Control Flow Checking Optimization Based On Regular Patterns Analysis

Zhiqi Zhu\(^1\), Joseph Callenes-Sloan\(^2\), and Benjamin Carrion Schafer\(^1\)

\(^1\)Department of Electrical & Computer Engineering, The University of Texas at Dallas, USA
\(^2\)Electrical Engineering Department, California Polytechnic State University, USA
Email: zhiqi.zhu, schaferb@utdallas.edu, jcallene@calpoly.edu

Abstract—With the continuous sub-micron process scaling, reliability of integrated circuits has quickly become a first-order design concern. In modern computing systems, transient errors are increasingly likely to corrupt the computation by altering the control flow or sequencing of instructions, leading to catastrophic failures. Prior work on control flow checking provides good coverage but at a high cost. In this paper, by exploring regular control flow patterns found in most applications, we propose the optimization schemes for software signature control flow checking that could reduce the error detection overheads. Specifically, we leverage the fact that most applications have: (1) simple fan-in / fan-out control flow patterns, and (2) most of control flows can be predicted during the compilation stage through static branch prediction heuristics. By exploiting these opportunities, we propose two techniques to reduce the number of inserted codes at common paths and simplify control flow checking of irregular patterns with minimal overheads. Experimental results on a variety of applications demonstrate that our approaches could reduce checking overhead by almost 2.5x on average while leading to similar fault coverage compared to traditional control flow checking.

Index Terms—transient error detection, control flow checking, static branch prediction, software signature, faults injection

I. INTRODUCTION

As CMOS technology is moving further into the nano-meter regime, the continual shrinking of transistor dimension has led to higher performance and power efficiency. However, at the same time, lower operating voltages and higher operating frequencies make state-of-the-art circuit designs more sensitive to environmental conditions. Soft errors are a glitch in semiconductor devices induced by neutrons from cosmic rays and high energy particles that are emitted by radiative impurities. These transient faults cause the circuit to fail catastrophically. Soft errors rate (SER) is measured in units of Failure in Time (FIT), and it is expected to increase exponentially to about once-a-day in the future systems [1]. Thus, the mitigation of transient faults is becoming a critical concern for the computing community.

One of the most common faults during execution is in the form of control flow error. It tends to be critical for dependable systems because it always leads to execution hangs, crashes, or even termination without any anomaly indication or warnings (e.g. silent data corruption) [2]. Therefore, efficient control flow checking mechanisms are needed to protect against transient errors. Most previous works in terms of error masking are based on hardware redundancy. To verify the run-time behavior, control flow instructions are replicated on independent hardware channels and fault-free execution could be derived from their outputs comparison. Common approaches include DWC (duplication with compare) or TMR (triple-modular redundancy) [3]. TMR has the benefit of fully mask faults, and also it could be implemented in time domain by re-computing the result on the same hardware channel [4]. However, these detection schemes suffer from prohibitive high area or performance overheads.

Therefore, for emerging power constrained systems, software-based control flow checking techniques have become the attractive alternative solution for faults detection. These techniques are based on the methodology that control flow behaviors could be supervised by inserting signature checker at the basic block granularity during code compilation, and any illegal control flow transfer could be detected by signature disparity at run-time. Conventional control flow checking approaches provide good fault coverage with minimal detection latency, but they rely on complicated run-time signature calculation especially for complex control flow structures. It makes them costly with performance overheads as high as 70% approximately [5].

To address these issues, in this paper, we present two control flow checking software signature optimizations with different performance/coverage trade-offs. First, we proposed Common Signature Control Flow Checking (CSCFC) which could sacrifice minor coverage for performance improvement. This method exploits the fact that most of execution time is spent on traversing regular control flow patterns with simple fan-in / fan-out structures, and dynamic control flow target calculation could be avoided by applying a common signature on all legitimate successors of a basic block. The second approach is referred to as Hot Successor Control Flow Checking (HSCFC). In this case, we found that run-time control flow behaviors could be predicted by static branch prediction at compilation stage, so that signature calculation or calibration is only needed at rare basic blocks. Both methods make use of

\(1\) Department of Electrical & Computer Engineering, The University of Texas at Dallas, USA
\(2\)Electrical Engineering Department, California Polytechnic State University, USA

