# Low Cost Control Flow Protection Using Abstract Control Signatures

Daya Shanker Khudia and Scott Mahlke

Advanced Computer Architecture Laboratory
The University of Michigan
{dskhudia, mahlke}@umich.edu

## Abstract

The continual trend of shrinking feature sizes and reducing voltage levels makes transistors faster and more efficient. However, it also makes them more susceptible to transient hardware faults. Transient faults due to high energy particle strikes or circuit crosstalk can corrupt the output of a program or cause it to crash. Previous studies have reported that as much as 70% of the transient faults disturb program control flow, making it critical to protect control flow. Traditional approaches employ signatures to check that every control flow transfer in a program is valid. While having high fault coverage, large performance overheads are introduced by such detailed checking. We propose a coarse-grain control flow checking method to detect transient faults in a cost effective way. Our software-only approach is centered on the principle of abstraction: control flow that exhibits simple run-time properties (e.g., proper path length) is almost always completely correct. Our solution targets off-the-shelf commodity embedded systems to provide a low cost protection against transient faults. The proposed technique achieves its efficiency by simplifying signature calculations in each basic block and by performing checking at a coarse-grain level. The coarse-grain signature comparison points are obtained by the use of a region based analysis. In addition, we propose a technique to protect control flow transfers via call and return instructions to ensure all control flow is covered by our technique. Overall, our proposed technique has an average of 11% performance overhead in comparison to 75% performance overhead of previously proposed signature based techniques while maintaining approximately the same degree of fault coverage.

***Categories and Subject Descriptors*** B.8.1 [*Performance and Reliability*]: Reliability, Testing, and Fault Tolerance

***General Terms*** Experimentation; Reliability

***Keywords*** Soft Errors; Control Flow Checking; Fault Injection

## 1. Introduction

In the quest to make chips faster, cheaper and energy efficient, transistors are being scaled down in size. As silicon technology is moving deeper down into the nanometer regime, reliability of microprocessors is emerging as a critical concern for manufacturers. Factors such as increasingly smaller devices, reduced voltage levels, and increasing operating temperatures exacerbate the problem of reliability of these components. Furthermore, billions of transistors are packed into modern microprocessors, and a fault in even a single transistor has the ability to corrupt the output of the application or crash the entire system.

In this work, we focus on the reliability concerns caused by soft errors. **Soft errors**, also referred to as Single Event Upsets (SEUs) or **transient faults**, are caused by high energy particle strikes from space or circuit crosstalk in an electronic circuit. A high energy particle such as a neutron from cosmic rays or an alpha particle from packaging material impurities releases charge in the circuit that in turn can disturb the functionality or the charge stored at a semiconductor device. As the name suggests, transients faults do not cause permanent damage to the chip and devices work correctly once the effect of the fault is over.

The semiconductor industry has reported many instances of the problems caused by soft errors over the last few decades. In 1978, one of the first soft error instances occurred when the packaging material used in the chip produced by Intel became contaminated with uranium from a nearby mine [23]. In another instance of soft errors, Cypress semiconductor reported that a single soft error caused a billion-dollar automotive industry to halt every month [40]. In 2005, HP also reported [27] that cosmic rays were the cause of frequent crashes of its 2048-CPU system installed at the Los Alamos National Laboratory. These studies illustrate the issues caused by soft errors and necessitate the need for reliability solutions at all levels (e.g., circuit, architecture or application level) of the system stack.

Traditionally, memory cells have been more vulnerable to transient faults and are usually protected by mechanisms such as parity checks or Error Correcting Codes (ECC). The use of smaller transistors to implement logic circuits in microprocessors increases susceptibility of logic circuits to transient faults. Shivakumar et al. [35] reported that Soft Error Rate (SER) for the logic on chip is steadily rising with technology scaling while SER for memory is expected to remain stable. SER is the rate at which a component encounters soft errors. Also, SER scales with number of transistors and level of integration [12]. Without actively addressing these issues, SER is expected to rise significantly in new products. Moreover, Venkatasubramanian et al. [37] reported that more than 70% of the transient faults lead to disturbance in control flow and are the cause of control flow errors. Control flow errors are defined as the incorrect change in the sequence of instructions executed by processors under the influence of external events such as soft errors.

Traditional solutions in server space for reliability have provided fault tolerance via DMR (dual-modular redundancy) and TMR (triple-modular redundancy). IBM Z-Series [5] servers and HP NonStop [6] systems are two pioneers of such schemes. These solutions incur a large energy and/or performance overhead and are not directly applicable in the embedded design space. Signature based solutions [29] employ signature updates in every basic block and check that all control flow transfers lead to a correct target address. This checking results in high instruction overheads due to the combination of computing, updating, and checking the unique control signatures of each potential control flow edge. Typical performance overheads of prior work are on the order of 75% (Section 2.3 describes such techniques in detail).

In this work, we propose Abstract Control Signatures (ACS) to provide a practical low cost solution for Commercial Off-the-Shelf (COTS) embedded microprocessors to protect against control flow target (i.e., the branch destination address) errors. These errors are usually not covered by redundancy-based data protection techniques [10, 15], yet they lead to a disproportionately high number of incorrect executions. ACS is a software-only solution and does not require any modifications in the hardware. Our solution



*LCTES 2013*, June 20–21, 2013, Seattle, Washington, USA.

is based on the principle of abstraction and the insight that control flow that exhibits simple but repeated properties of correctness is almost always entirely correct. ACS achieves abstraction by checking simpler properties (e.g., path length) and promoting control flow signature checking from individual basic blocks to group of blocks.

ACS is targeted for COTS commodity systems. In the commodity embedded market, achieving performance targets in a cost-effective manner is of paramount importance. Due to the associated cost of providing high reliability, commodity systems typically cannot target 100% protection against faults. Our solution is designed considering these requirements of embedded market space. The proposed solution provides opportunistic fault coverage but does not guarantee 100% fault coverage and hence is not applicable to mission critical systems. The contributions of this work are as follows:

- A novel abstraction based technique to insert simplified signatures. Under the proposed scheme, more complex signatures can be used to explore trade-offs in performance overhead and fault coverage.
- A novel region based method to insert checking at a coarse granularity abstracting away the details of fine-grain control flow.
- A global signature based method for protecting control flow transfers through *call* and *return* instructions.
- Microarchitectural fault injection experiments to validate ACS.

## 2. Background and Motivation

In this section, we present background details that are necessary to understand ACS and discuss the motivation behind the approach.

