follow.

1. **B1**: ***errs1*** = *Boolean (G1* $\neq$ *B1.pred)*
   errs2 = *G2*
   G1 = *B1.succ*
   G2 = *1*
2. **B2**: *G2* = *0*
3. *If B3 on the execution path, we will check if its predecessor is both conditional and also equal to G1.*
   B3: ***errs1*** = *Boolean (G1* $\neq$ *B3.pred)*
   errs2 = *G2*
   G1 = *B3.succ*
   G2 = *1*

RCFC Example Continuing the example from Section II, we will now describe how *RCFC* can be used for the same piece of example code (see Fig. 3). We use the global register ***G1*** to store the successor signature of executing unconditional block. When program begins to execute a new unconditional block, we could check if its predecessor match to the signature saved in ***G1***; With global register ***G2***, we could inspect the alternating sequence of conditional and unconditional blocks at run time. At the beginning of each unconditional blocks, this checking algorithm could be implemented by inserting *branch* instructions to ensure program control will divert to the error recovery routine if errors happened.



```
a = load G1
br a ≠ pred, error
br G2, error
store (G1, succ)
store (G2, 1)
```

```
store (G2, 0)
```

```
a = load G1
br a ≠ pred, error
br G2, error
store (G1, succ)
store (G2, 1)
```

```
a = load G1
br a ≠ pred, error
br G2, error
store (G1, succ)
store (G2, 1)
```

Fig. 3: Regularized Control Flow Checking

Based on the above examples, we can observe the amount of additional inserted check instructions for each approach. For the traditional *CFCSS* in Section II, we inserted 9 instructions in total at each pair of conditional and unconditional blocks. In contrast, by proposed *RCFC*, we need just 6 instructions for protecting the same control flow.

C. *Preprocess at compile time*

From above discussion, we know that the computing based on the intensive *loop* operations should have the regular control flow structures. However, we know there are still some rare special cases of control flow that do not follow common structures in practical applications. For example, two blocks with the same type (e.g. conditional/unconditional) are connected together. So we need to do some pre-processing for original program before applying our checking algorithm. At the compile phase, if multiple unconditional blocks connected statically, they should be combined together into a single block. This work could be achieved by compiler build-in optimization pass. Further transformation of the optimized code should be also done in this phase if control flows are still in special case: insert an unconditional block between two conjoint conditional blocks. These two steps generalize the original program to have the common regular structure, and make sure our algorithm could be applied to most applications.

IV. Methodology

A. *Benchmarks*

We investigate the performance overheads of the techniques for a set of multimedia and scientific applications from the *StreamIt* benchmark set [14]. These applications have been widely used in high performance computing systems, especially in the context of audio, image, and signal processing.

B. *Compiler Infrastructure*

In this work, we use *LLVM* compiler [13] to implement both the traditional *CFCSS* and our algorithm. First, we could compile the application source code to *LLVM's* intermediate representation called *LLVM-IR*. Then we will propose the *LLVM* transformation pass that inserts checking instructions to implement error detection according to the different algorithms. Finally, we will measure the performance overheads. Because both algorithms insert the same type of instructions (e.g. *br*, *xor*, *store*, and *load*), we could simply count the number of *LLVM-IR* instructions to get overhead results no matter what kinds of *ISA* will be used for different architectures.

V. Results

The fraction of the application which contains the inherent pattern is an important factor in determining the overhead of the approach. In this Section we first observe the percentage of regular control flow structures found in the different applications. For each column in Table I, we show the number of *BB*s in the original program, amount of inserted *BB*s at pre-processing phase, and the percentage of each application that contains the regular control flow structures. By the calculation results, we know that a large fraction of the applications (over 90%) contains regular control flow structure.

TABLE I: Percentage of regular control flow structures

Application	Original_BB	inserted_BB	Common factor
matrixmult	10172	207	98.01%
matmul-block	7636	306	96.15%
audiobeam	9055660	1565569	85.26%
bitonic-sort	2354	294	88.89%
fft	5509	659	89.31%
dct	5027	401	92.62%
filterbank	2299138	249909	90.20%
fmref	34116	6963	83.05%

A. *Fault Coverage*

The theoretical detection analysis of *RCFC* algorithm will be given in this section, and it uses a similar methodology as previous control flow checking studies [1]. As we discussed above, the control flow errors can significantly impact the correctness of a program's execution by changing the branch targets. For example, the usual successors of *node2* are *node3* and *node7* in Fig. 4. However, due to the hardware transient error, a bit flip in the target addresses of *branch* instruction at *node2* will lead to an unpredictable jump to *node4*. This illegal branch behavior will be likely to damage the accuracy of computing, and produce erroneous results.

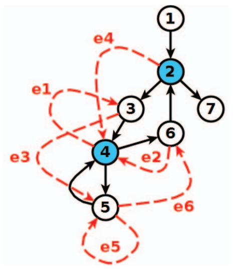


Fig. 4: The black solid lines are valid jumps between blocks, and the red dash lines show us the illegal branch behaviors.

The *RCFC* detection scheme of illegal control flow behaviors in Fig. 4 are summarized by TABLE II. ***G**** stores the value that will be used in the checking instructions, and ***G*** updates the value before exiting from the current node. For example, the erroneous control flow behavior from *node4* to *node3* (i.e.error **e1**) could be detected because *err1 = Boolean (G1* ≠ node3.pred) = Boolean (node4 ≠ node2) = true*; for another instance, when the error **e3** occurs, we could detect it by *err2 = G2* = true*.