Email: zhiqi.zhu, schaferb@utdallas.edu, jcallene@calpoly.edu
static regular pattern analysis to optimize control flow signature instrumentation, this makes our proposed methods more efficient than previous works which with intensive signature calculations. In summary, in this work we make the following contributions:

- Propose a novel control flow checking optimization (CSCFC) by applying common successor signature at regular control flow patterns. This method trades-off minor fault coverage for performance.
- Present an additional method hot successor control flow checking (HSCFC) that could further extend fault coverage by leveraging static branch prediction heuristics. It generally simplifies the control flow checking, and the dedicated signature monitor only applied on less frequent execution path.
- Perform the fault injection to simulate the effect of transient control flow errors. Through extensive experimental results, our proposed methods are demonstrated as the effective control flow checking solutions when compared to conventional approaches.

The rest of paper is organized as follows. Background and related work are discussed in Section-II. The proposed algorithms of control flow checking are described in Section-III. Section-IV shows the experimental results that compare to the state-of-the-art approach. Finally, Section-V concludes.

II. BACKGROUND AND RELATED WORK

Transient hardware faults can adversely impact the execution flow in different ways. Fig. 1 shows an example of Control Data Flow Graph (CDFG) that basic block $BB$ is defined as a code section of a single-entry and single-exit, and edges between basic blocks represent the possible control flow transitions based on the execution data. Control flow disturbances could appear when branch condition data is corrupted by a bit-flip error, and it modifies the branch target as a legitimate but incorrect successor. General hardware redundancy approaches such as DMR [6] and ECC protection [7] can protect against this error at the cost of replication. Moreover, branch target address itself is also fault-sensitive and it could result in computation corruption due to illegal control flow transfer (e.g. $BB1 \rightarrow BB4$ or $BB2 \rightarrow BB3$) as shown in Fig.1. Such errors tend to represent a disproportionately large number of failures because they could always lead to the execution terminated without any error indication. For this reason, in this work, we mainly focus on detecting the errors related to faulty branch targets, rather than condition data corruption.

A. Software-Based Control Flow Checking

The basic idea behind software-based control flow checking is using software signatures to monitor the control behaviors at run-time. Control flow checking by software signature (CFCSS) [8] is the most representative method as it does not require additional hardware support [9] [10]. Fig. 2 shows an overview of classical software-based control flow checking as well as our two proposed methods which will be described in the next section. For traditional control flow checking with software signature (CFCSS), the high-level source code is compiled to assembly language or intermediate representation (IR), and this low-level code is then partitioned into basic block which composed of a list of sequential instructions. Then, based on program structures, three types of control flow information are extracted statically: (i) static basic block signatures, (ii) difference with predecessor and (iii) adjustment signature at $fan-in$ node. Finally, the signature instrumentation pass is applied to supervise the control transfers. CFCSS monitors run-time behaviors at the level of basic block, and it could be summarized as follows:

1. Compile source code and extract control flow information from assembly code.
2. Assign a unique static signature “$s$” to each basic block, and compute the predecessor difference “$d$” as well as adjust signature “$adj$”.
3. Insert instructions at each basic block to calculate dynamic signature base on the signature records and predecessor difference (e.g. $G \oplus d$). Calculation result should be also calibrated by adjusting signature “$adj$” if basic block is $fan-in$ node.
4. Compare signatures (e.g. static v.s. dynamic) to verify the control flow transfer between basic blocks.

CFCSS provides good fault coverage with fine-grained control flow checking, however, high performance overhead is also introduced by the complicated signature calculation at each basic block. According to the experimental results shown in Section-IV, CFCSS is costly and prohibitive when it applied to control intensive applications especially.

Many other software-only techniques have been proposed to protect control flow integrity (CFI) in the past decade [11]. Khudia and Mahlke developed abstract control signature (ACS) [5] to reduce the checking frequency by raising checking levels to the abstract region. Specifically, an interval is defined as a region that contains a set of basic blocks, and the number of the basic block inside of each interval...
is recorded by a region signature at compiling time. At run- 
time, a counter is incremented at each executing basic block, 
and control flow checking is then performed at the exit of 
each interval by checking the equivalence of the counter value 
with the region signature. Because ACS coarsens the checking 
granularity, its detection latency is sensitive to the size of 
interval (number of BBs). Therefore, it could be an infeasible 
solution for timing critical application (e.g. real-time system). 