### 2.1 Fault Detection

In order to protect against transient faults, detection of these faults is a necessary first step. Fault detection can be achieved by introducing some form of redundancy. For example, time redundancy involves executing the same instructions twice on the same hardware, space redundancy involves executing the same instructions on duplicate hardware and information redundancy involves usage of parity, ECC etc. High reliability systems typically use a mixture of fault detection techniques such as DMR/TMR and/or ECC for protection against soft errors. These solutions are too expensive in terms of energy/performance/area overheads (∼100%) to be used in the embedded market. A relatively inexpensive class of solutions for commercial market use time redundancy based software-only techniques. *Data flow* and *control flow* checking are usually employed in software-based techniques [8, 10, 29, 30, 37] against soft errors. Data flow checking ensures that computation (e.g., addition) is correct. Software-based data flow checking techniques work by replicating instructions. Control flow protection techniques usually employ signatures to ensure correct control flow [29, 37]. A brief comparison of related technique to ACS is shown in Table 1. The techniques are classified based on their relative performance overhead and whether they handle data flow errors, control flow errors or both. Control flow protection techniques are further classified into two categories based on whether they protect branches and call/ret instructions. The techniques are also classified based on their relative performance overheads. Techniques having overhead ∼70% or more are in high overhead row and those with ∼40% or less in low overhead row. Typically low overhead techniques reduce overhead by sacrificing on fault coverage. A more detailed description of related work is presented in Section 6.

Table 1: Brief comparison of ACS with other techniques.

| | Data flow | Control flow | |
|---|---|---|---|
| | | Branch | calls/rets |
| High overhead | DMR, TMR<br>SWIFT [29]<br>EDDI [30] | DMR, TMR<br>SWIFT<br>EDDI<br>ALLBB [8]<br>ACFC [37] | DMR, TMR<br><br><br>ALLBB (ret only) |
| Low overhead | Shoestring [10]<br>ProfileBased [15]<br>ACS+ProfileBased | <br><br>ACS | <br><br>ACS |

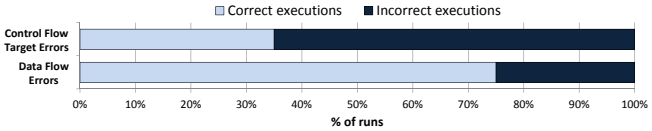



Figure 1: Control flow target errors are ∼2.5x as likely to cause incorrect executions.

Figure 1 shows the number of incorrect executions resulting from errors in register files (corrupting the data) and branch targets for SPECINT2000 benchmarks. A high masking rate (∼75%) for data errors is consistent with the reported masking data in previous works [10, 39]. On average, errors in the branch targets are ∼2.5x more likely to result in incorrect executions. Hence, in this paper, we focus on efficient detection of control flow errors, in branches as well as call/ret instructions, and our technique can be combined with previously proposed [10, 15, 29] code duplication based solutions for a complete solution (see Section 5.3 for a combined solution).

### 2.2 Control Flow Errors

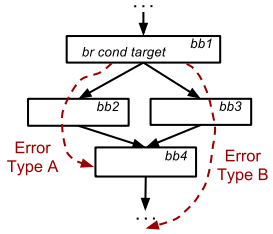



Figure 2: Control Flow Target Errors: Corruption of branch target can result in nearby (Type A) or far away (Type B) displacement of control flow.

To better understand control flow protection techniques, we need to comprehend the various cause of control flow errors. A control flow error can occur in a non-control flow (e.g., *add*) or in a control flow (e.g., *branch*) instruction. A non-control flow instruction of the application can be converted into a control flow instruction by a soft error thus erroneously affecting control transfers. Errors occurring in control flow instructions can be divided into two categories: Firstly, **control flow condition errors** are caused by the errors in the direction of a conditional branch. Secondly, **control flow target errors** are caused by the errors in the destination of a branch. Branch conditions are usually protected by data flow protection schemes by duplicating the computation leading to a condition. As shown later in Figure 10 (Section 4.3), the errors in branch targets result in disproportionately high number of incorrect executions. Hence, we focus on the control flow disturbances caused by the errors in branch targets. From here onwards, unless otherwise specified, the use of control flow errors with respect to ACS refers to the errors in branch targets. Figure 2 shows a part of a Control Flow Graph (CFG) containing 4 Basic Blocks (BBs). Two types of errors that affect branch target are also shown in the Figure. Type A errors cause the erroneous jump to nearby locations and Type B errors direct the control flow to far away locations. Type A errors cause the program to skip a few instructions and are more likely to result in masking or program output corruptions. In contrast, Type B errors are more likely to crash the program either by directing the control flow to out of program scope or to a different function in the same program. In Section 3, we describe how our proposed method handles these control flow errors.

### 2.3 Signature Based Techniques and Associated Overheads

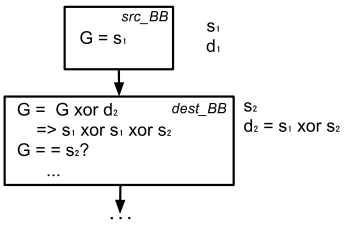



Figure 3: Basic signature scheme: If the correct control flow transfer takes place, G at $dest\_BB$ would be equal to $s_2$ otherwise not.

Many of the previously proposed software-only techniques for control flow protection embed signatures or assertions into BBs at compile time [3, 13, 29]. This section briefly describes the fundamentals of these signature based techniques, especially CFCSS [29]. CFCSS assigns a unique signature $S_i$ to each BB in the program. A general purpose register (G) is used to hold the signature of the currently executing BB. G is initialized to the signature of first BB when a program starts. Subsequently, whenever a transition is made from $src\_BB$ to $dest\_BBs$ the value of G is updated with the newly computed value. This new value is calculated by taking the *xor* of G and the static signature difference (*xor*) of $src\_BB$ and $dest\_BB$. After this, G should be equal to the unique signature assigned to $dest\_BB$. A comparison of G with unique value of $dest\_BB$ is inserted in $dest\_BB$ to make sure that control flow is correct. If this comparison fails, an incorrect control flow transfer has taken place. A simple case of this scheme is shown in Figure 3. For a complex case of branch-fan-in nodes, extra dynamic adjusting signatures must be inserted to avoid aliasing [29]. This necessitates the need for multiple signature updates in branch-fan-in nodes and dynamic signature computation in predecessors BBs of the branch-fan-in nodes. These extra updates contribute to the overhead of such a scheme.

Essentially, every BB in the application contains signature computation or update instructions as well as comparison instructions for ensuring correct control flow. The cost of embedded signature checking at runtime in every BB can be prohibitive, making these techniques impractical. We have implemented CFCSS and in our experiments on small benchmarks (the same ones used in CFCSS [29]) we observe, on average, a performance overhead of 68%. Though, for *Insertsort* benchmark from the set of benchmarks, it is as high as 222%. For real representative benchmarks from SPECINT2000, we observe up to a 144% overhead (75% on average) for the CFCSS technique. The opportunity to reduce this huge overhead is one of the motivations behind proposing ACS.