TABLE II: Analysis of error detection

	Node	G1*	err1	G2*	err2	G1	G2
e1	n3	4	**1**	0	0	4	1
e2	n4	4	-	1	-	4	0
	n6	4	**1**	0	0	2	1
e3	n5	4	0	1	**1**	4	1
e4	n4	2	-	0	-	2	0
	n5/n6	2	**1**	0	0	4/2	1
e5	n5	4	0	1	**1**	4	1
e6	n6	4	0	1	**1**	2	1

B. *Performance Overhead*

In this part, we will compare the performance overheads between *CFCSS* and our algorithm. For each application in the *StreamIt* benchmark,we could calculate the overheads by

$$Overhead = \frac{Insts.\_Checking - Insts}{Insts}$$

Where *Insts._Checking* is number of dynamic instructions executed after applying the checking algorithm, and *Insts* is instructions in the original application. The overhead comparison between traditional *CFCSS* and proposed algorithm is given in Fig. 5. From this experimental result, we know that the overhead of control flow checking is reduced dramatically in the common cases. The average reduction is 25.9%, and 36% approximately in the best cast (e.g. for *matrixmult*).

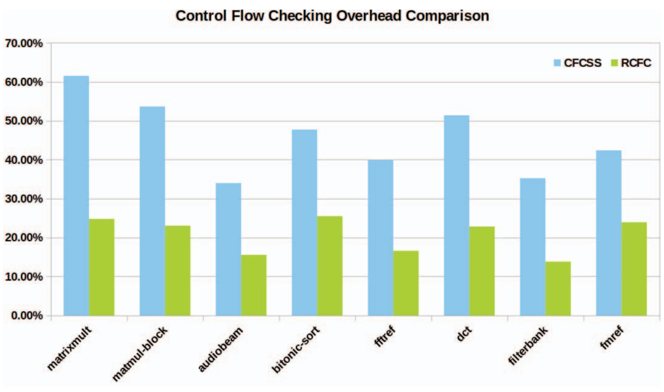


Fig. 5: Overhead comparison between the traditional *CFCSS* and our proposed *RCFC* approach.

C. *Discussion*

We observe that the overheads reduction strength varies among different applications, and there are two notable reasons for that diversity. The first one is about program characteristics: some applications are control-intensive so control flow transfers are widespread, in such case the overheads reduction is obvious; the other important reason is due to the structure of application. For example, most part of *matrixmult* have the common regular control flow structure, so it needs less work on generalization at preprocessing phase, therefore it avoids the additional overheads and harvests extra benefits.

VI. Conclusion

Techniques for detecting and recovering from transient errors will be the primary consideration in future reliable system design. There are several algorithms have been proposed for detecting the run-time transient control flow errors, however, power constrains and large performance overheads become a significant obstacle. In this paper we proposed a novel *Regularized Control Flow Checking* approach that could detect control flow errors at execution time with significantly less runtime overhead by leveraging the inherent regular control flow found in the majority of applications. We implemented our algorithm using *LLVM* compiler, and provided experimental results for typical multimedia and scientific applications. Future work involves exploring other types of regular characteristics found in application's control flow, so it will further improve the efficiency of control flow resilience techniques.

References

[1] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Control-flow checking by software signatures," *Reliability, IEEE Transactions on*, vol. 51, no. 1, pp. 111–122, 2002.

[2] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi, "Modeling the effect of technology trends on the soft error rate of combinational logic," in *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*. IEEE, 2002, pp. 389–398.

[3] S. Feng, S. Gupta, A. Ansari, and S. Mahlke, "Shoestring: probabilistic soft error reliability on the cheap," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 1. ACM, 2010, pp. 385–396.

[4] S. Harada, M. Yoshimura, and Y. Matsunaga, "Tmr based error correction method considering trade-off between area and soft-error tolerance," in *Proc. 19th International Workshop on Logic and Synthesis*, 2010, pp. 69–75.

[5] A. Shrivastava, A. Rhisheekesan, R. Jeyapaul, and C.-J. Wu, "Quantitative analysis of control flow checking mechanisms for soft errors," in *Design Automation Conference (DAC), 2014 51st ACM/EDAC/IEEE*. IEEE, 2014, pp. 1–6.

[6] R. Vemu, J. Abraham *et al.*, "Ceda: Control-flow error detection through assertions," in *On-Line Testing Symposium, 2006. IOLTS 2006. 12th IEEE International*. IEEE, 2006, pp. 6–pp.

[7] F. Perry and D. Walker, "Reasoning about control flow in the presence of transient faults," in *Static Analysis*. Springer, 2008, pp. 332–346.

[8] D. S. Khudia and S. Mahlke, "Low cost control flow protection using abstract control signatures," in *ACM SIGPLAN Notices*, vol. 48, no. 5. ACM, 2013, pp. 3–12.

[9] J. Wei and K. Pattabiraman, "Blockwatch: Leveraging similarity in parallel programs for error detection," in *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*. IEEE, 2012, pp. 1–12.

[10] S. K. Sahoo, J. Criswell, C. Geigle, and V. Adve, "Using likely invariants for automated software fault localization," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 1, pp. 139–152, 2013.

[11] A. S. P. Panchekha, T. M. K. S. McKinley, and D. G. L. Ceze, "Expressing and verifying probabilistic assertions," 2014.

[12] R. Venkatasubramanian, J. P. Hayes, and B. T. Murray, "Low-cost on-line fault detection using control flow assertions," in *On-Line Testing Symposium, 2003. IOLTS 2003. 9th IEEE*. IEEE, 2003, pp. 137–143.

[13] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*. IEEE, 2004, pp. 75–86.

[14] W. Thies, M. Karczmarek, and S. Amarasinghe, "Streamit: A language for streaming applications," in *Compiler Construction*. Springer, 2002, pp. 179–196.