More recently, the authors in [12] exploit the typical regular 
patterns found in streaming applications to optimize the 
checking mechanism. The proposed regularized control flow 
checking (RCFC) approach reduces the performance overhead 
by leveraging the alternating sequence of basic blocks that 
terminate with different types of branch instructions (i.e. 
conditional vs. unconditional). This method limits control 
flow tracking frequency by imposing signature comparison 
only at unconditional blocks. However, before applying the 
checking scheme, RCFC requires control flow regularization 
in order to ensure that the entire program follows the same 
regular format. This pre-processing step introduces additional 
overheads and makes RCFC impractical for application with 
irregular control flow structures intensively. Therefore, it is 
important to investigate a cost-effective method that is generic 
for any program with consideration of different control 
flow structures.

B. Static Control Prediction

Control flow transfers during execution rely on the termina-
tor instruction at each basic block. Thus, predicting the 
branch target provides a good insight of the runtime behavior 
of a program. Typical branch prediction is done by profiling 
the program, it has been evaluated as a highly accurate 
control flow predictors. However, profile-based prediction is 
time-consuming because program codes need to be executed 
multiple times to generate the useful trace of control flow 
information [13]. Therefore, most commercial compiler infras-
tructures provide static branch predictor [14] to estimate run-
time control flow behaviors, and it typically uses the following 
prediction heuristics:

1. The probabilities of all outgoing edges from a basic block 
   add up to 100%.
2. If a basic block is terminated by an unconditional branch, 
   then run-time control flow target is always its unique 
   successor.
3. If a conditional branch has a backward outgoing edge 
   (e.g. passes control to an instruction with previous ad-
   dress), then it should be related to “loop” structure.
4. The loops usually choose iteration rather than exit. Thus, 
   the successor node on the backward path is more likely 
   of being taken as branch target.

Static branch prediction exploits control flow information 
by investigating program structures, it shows the low miss 
rates when execution is deeply nested in iterative executions. 
For convenience, in the later sections, we will refer to the 
most likely successor with higher “taken” probabilities as hot 
successor, and the corresponding basic block as hot block.

III. PROPOSED APPROACHES

In this section, our proposed control flow checking opti-
mizations are discussed in details. As one of the cornerstones,
regular control flow structures found in most programs are investigated with typical examples. Moreover, proposed methods that take advantage of static control flow analysis are explained: common signature control flow checking (CSCFC) simplifies control flow checking by applying common successor signature on regular structures; hot successor control flow checking (HSCFC) leverages static branch prediction to optimize verification on critical path of execution. Fig. 2 shows an overview of these methods as well as highlights their differences compared to traditional CFCSS.

A. Regular Control Flow Structures

Many applications contain highly regular control flow patterns. For example, the use of “goto” statement is discouraged in modern languages even though it is still a reserved programming keyword. It’s generally avoided because it may form “spaghetti” code and make the program hard to read. In fact, many compiler optimizations are based on the assumption that the control flow follows a fairly regular pattern (i.e. reducible directed acyclic graph) that run-time behaviors are mostly predictable statically. For example, if a basic block is terminated with an unconditional branch instruction, then the control transfer target is always its single successor; for a conditional basic blocks that has two successors, its control flow behavior depends on run-time condition data, but the dynamic target is still confined to its legitimate successors. Fig. 3 highlights the typical examples of regular control flow structures that dominate execution.

if-then-else: This structure evaluates the logic expression at the divergent point and selects one execution path based on the branch condition evaluation. As shown in the control flow graph in Fig. 3 (a), the fan-out block BB0 has two successors BB1 and BB2, and these two blocks converge at fan-in BB3. The dynamic behavior of the branch instruction at BB0 is uncertain at compile time, however, any control flow target except BB1 or BB2 can be treated as an error.

loop: Loop is one important construct in most programs, and it is widely used to repeat the same operations until a termination condition is reached. The "while" loop example shown in Fig. 3 (b) consists of three basic blocks: the header block BB0 specifies the termination condition, iteration body block BB1 performs data manipulations, and BB2 defines the operation after loop exit. At run-time, BB1 is more likely to be considered a branch target because loop iteration will take most of the execution time.

switch-case: For the real applications, control flow is much more complicated because a basic block may involve n possible successors. In Fig. 3 (c), “switch” statement allows a variable to be tested for equality against a list of values: each value is called a case, and the variable being switched on is checked for each switch case. In this case, even though control flow behaviors depend on run-time data, the legitimate successors of divergent block BB0 could be still investigated during compile time.