## 3. Abstract Control Signatures

Fundamentally, there are two critical aspects of any signature based control flow protection scheme. The first is signature computations (or updates) in each BB and the second is signature comparisons (or checking) to check for erroneous control flow. These two computations are the main contributors to the performance overhead of signature-based control flow checking schemes. To reduce performance overhead, we propose raising the level of abstraction of signature checking and simplifying signature updates in every BB. The abstraction level is raised by working at the levels of **regions***. The whole program is divided into regions that are larger than just a BB. These regions are more than just a collection of BBs and ideally should possess certain properties that help in minimizing the number of signature comparisons and signature updates. Each region has a signature variable associated with it. For example, one

*In this paper, **region** is used to refer to a single entry multiple exit code section that satisfies the following property among others: loop back edges are only allowed to the entry node (see Section 3.1).

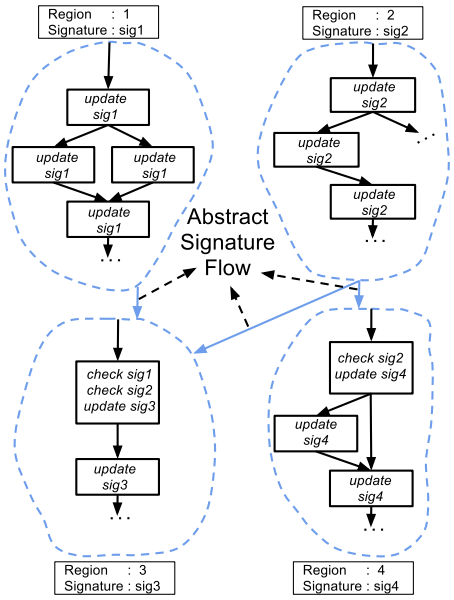



Figure 4: Abstract signatures: The whole program is divided into regions at a higher abstraction level. Such regions are enclosed by dashed light blue (grey) lines in this Figure. Every region is assigned a signature. Every abstract region updates its signature based on the control transfers among the BBs inside it. These signatures are only checked in other abstract regions.

desirable property of regions is to have a single entry point so that the associated signature variable need not be initialized at every entry point. As shown later (Section 3.2), this reduces the number of required signature comparisons. The signature variable associated with a region is checked in other regions that are the target of the control flow edges from the region under consideration. Essentially, signature information flows between these abstract regions. The signature associated with each region represents the correctness of control flow internal to that region. In this sense, checking control flow outside regions abstracts away the details about control flow inside a region, hence the name ACS (Abstract Control Signatures). A high level diagram for ACS concept is shown in Figure 4. In Figure 4 the signature *sig1* is associated with region 1 and is updated inside the BBs of region 1. Assuming a BB in region 1 has a control flow edge to a BB in region 3, *sig1* would only be checked in that BB in region 3.

### 3.1 Design of ACS

The idea of ACS is very generic and can be realized in various ways. ACS can be implemented by forming regions at various granularity levels and different signature updates according to the required trade-offs in performance overhead and fault coverage. The signature update inside each BB can also be tuned. For example, the signature update inside each BB can be as simple as having a parity bit set/reset and the corresponding check would be to check against 0 if even number of blocks were traversed and against 1 if odd number of blocks were traversed. These updates can be more complex such as usage of hash functions or *xors*. Similarly, the region formation can also be customized. For example, if the region is a single BB then this scheme is the same as regular signature checking in each BB.

For ACS implemented as a part of this work, we have made following choices for signature updates and regions. We use a simple counter variable as the signature. For signature updates, we increment the signature by 1 in the beginning of every BB. The intuition behind using increment by 1 is as follows: Consider 2 points in a

program, X is a region entry and Y is the corresponding region exit. If control reaches X, we expect it to reach Y. If in going from X to Y, a valid number of BBs are traversed and the first instruction in each of those BBs is executed, we hypothesize that control flow is likely correct. Obviously, this is not always true, but our experiments have confirmed that small disruptions (fault in the lower bits of the branch target) in the control flow will result in changes to the path length due to positioning of counter updates at the beginning of BBs and large disruptions (fault in upper bits) will result in Y never being reached. Thus, if the hypothesis is statistically true, individual control transitions need not be checked with minimal loss in fault coverage. This allows only the higher level information to need checking. To see the usefulness of such counters, let us consider the control flow errors shown in Figure 2. On one hand, Type B errors (far away erroneous jumps) that would transfer control from one region to another, are easily caught. On the other hand, Type A errors (nearby erroneous jumps) are likely to skip the signature updates, so they are also caught. We use intervals [2] as regions because of the desirable properties they possess.

**Intervals:** An interval is a set of BBs such that every BB except the header BB in the interval has its predecessors in the interval. An interval satisfies the following, and many other, properties.

1. The header block of an interval dominates all the BBs in that interval. Basically, this implies that control can only enter at the header node of an interval.
2. If a loop is part of an interval then the loop header and interval header are the same. The header BB of a loop is the target BB of back edges in that loop.

Figure 5 shows an example of intervals for a CFG that has nested loops. Interval 1 contains only *bb1* and its header is also *bb1*. Interval 2 contains all the remaining blocks shown in the Figure. Interval 2 contains a loop and note that loop header *bb2* is also the header node of the interval 2. Another interesting observation is that the outer loop is never contained in a single interval. We use intervals formed according to the maximal interval definition [26]. A **latch BB** of a loop is defined as the block that has a branch to the header of the loop. For example, *bb_latch1* is the latch block for the inner loop starting at *bb2*.

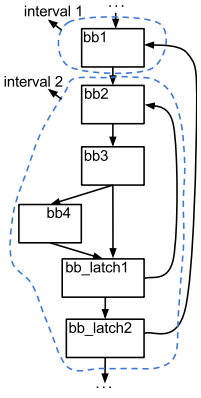



Figure 5: Intervals in the Figure are shown by enclosed dashed light blue (gray) lines. This Figure shows two intervals for a control flow graph that has a nested loop.

A basic overview of the implemented scheme is shown in Figure 6. The counter $C_1$ (signature for the shown region) is incremented by 1 in each BB, and in the successor BB of *bb4*, a check would be inserted to make sure that the value of $C_1$ is 3. In the presence of a control flow error, assume that the transition happens such that after *bb1*, either signature updates of *bb2* or *bb3* is skipped or *bb4* is executed. The signature value would not be 3 in the successor BB of *bb4* and this would be detected. We put the increment as the first instruction in the BBs so that the signature won't get updated in case of small erroneous jumps. Thus, very small changes to

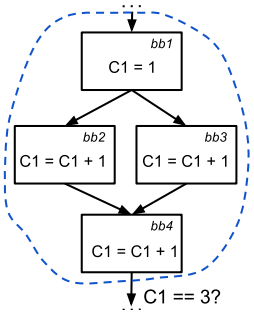



Figure 6: Every interval is associated with a signature. In our scheme, signature are simple counters. The signature is initialized in the header and incremented by 1 in other blocks. The signature checks are made in the BBs that are destination BBs of exits out of an interval.

the branch target are caught because of this positioning of signature updates.