B. Common Signature Control Flow Checking (CSCFC)

Because regular control structures with simple fan-out nodes are commonly found in many applications, control flow checking could be optimized by leveraging a common signature to record branch successors. We call this technique Common Signature Control Flow Checking (CSCFC). Algorithm-1 summarizes this method as the following steps:

Step 1 - Basic Block Splitting: Inter-procedural control is a common control flow structure in many programs, and it’s always been expressed as function calls or subroutines. Error handling of call / return is intricate due to the fact that callee function may be invoked at different sites of the program. However, a common solution is splitting “caller” into two basic blocks that one includes instructions before call statement and the other with instructions afterward [15]. Fig. 4 shows an example where the original BB_caller is separated by two sub-blocks. BB_entry is called by subBB1, and subBB2 is set as the successor node when subBB1 regains control from callee. Thus, our proposed method starts by splitting bb as needed (line 3).
Step2 - Common Signature Assignment: For the basic block that terminated with an unconditional branch instruction, dynamic control flow target is always its unique successor, hence, a BB signature is assigned for its successor statically (line 13). For fan-out basic blocks with switch or conditional branch instruction, we applied the same signature on all of successor candidates (line 11). With common signature assignment, successor update could be promoted ahead before branch instruction. In other words, successor signature record “C” from previous node (e.g. predecessor) could be used directly for control flow checking (line 20), therefore, signature calculation employed by traditional CFCSS method could be avoided.

Step3 - Signature Checking Insertion: The last step inserts control flow checking into each basic block and outputs the newly program with signatures checkers (line 24). However, there are still some rare complex control flows that may not follow regular patterns. For example, as nested “if-then” structures shown in Fig. 5, fan-out blocks BB0 and BB2 have a common control target. By applying CSCFC, BB1 and BB3 will share the same static signature with BB4. In this case, crossover-errors indicated by dashed lines might not be detected due to signature aliasing. Such an error could reduce faults coverage and it will be discussed in the next section.

C. Hot Successor Control Flow Checking (HSCFC)

In this section, we describe a second method that could optimize control flow checking by predicting the most likely execution path statically. This proposed technique keeps tracking hot successor on the critical paths (e.g. traverse frequently) to simplify control flow checking further. We call this approach as Hot Successor Control Flow Checking (HSCFC). According to the experimental results shown in Section-IV, HSCFC could provide more comprehensive fault coverage, with minor control flow checking costs on irregular structures. Algorithm-2 summarizes the proposed technique which is composed of 4 main steps.

Step1 - Basic Block Splitting: Similar to the previous method, HSCFC starts by splitting the basic block that contains a function-call inside.

Step2 - Static Signature Assignment: Instead of applying common signature at successors of fan-out node, HSCFC assign a unique static signature to each basic block (line 6).

Step3 - Hot Successor Signature: Base on static branch prediction, successor that defines loop iteration body is hot basic block of unconditional branch, because it always has
higher “taken” probability than its sibling (line 11). However, for “if-else” structure shown in Fig.3 (a), static predictor is less effective because \( BB1_{\text{prob}} = BB2_{\text{prob}} = 50\% \). In this case, we could assume that one of candidates is pseudo-hot successor. \( \text{HotBBList} \) is used to record hot successor of each basic block (line 12).

**Step4 - Checking Instruction Insertion:** First, we performed control flow structure analysis during compilation. Then, different control flow checkers are inserted at each basic block according to basic block classification. Basic block could be sorted as one of the following categories:

- **hot_bb**: if basic block is hot successor for all of its predecessors.
- **fan-in**: basic block has more than one predecessors.
- **fan-out**: basic block has multiple successor nodes.
- **sibling_bb**: basic blocks that share the same predecessor.
- **regular_bb**: basic block within simple fan-in / fan-out structure that satisfies that any successor of fan-out basic block is not fan-in node.