However, if we naively insert the increments in each BB of the program, the counter value at the exit points of the interval will depend on 1) the path taken during runtime 2) the particular exit taken. For example, consider the CFG shown in Figure 7. If at runtime, edge *bb1* $\rightarrow$ *bb2* is traversed, the signature value at the exit out of *bb3* would be 3 since each BB increments signature by 1. However, if the edge *bb1* $\rightarrow$ *bb3* is traversed signature value at the same point would be 2. Another similar problem exists if there are multiple exit points from an interval. The signature values at the exit points of an interval would be different if the exits originate from different BBs. Different signature values from an interval exit would imply that checks would need to be inserted with different values. To solve this problem, we make sure that from every exit out of an interval, the same signature value needs to be checked no matter which exit is taken. To tackle the aforementioned problems, we have developed a method to calculate extra balancing increments required along edges. The details of this method are described in the next subsection.

### 3.2 Calculating Balancing Increments

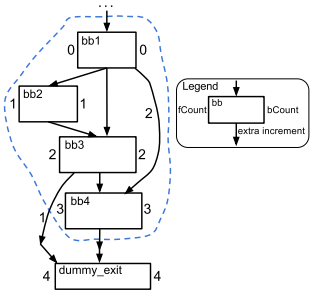



Figure 7: The extra increments required to be inserted along control flow edges is shown. This balances out signature values at the exits out of an interval.

The goal is to calculate the extra balancing increment required to be inserted along the imbalanced edges in the CFG. Figure 7 shows an imbalanced CFG. An imbalanced CFG implies that at every exit there could be multiple signature values depending on the path traversed during runtime. If the CFG is not balanced, we will need to check against multiple values at exit points. Checking against multiple values will require multiple comparison instructions.

We solve these problems by using a technique of slack distribution, a modified version of the algorithm used by Chu et al. [9] for optimal work partitioning. Our adapted version of the technique works as follows: First, every exit out of an interval is connected to a dummy exit node. All BBs in the interval are assigned a $fCount$

of 0. All $edgeWeights$ are initialized to 1 and represent an initial increment along the associated edge. $fCount$ is a number associated with each BB that represents the path length from the header of an interval to the BB under consideration. The algorithm starts from the header BB of the interval. By iterating over predecessors, the sum of $edgeWeight$ and $fCount$ for each predecessor is calculated. $fCount$ for the current block is then maximum value over all predecessors. This can be written as follows: $fCount(bb) = \max_{x \in predecessors(bb)}(fCount(x) + edgeWeight(x \rightarrow bb))$. For every interval, this calculation is repeated until there is no change in $fCount$ value of any BB. The pseudo code of the algorithm is described in Algorithm 1. Every BB is also associated with a number called $bCount$. $bCount$ is the number calculated starting from dummy exit nodes and traversing the predecessors. $bCount$ are initialized to $fCount$ for each BB. Using an algorithm similar to the one shown in Algorithm 1, $bCount$ is calculated for every BB in the interval. The update equation of $bCount$ is as follows: $bCount(bb) = \min_{x \in successors(bb)}(bCount(x) - edgeWeight(x \rightarrow bb))$. Note that during the calculation of $fCount$ and $bCount$ only the successors and predecessors that are in the interval are considered. The dummy_exit block is considered a part of the interval during analysis. Once the $fCount$ and $bCount$ calculation is completed for every BB in the interval, the amount of extra balancing increment to be inserted along an edge between $srcBB$ and $destBB$ can be calculated as follows: $extraIncrement[srcBB \rightarrow destBB] = fCount[destBB] - edgeWeight[srcBB \rightarrow destBB] - bCount[srcBB]$.

Figure 7 shows an example of extra increment calculation for a CFG. Numbers on the left side of blocks represent $fCount$ and numbers on right side of the BB represent $bCount$. Numbers on the edges are the extra increments required to be inserted along that edge. e.g., based on the algorithm described above edge *bb1* $\rightarrow$ *bb3* edge gets an increment of 1 and edge *bb1* $\rightarrow$ *bb4* gets an increment of 2. Once this step is executed, all the required increments are inserted along all edges of an interval.

```
Create dummy_exit block and connect all exit edges to this block;
Initialize all edgeWeight to one;
Initialize all fCount to zero;
change = 1;
while change do
    change = 0;
    for each bb in Interval do
        maximum = max(fCount(x) + edgeWeight(x → bb)) for x in
        predecessors[bb] and x → bb is not a backEdge;
        if fCount[bb] < maximum then
            change = 1;
            fCount[bb] = maximum;
        end
    end
end
```

**Algorithm 1**: Algorithm for calculating $fCount$ for every BB in an interval.

### 3.3 Error Detection Analysis

Let $C_i$ be the counter associated with an interval. Every block inside that interval updates the counter by 1 and at every exit out of the interval the counter value should be the maximum path length (since we insert balancing increments) through that interval. Let that max value for an interval be $CMax$. If $C_i$ is not equal to $CMax$ when control exits out of the interval then the control flow inside the program got disturbed. For all the intra-interval control flow errors, if any update to the path length counter is skipped, the path length calculation would be wrong and hence the control flow error will get caught. Erroneous jumps to other intervals are detected as the path length is not correct at the entry point of those intervals. However, there could be multiple paths of same length inside the interval. In the presence of single errors, the probability of traversing a different path of the same length path and still having the same $CMax$ at exits is very low as explained below. We refer to this probability as aliasing probability. Consider two BBs $BB_i$ and $BB_j$ and assume that an error occurs while executing the branch in $BB_i$ transferring control to $BB_j$. In such a case and under single bit errors, aliasing occurs if all of the following three conditions are satisfied:

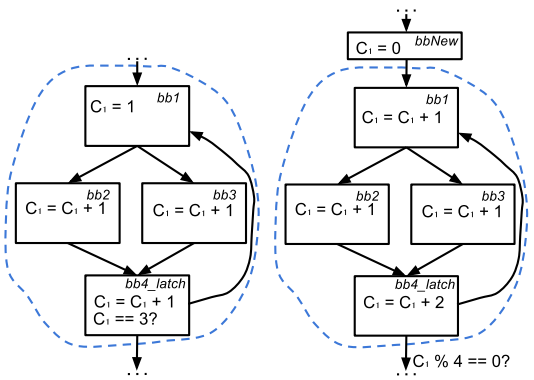
$$\begin{cases} pathLength(BB_j) == pathLength(BB_i) + 1 \\ BB_j \notin successors(BB_i) \\ BB_i \text{ jumps to the first instruction of } BB_j \end{cases}$$

$pathLength(BB_i)$ is the length of the path (number of BBs required to be traversed) from the interval header to $BB_i$. The first condition implies that the path length at erroneous destination block should be 1 more than source block. The second condition requires that $BB_j$ is not a valid successor of $BB_i$ according to the CFG and the third condition requires the jump to be at the beginning of the BB. If the jump is not at the beginning of the $BB_j$, the counter update would be skipped and the error would be caught. Fortunately, this a very specific case, so the aliasing probability is very low, dependent on the structure of the CFG. For SPECINT2000 benchmarks, the probability of such an aliasing is on the order of $10^{-5}$. This is calculated by analyzing the CFG for such a case. This probability encompasses the aliasing probability between predecessor blocks (an erroneous jump between two predecessors) of a common successor BB in the same interval. An erroneous change in branch condition can transfer control to a statically valid target in the CFG and is another case of aliasing. We assume that such a case can be handled by data flow protection methods.