**Regular Checker:** Instead of applying a general checking solution, light-weight control flow checkings are inserted into regular basic blocks: signature comparison is applied immediately at hot_bb, and calibration at non-hot sibling_bb should be performed before signature checking (line 19 to 22). This optimization inserts fewer instructions on frequent traversing path and simplifies checking at regular basic blocks. As an example, different checking solutions are applied on the same regular control flow structures. Fig.6 compares HSCFC and CSCFC as well as the conventional control flow checking (CFCSS). It can be observed that our proposed approaches improve checking performance by inserting less number of instructions in comparison to CFCSS.

**Special Case Checker:** In order to improve faults coverage, additional signature checking could be applied on rare complex control flow structures. As the conjoint fan-in-out structures shown in Fig.7, we assume that fan-out nodes \( BB1 \) and \( BB2 \) share the common pseudo hot successor \( BB4 \) because it could be invoked by different predecessors (e.g. \( BB1 \to BB4 \) or \( BB2 \to BB4 \)). Thus, as similar as hot_bb, only hot signature checking is needed (line 25) at \( BB4 \). However, differ from regular case, non-hot signature \( N \) checking (line 31) is applied on \( BB3 \) and \( BB4 \) to differentiate between each other so that crossover errors mentioned previously in Fig. 5 could be detected.

Fig. 6: Control flow checking comparison for regular “loop” structure: (a) conventional method (CFCSS[8]) (b) Common Signature Control Flow Checking (CSCFC) (c)Hot Successor Control Flow Checking (HSCFC)

![Control flow checking comparison for regular “loop” structure](image_url)

---

Fig. 7: Compressed fan-in-out: solid lines represent hot paths and dashed lines connect to non-hot blocks.

![Compressed fan-in-out](image_url)

A more complicated case is related to fan-in node that with inconsistent hot block assessment from different predecessors, and this disparity makes control flow checking more intricate. Fig.8 shows such an example where \( BB3 \) is the post-dominator for different execution paths. Specifically, it could be either called as \( BB2 \)’s unique hot successor or invoked by non-hot path from \( BB4 \). With only checking hot successor signature...
Algorithm 2 *Hot Successor Control Flow Checking*

**Input:**

- HotBBDiff = \{hot₁, ..., hotₖ\}: hot basic block list
- SibDiff = \{d₁, ..., dₖ\}: sibling difference

**Output:**

- BBList_check - BB list with signature checking

1: /* STEP1 - split caller bb */
2: SplitCallerBB();
3:  
4: /* STEP2 - static BB signature */
5: for all bbᵢ ∈ BBList do
6:   bbᵢ_sig = sᵢ
7: end for
8:  
9: /* STEP3 - branch prediction */
10: for all bbᵢ do
11:   if (succ₁_prob() ≥ succ₂_prob()) then
12:     HotBBList ← succ₁
13:   end if
14: end for
15:  
16: /* STEP4 - check signatures */
17: for all bbᵢ do
18:   if (regular_bb) then
19:     if (bbᵢ ∈ HotBBList then
20:       calib_sig(H, dᵢ)  
21:     end if
22:     cmp(bbᵢ_sig, H)
23:     end if
24:   else
25:     if (fan_in) then
26:       hot_cmp(bbᵢ_sig, H)
27:     else
28:       non_hot_cmp(bbᵢ_sig, N)
29:     end if
30:     cmp(bbᵢ_sig, N)
31:     end if
32:   end if
33: end if
34: // update hot sign and BBList
35: H = hotᵢ_sig
36: BBList_check ← bbᵢ
37: end for

*H*, fault-free control flow from *BB₄* to *BB₃* might be reported as an error. Thus, multiple collaborative signature verification (line 27-29) on both *H* and *N* should be applied at *BB₃*. Similarly, the last step of *HSCFC* inserts the control flow checking instructions and updates BB list (lines 22 to 38).

**IV. Experiments**

In this section, we present the experimental setup, including the fault injection model, and evaluate our proposed control flow checking methods against two other well-known methods.