### 3.4 Insertion of Checking Instructions

An important part of the technique is to find the BBs where the comparison instructions should be inserted. Each interval has a unique signature variable. We compare this variable with the statically known $CMax$ to test that the proper number of increments occurred. For our initial implementation, we chose to insert checks at all the exit points of an interval and in the latch block of loops. However, this is suboptimal and in the next section we show that how this can be further optimized.

### 3.5 Optimization for Loops




(a) Checks for loops (b) Optimized insertion of checking instruction

Figure 8: Optimizing signature checking for loops: The checks on signatures are moved out of loops to exit blocks so that they are not executed in each iteration.

A naive way to insert a checking instruction for a loop is as shown in Figure 8(a). The latch block contains the checking instruction (the instruction that compares $C_1$ to 3). However, in this situation, the check is executed once every loop iteration. This can be optimized as shown in Figure 8(b). In the optimized case, the checking instruction is moved out of the loop and check is now made against a remainder. Essentially, a multiple of loop path length gets tested by the remainder. Remainders are a costly operation and one important issue to consider here is the fact that loop increments are inserted in

such a way that the remainder is always taken by a power of two. This is shown in Figure 8(b), in *bb4* counter $C_1$ is incremented by 2 instead of 1 to make sure that remainder by 4 is taken. If this is the case the remainder instruction can simply be converted to a bitwise *and* instruction (e.g., remainder by 4 can be computed as bitwise *and* with 3).

### 3.6 Call and Return Instructions

A source of control flow transfers are *call* and *return* instructions. In this work, we propose a new technique to protect the control flow from caller to callee header and the return from callee to caller. The idea is akin to the path length approach used for branches except each function call has a unique path length (a unique number) that is checked upon entry of the callee and upon return to the caller to ensure call/returns go to and return from proper targets. We make the path length unique for each function to ensure that there is no aliasing among calls to different functions. The technique works as follows: Let $F$ be the set of all the functions in an application. Every $f_i \in F$ is assigned a unique code such a way that the following is true.

$$HammingDistance(ACode(f_i),\ ACode(f_j)) > 1$$
$$\forall f_i,\ f_j \in F\ and\ i \neq j$$

For binary numbers $a$ and $b$, Hamming distance is equal to the number of positions at which the corresponding bits in $a$ and $b$ are different. Simply put, Hamming distance is the number of errors required to transform $a$ to $b$, and vice versa. $HammingDistance(a, b)$ is the Hamming distance between $a$ and $b$. $ACode(f_i)$ represents the code assigned to $f_i$. Every function in the application is assigned a unique code in a way such that the Hamming distance between any two codes is greater than 1. This ensures that there is no aliasing among calls because of erroneous transition from one function to another function in presence of single bit errors. Figure 9 shows an example of instrumented code. The Global Signature Register (GSR) is updated before and after the call as shown. RConst is a convenient constant (power of 2) chosen in a way such that the costly remainder operation can be converted to simple bit shift operation. 34 is the Hamming code assigned to the function Foo. Inside the callee, the GSR is updated in the entry BB of the callee and return BB of the callee. This ensures that other calls inside Foo can use the same type of instrumentation. If source code of a call (e.g., library calls) is not available then the increments inside the callee cannot be inserted and only the increments around the call are inserted. In such a case, the transition to the beginning of the callee cannot be checked but the instrumented code ensures the return from callee should be right after the call instruction. Calls through pointers and compiler built-ins (i.e., compiler intrinsics) are treated in the same way as library calls.

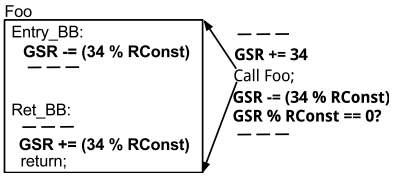



Figure 9: Handling call and return instructions. Instructions in bold represent the inserted instructions.

## 4. Experimental Setup

A common practice in the literature to evaluate transient fault detection solutions is to use Statistical Fault Injections (SFIs) into a microarchitectural model of a processor. We believe that SFI provides the opportunity to inject faults into various hardware structures and hence are close to real transient fault scenarios. SFI has been previously [10, 32] used in validating the solutions proposed to protect against soft errors.

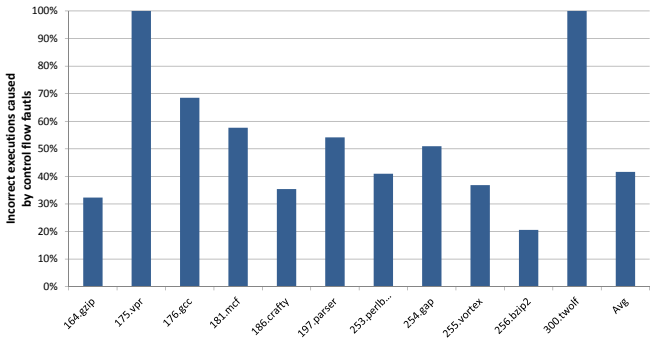



Figure 10: The incorrect executions as a percentage of unmasked faults caused by disturbance in control flow targets. Faults are injected in register file as well as branch targets.

### 4.1 Compiler Transformations

We have used the LLVM [16] compiler infrastructure to insert ACS into an application's code. Firstly, application source code is converted into LLVM's internal representation called LLVM IR (Intermediate Representation). ACS is implemented as a pass over LLVM IR. ACS insertion pass should be run after all the optimization passes on IR so that these passes do not interfere with ACS code. Our ACS insertion pass takes IR as input and as output it generates IR with signature computations and checks embedded into it. An LLVM interval formation pass is internally run and the information is used to insert control flow checking signatures. Some optimization passes such as constant propagation in code generation phase can propagate the constant initialization of signatures into the next BB. This can effectively remove the effect of inserted signature from the BB where the signature was initialized to its successors BBs. We have disabled such optimizations during the phase when LLVM prepares the IR for code generation.

### 4.2 Benchmarks

We have used 11 benchmarks from the SPECINT2000 benchmark suite (*gzip, vpr, gcc, mcf, crafty, perlbmk, parser, gap, vortex, bzip2, twolf*) as representative workloads in experiments. All these benchmarks were compiled with -O3 option of gcc frontend for LLVM. SPECINT2000 benchmarks. In the context of embedded systems, if the change in execution time affects program output, these programs might not run correctly after control flow protection. We do not consider multithreaded benchmarks in this work. However, we do not foresee any problems of using ACS with multithreaded programs.

### 4.3 Fault Injection Campaign

To evaluate the proposed approach, we ran an extensive fault injection campaign. An acceptable way in literature to model transient faults is using single bit-flips. These faults are inserted by flipping a random bit at a random cycle during the course of the application run. We injected faults in the register file (a large part of the processor's architectural state) and branch targets. A fault in a register used as branch target or in the computation of branch targets for indirect branches can disturb the control flow. Figure 10 shows the results of this experiment and Y-axis in the Figure is incorrect executions caused by control flow target errors as a percentage of the unmasked faults. The results show that a large percentage (on average 42%) of the unmasked faults result in incorrect executions and are caused by control flow faults. *175.vpr* and *300.twolf* (100% bars in the Figure 10) have high masking rate and all the remaining incorrect executions for these two benchmarks are caused by control flow faults. Even though the size of branch target (32 bit) is smaller than register file (16 registers of size 32 each), the contribution of branch target errors to incorrect executions is disproportionately high. Hence, control flow faults are an important category of faults to consider. Therefore, for the rest of the experiments,

we chose to inject faults in branch targets only. Injecting faults in branch targets represents stress testing (a pessimistic case) control flow protection schemes since all the injected faults are guaranteed to disturb the control flow and subsequently do not inflate coverage numbers as they result in less masking compared to data errors as shown in Figure 1. The same method of fault injection is used for the baseline (CFCSS). To inject a fault, the program runs normally until it encounters the first control flow instruction after the selected random point is encountered. Once the control flow instruction is selected, a random bit is selected from the target address of the control flow instruction. This selected random bit is flipped to complete the injection of fault. Faults in PC (Program Counter) and other address circuitry are expected to disturb the control flow in a similar manner. Our technique is also capable of detecting faults injected into other microarchitectural units that affect the program control flow.

We used the GEM5 [7] simulator to simulate the workloads and implemented fault injection infrastructure into this simulator. The simulator was run in ARM syscall emulation mode and modeled the ARMv7-a profile of ARM architecture. To obtain performance overhead, workloads are simulated in an out-of-order model of the target processor. We use atomic model for processor configuration to inject control flow faults. The details of the processor configuration for out-of-order model used for the experiments are in Table 2.

Table 2: GEM5 Simulator parameters (models an ARMv7-a profile of ARM architecture).

| **Processor core @ 2GHz** | |
|---|---|
| Simulation configuration | out-of-order core |
| Simulation mode | Syscall emulation |
| Physical integer register file size | 256 entries |
| Reorder Buffer Size | 192 entries |
| Issue width | 2 |
| **Memory** | |
| L1-D cache | 64KB, 2-way |
| L1-I cache | 32KB, 2-way |
| DTLB/ITLB | 64 entries (each) |

We have chosen to inject 1100 faults per technique to evaluate the solution. The statistical significance of these faults can be calculated by leveraging the work done by Leveugle et al. [17]. The calculation for our experimental setup shows that we need 96 fault injection trials for each benchmark to have a 10% margin of error and confidence level of 95%. Note that the margin of error only applies to fault coverage data. The performance overhead shows the exact simulation cycles consumed by the simulator. Therefore, we chose 100 fault injection trials for each benchmark to yield results with reasonable accuracy in a timely manner. After the fault injection, the program runs until completion. The result of each simulation trial is classified into one of the following five categories:

- **Masked**: The injected fault did not corrupt the program output. Application-level or architecture-level masking occurred in this case.
- **HWDetect**: The injected fault produces a symptom such as a page fault so that a recovery can be triggered. A fault is considered under this category only if the symptom is produced within a number of cycles (2000 for our experiments) after the fault was injected.
- **CFDetect**: The injected fault was detected by the control flow checking instructions inserted at the time of compiler transformation.
- **Failure**: The injected fault resulted in out-of-bound address access and resulted in simulation termination. Also, faults causing infinite loops in the program are classified under this category.
- **SDC**: Faults that corrupt the program output are classified into this category. These are Silent Data Corruptions.

Traditionally, the fault tolerance research community considers a program to be correct if the architectural state is correct at every cycle. Li et al. [20] showed that 17.6% of the multimedia and AI applications showed correct results even though they had architecturally incorrect states. We believe that user-visible program output corruptions truly matter to end users and cycle-by-cycle correct architectural state is not important to them. So in the context of evaluating this work, a program is considered to have executed correctly if the final output of the program matches. The result classifications of the injection experiments in this work are based on the fact that only program output corruptions really matter. Therefore, for this work we do not regard the number of faults that propagate to the microarchitectural state as a metric of importance. The percentage of faults that actually do corrupt program output are considered harmful because these faults corrupt program output without any hint of failure and represent the worse case scenario.

### 4.4 Recovery Support

ACS, like CFCSS, is a detection-only solution for control flow errors. Once a control flow error is detected, we rely on a recovery mechanism to recover from the detected error. A software-only recovery scheme such as Encore [11] or checkpointing-based recovery schemes can be used in conjunction with our solution. Feng et al. [10] and Wang et al. [38] proposed that future microprocessors with aggressive performance speculation will need recovery support. If available, the same scheme can be used by our solution. However, the cost of checkpointing-based and software-only schemes increases with respect to the number of instructions executed from recovery point. So, one important target for our scheme is to keep a bound on fault detection latency.

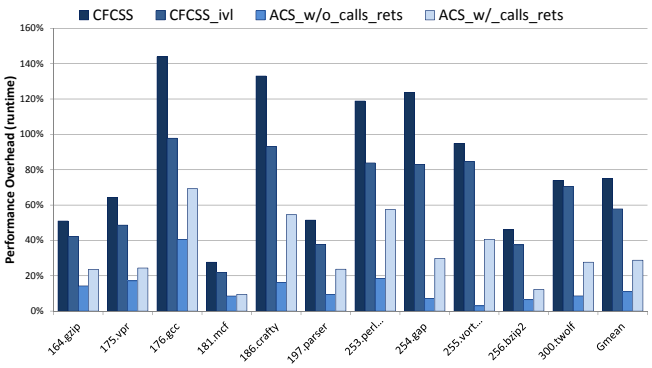



Figure 11: The performance (Runtime on simulated core) overhead for all techniques.