**A. Fault Injection Model**

In order to measure the fault coverage, we conduct a fault injection campaign on various applications. We inject the faults on the branch target address to simulate a control error. Fig. 9 shows an example where the next instruction address should be 0x400c18 if the equivalence check at 0x400caf is evaluated to be false (e.g. %a ≠ %b). However, with the injected fault, a bit-flips at target address deviates the control flow to an incorrect instruction at 0x400d11. In this work, we use Intel PIN [16] to emulate fault injections at the assembly code. For each experimental trial, we select a dynamic control instruction as the injection point randomly (e.g. that follows a random uniform distribution). Moreover, we calculate the error significance using Statistical Fault Injections (SFI) [17] and perform 10⁶ injection experiments for each benchmark in order to present results with at least 95% confidence interval.

<table>
<thead>
<tr>
<th>addr.</th>
<th>inst.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x400C18</td>
<td>add %a, 1</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0x400caf</td>
<td>cmp %a, %b</td>
</tr>
<tr>
<td>0x400caf</td>
<td>jne 0x400c18-0x400d11</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0x400d11</td>
<td>mov %r1, %r2</td>
</tr>
</tbody>
</table>

**B. Benchmarks**

In this paper, we evaluate our proposed control flow checking algorithms using applications from different benchmark suites such as *Parboil* [18] and *SPEC2006* [19], as well as *StreamIt* [20]. These benchmarks are widely used in studying the performance of computing architecture and compiler.
optimizations. Table-I provides the detail information of investigated applications.

### TABLE I: Benchmarks Information

<table>
<thead>
<tr>
<th>Bench</th>
<th>Application</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parboil</td>
<td>histogram</td>
<td>two dimensional histogram.</td>
</tr>
<tr>
<td></td>
<td>mri-grid</td>
<td>computes a regular grid of data representing an MR scan.</td>
</tr>
<tr>
<td></td>
<td>stencil</td>
<td>implementation of a seven point stencil.</td>
</tr>
<tr>
<td></td>
<td>sad</td>
<td>sum of absolute differences</td>
</tr>
<tr>
<td>SPEC2006</td>
<td>401.bzip2</td>
<td>data compression.</td>
</tr>
<tr>
<td></td>
<td>429.mcf</td>
<td>single-depot vehicle scheduling.</td>
</tr>
<tr>
<td></td>
<td>456.hmmer</td>
<td>search DNA sequence pattern.</td>
</tr>
<tr>
<td></td>
<td>458.sjeng</td>
<td>artificial Intelligence.</td>
</tr>
<tr>
<td>StreamIt</td>
<td>filterbank</td>
<td>multirate Signal processing.</td>
</tr>
<tr>
<td></td>
<td>ft</td>
<td>fast fourier transform</td>
</tr>
<tr>
<td></td>
<td>matmul_block</td>
<td>block-based matrix multiply</td>
</tr>
</tbody>
</table>

### C. Compiler Infrastructure

**LLVM** is a commonly used compiler front-end for program analysis and transformation [21]. Thus, we build our proposed control flow checking solutions on top of LLVM framework. First, high-level application source code is compiled into an intermediate representation **LLVM-IR**. Then, according to the different methods, the control flow checking instructions are automatically inserted at the beginning of each basic block by our framework. For proposed **HSCFC** approach, **hot successor** information is obtained statically from LLVM’s built-in **BranchProbabilityInfo** pass, which employs statistical methods to calculate each target block probabilities [22]. A basic block is defined as **hot successor** if its “taken” probability is greater than 80%, and this threshold value was found empirically that leads to the best prediction results.

### D. Experimental Results

1) **Performance**: We evaluate the performance of different control flow checking techniques by measuring their overheads in terms of instruction numbers compared to the check-free program. In Fig.10, **CFCSS** and **RCFC** show the checking overheads of previous approaches, while **CSCFC** and **HSCFC** show the overheads with our proposed methods. From the experimental results, our proposed methods could reduce the overhead by 2x on average (e.g. the last column highlights the average values). In the worst case scenario (e.g. 458.sjeng), **CFCSS** leads to more than 70% checking overhead, which could be reduced to 32.71% by applying **HSCFC** instead. **CSCFC** shows even better performance improvement with further overhead reduction especially for applications with regular characteristics (e.g. 429.mcf and sad), however, this benefit is at the cost of small faults coverage sacrifice as shown in the following result. Also, for **HSCFC**, the checking performance also depends on the control flow structures. For example, **HSCFC** simplifies checking on regular basic blocks and only inserts more instructions for special **fan-in-out** structures to improve coverage. Therefore, in this part, we measure the percentage of regular structures existing in different benchmarks. Fig. 11 shows that regular basic blocks are more than 80% in general cases.