## 5. Experimental Evaluation and Analysis

Using the experimental setup described in Section 4, we obtain performance overhead and fault coverage results. Figure 11 shows the performance overhead measured in terms of runtime. These overheads are in comparison to unmodified applications compiled at -O3 optimization level. CFCSS shows the runtime overhead for the CFCSS scheme [29] and CFCSS_ivl bar shows the instruction overhead if the interval information is used in conjunction with CFCSS to insert checking at a coarser granularity. CFCSS_ivl has the *xor* (same as CFCSS) signature update inside every BB and in contrast to CFCSS only signature checking is moved at a coarser granularity. Also, CFCSS_ivl does not have any loop optimizations (Section 3.5). The third and fourth bar for each benchmarks shows the runtime overhead when we use ACS. ACS_w/o_calls_rets bar in this Figure shows the overhead without the protection for calls and returns (Section 3.6) and ACS_w/_calls_rets the overhead if protection for calls and rets is included. Overall, the performance overhead is 75%, 57.8%, 11% and 28.8% for CFCSS, CFCSS_ivl, ACS_w/o_calls_rets and ACS_w/_calls_rets, respectively. We have

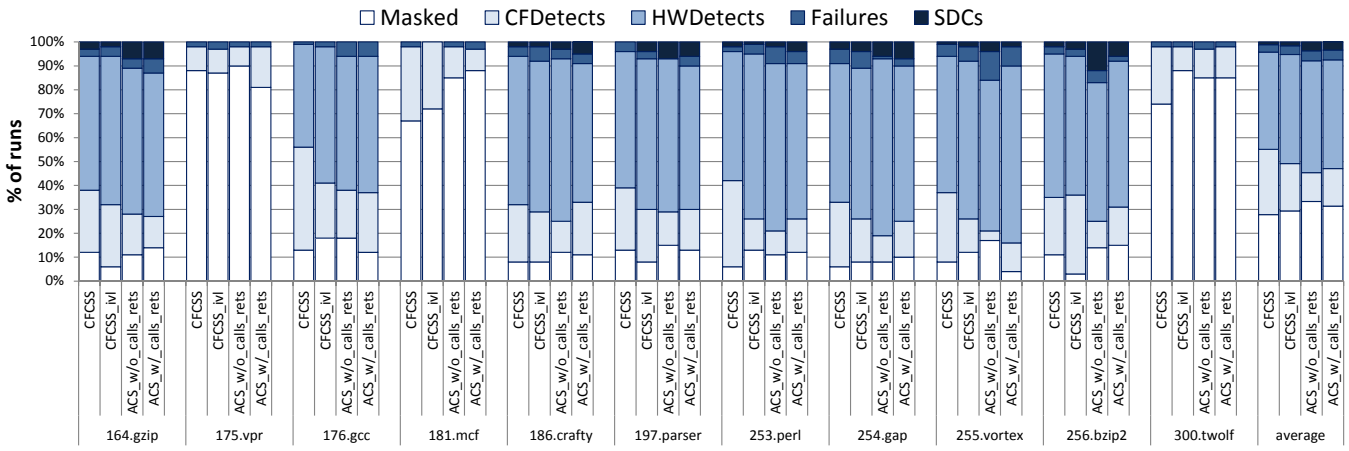



Figure 12: CFCSS bar shows the fault coverage for CFCSS and CFCSS_ivl shows the fault coverage with checking inserted using interval information. ACS_w/o_calls_rets shows the fault coverage without protection for calls/returns and ACS_w/o_calls_rets shows the fault coverage if calls/returns are also protected.

also measured the impact of code size expansion on application binaries and on average code size overhead is 22% with ACS. The code size overhead is largest for *176.gcc* showing largest performance overhead. To give more insight on the reduction in overhead, we measured the number of intervals and basic blocks in benchmarks. On average, there are 13302 basic blocks and 1993 intervals across the evaluated benchmarks and the number of checks required to be inserted are 2461. This represents a 5.4x decrease in the number of checks by abstracting from BBs to intervals.

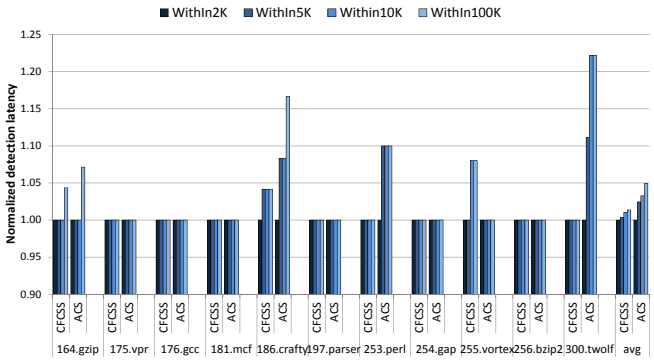



Figure 13: Comparison of fault detection latency with CFCSS. The fault detection latency is not adversely affected.

In the next experiment, we explore the fault coverage provided by these techniques. We define fault coverage as the percentage of faults out of total injected faults that do not result in Silent Data Corruptions (SDCs). SDCs are the most harmful errors because the program silently corrupts data while the user thinks that application worked as expected. The faults classified in *HWDetects* imply that these symptoms can be used to trigger recovery [10, 38]. Each bar in Figure 12 shows the distribution of faults among different categories when the instrumented application runs with fault injections. The four bars are the fault distribution for CFCSS, CFCSS_ivl, ACS_w/o_calls_rets and ACS_w/_calls_rets and the average fault coverage for these techniques is 98.8%, 98.4%, 96.6% and 96.3%, respectively. All these techniques reduce the number of SDCs in comparison to unprotected application, but ACS without calls/rets protection has only 11% performance overhead in comparison to 75% performance overhead of CFCSS.

### 5.1 Fault Detection Latency

Another important metric with regard to fault detection techniques is the detection latency. Fault detection latency is directly related to the overhead of a recovery scheme. A longer latency implies that either the fault cannot be recovered or the recovery overhead would be high. Figure 13 shows the latency of ACS with respect to CFCSS. *WithIn2K* represents the number of faults detected in less than 2000 (2K) cycles of injections. Similarly, *WithIn5K*, *WithIn10K* and *WithIn100K* represents the number of fault detected within 5000, 10000 and 100000 cycles of injection, respectively. These categories are cumulative and faults classified under *WithIn5K* include all the faults detected with in 5K cycles, i.e., it subsumes the faults classified under *WithIn2K*. Similar rules apply for faults detected with in 10K and 100K cycles. The bars in the figure are normalized with respect to the number of faults detected in *WithIn2K*. For example, the *WithIn5K* bars represent the ratio of the number of faults detected with in 5K cycles and number of faults detected with in 2K cycles. In case of ACS, on average, *WithIn5K* contains 2% more than *WithIn2K*. Similarly, *WithIn10K* and *WithIn100K* contain only 3% and 5% more faults than *WithIn2K*. The same numbers for CFCSS are 0%, 1% and 1% for 5K, 10K and 100K cycles, respectively. Overall, ACS only increases the detection latency for at most 5% of the faults detected within 2K cycles.