![Fig. 10: Overheads Comparison of Control Flow Checking](image)

Moreover, it should be noted that checking overheads varies among the different applications. The main reason for this diversity is due to different amount of control flows found
in different applications. Fig. 12 shows the distribution of different instructions (e.g. control or data) in each benchmark. Some applications are data-intensive (e.g. stencil or fft), and therefore they are less sensitive to applied algorithms. Otherwise, for control-intensive applications with frequent control flow transfers, the checking performance highly depends on the efficiency of the detection algorithm. For example, 18.99% of total instructions in 429.mcf are related to control flow behaviors, the overheads could be reduced from 72.85% to 34.87% when applying HSCFC instead of the traditional CFCSS.

2) Faults Coverage: Fault coverage is defined as the ratio of detectable errors to total faults, and it is another important evaluation metric. Using the experimental fault injection model described previously, we evaluate faults coverage provided by the different control flow checking algorithms. The injected faults are classified into one of the following categories according to its effect:

1. **OS** - control flow error produces a symptom such as segmentation faults, and execution is terminated by system anomaly.
2. **CF_Detected** - the injected fault could be detected by control flow checking instructions.
3. **Masked** - control flow errors did not corrupt the program output, and application level masking is included.
4. **SDC** - a silent data corruption occurs (i.e. computation is incorrect but without any error indication or warning).

In Fig.14, we show the faults distribution among different categories when applying different control flow checking techniques. From this result, we see that erroneous control flow targets will destroy the control flow integrity and most of them could be detected by monitoring the run-time basic block signature. Also, we observe that HSCFC roughly provides the similar detection rate as compared to previous works, while CSCFC shows a minor decrease in coverage primarily due to false negative related to the tight fan-in-out structures. Average faults coverage are 96.94%, 93.67%, 92.01%, 96.11% for CFSSC, RCFC, CSCFC and HSCFC respectively (e.g. the last column shows the average coverage). The most harmful error SDCs is limited to less than 10% for all of the control flow checking algorithms.

3) Hot Successor Prediction Accuracy: Static control flow prediction plays an important role on the effectiveness of HSCFC. Thus, to evaluate the static predictor, we compare the run-time branch target to the predicted successor at each control flow instruction. Fig.13 shows the prediction accuracy for different benchmarks. From this result, we can observe that control flow target could be predicted accurately with approximate 95% hit rates on average.

4) Detection Latency: Interval between fault occurrence and its detection is defined as fault detection latency. In many contexts, such as safety-critical systems, faults should be detected as early as possible especially. For example, when rollback recovery schemes are applied, earlier detection is helpful for preventing high recovery overheads. HSCFC and CSCFC monitor control flow at the granularity of each basic block, both methods show similar detection latency compared to CFSSC. For RCFC, it doubles the detection latency because it enlarges checking intervals (e.g. latency ≈ \(BB\_size \times 2\)).

V. CONCLUSION

Hardware faults are quickly becoming a design critical issue for computing systems build out of integrated circuits with sub-micron transistors. Control flow error is one of the hazards that may result in catastrophic execution failures due to soft errors. In this work, we discussed the performance limitation associated with conventional software signature approach, and also proposed two optimizations for reducing checking overheads by exploiting regular control flow structure. In particular two methods were presented. HSCFC employs static branch prediction to simplify the control flow checking on critical execution path, and CSCFC exploits the fact that most applications have simple control structure that applying common successor signature could be beneficial for overhead reduction. From our experimental results, we show that both techniques work well and they could be used to reduce the control flow checking overheads. Also, both of the proposed algorithms provide similar fault coverage compared to previous works.
Fig. 14: Fault coverage with different checking algorithms.

REFERENCES