### 5.2 Analysis of SDCs

In this subsection, we discuss some of the cases that escape the detection by CFCSS and ACS control flow methods and eventually result in silent data corruptions. LLVM IR supports the *switch* statements as the terminating instruction of BBs. When the code generation phase converts this switch statement to machine instructions, it is converted into multiple branches. Since these branches were not visible to our code instrumentation pass, these do not get protected by ACS or CFCSS. Some of the faults that affect such unprotected branches eventually cause SDCs. One way to handle these *switch* statements is to convert all the *switch* statements to *if-else* in the LLVM IR itself before running our code instrumentation pass. Another frequent case of SDCs is the faults that displace target address (i.e., faults in low order bits) only by few instructions usually result in SDCs. For example, we noticed that a fault in second bit of target address of a back edge caused only two extra instructions to be executed. Those two extra instructions happened to be immediate *mov* instructions and they just disturbed the value of two registers. Affected registers were written to memory and hence caused SDC. A similar problem also exists with CFCSS.

### 5.3 Data and Control Flow Protection

In this subsection, we present the results for combining a profile-based data flow [15] and our proposed control flow solution. Figure 14 shows the performance overhead on the primary vertical axis and fault coverage on the secondary vertical axis when a com-

bination of ACS and profile based data flow protection is used. SWDetects category in the fault outcome classification represents the number of faults detected by software (both data and control flow) and other category are same as previously mentioned. Control flow condition errors are handled by duplicating the computations for branch conditions. A combined solution incurs an average performance overhead of 47.4% and provides 96.5% fault coverage. The binary is 35% larger and overhead on dynamic instructions is 55.4%. SWIFT [32] is another solution that used data duplication.

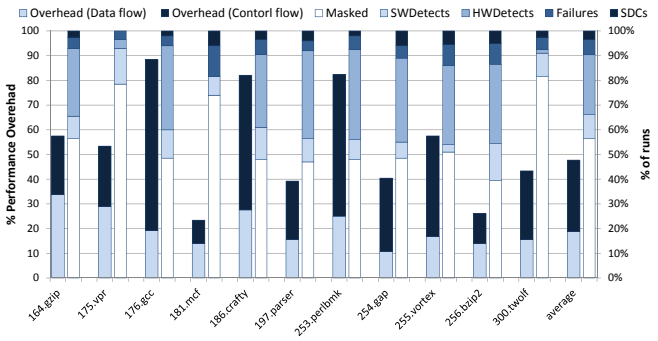



Figure 14: Performance overhead and fault coverage for complete data and control flow protection.

By leveraging the ideas from CFCSS, SWIFT also enhances control flow protection. In comparison to ACS with data duplication, SWIFT incurs an increase of: 2.3x for dynamic instructions, 2.3x for binary size and 1.53x for execution time over the same set of benchmarks as used in this work even though the performance overhead of SWIFT was measured on a aggressive server class workstation targeting a different ISA (IA64) than our evaluations (ARM). An IA64 system can take better advantage of instruction level parallelism introduced by duplication of instructions.

### 5.4 Discussion and Limitations

Similar to other signature based schemes [3, 29], ACS cannot detect faults in branch conditions. Though other schemes [13, 36] can detect errors in a branch condition if the error occurs after the branch condition is evaluated. This still misses the errors happening before condition evaluation and in variables used in evaluation of that condition. Corrupt branch conditions or other variables used to compute branch conditions can cause control flow condition errors. These errors in branch conditions can be handled by combining ACS with data flow protection based methods as described in Section 5.3. In this paper, we focus on the faults in branch targets and other variables used in computing branch targets.

In the presence of an error in the inserted checking code, the following scenarios can occur: 1) If the check evaluates to True, then the error in signature comparison branch will result in skipping the signature updates of next basic block, hence the error will be caught at the next check. 2) If the check is wrong (i.e., an error has already occurred), then considering that a transient fault is a rare event, a second error in this short span of time in signature comparison is probabilistically unlikely to occur.

In LLVM, the CFG is the basis of data flow analysis and many optimizations. To facilitate this data flow analysis, LLVM doesn't allow the address of a BB to be taken and then jump to it. Jumps to a location specified in a variable can only exist in the form of call instructions and for other control flow instructions target BBs are known at LLVM IR level. So at LLVM IR level, there is no special handling for *indirect branches* is required.

## 6. Related Work

Control flow protection is becoming an increasingly important concern for reliability researchers. Two particularly noteworthy pieces of software-only work in this area are CFCSS [29] and ECCA (Enhanced Control Flow Checking using Assertions) [3]. In our experimental results, we have compared our work with CFCSS in detail. ECCA assigns a unique prime identifier to each BB in the program and checks prime identifier at runtime using an assertion in every BB. The authors of [37] reported that ECCA incurs 150% memory overhead. Venkatasubramanian et.al [37] use parity in each BB to check for correct control flow. Control flow is checked by special variables inserted in each routine. The main difference with respect to these techniques and ACS lies in the fact that we raise the level of abstraction for checking and the signature update is simplified in each BB. Borin et al. [8] presented a control flow error detection technique where the signature checks are made in 1) every BB, 2) only in the BBs with back edges and BBs with return instructions, 3) only in BBs with return instructions and 4) only at the end of the application. This previous work reports 77% overhead for the case 1 and 37% for the case 3, in comparison to 11% overhead of ACS. Fault coverage data or detection latency for these different checking granularity is not reported in the paper. It is expected that delaying the checking to loop end points (blocks with back edges) and function ends (return blocks) will result in relatively more failures and program corruptions or will affect detection time. CEDA [36] is an assertion based scheme that assigns static signatures while minimize aliasing. The overhead of CEDA for common benchmarks is 27.1% in comparison to 11% of ACS. CEDA work also presents comparison with CFCSS and YACCA [13]. The performance overhead of CEDA reported in that paper is comparable to CFCSS for the chosen five benchmarks with a slightly better fault coverage. Since ACS has lower overhead than CFCSS, it will also have lower overhead than CEDA. The paper reports YACCA's overhead even larger than that of CFCSS and CEDA.

A comparison with SWIFT [32] is already described in Section 5.3. Other works such as CRAFT and PROFIT [33] improve upon the SWIFT solution by using additional hardware structures and architectural vulnerability factor (AVF) analysis [28]. Our goal in this work is to make the control flow protection practical for commodity embedded systems by reducing the performance overhead. Our experimental results demonstrate that this can be achieved at significantly less performance overhead than these previously proposed techniques.

Symptom detection based solutions rely on anomalous microarchitectural behavior to detect soft errors. A light-weight approach for detecting soft errors, ReStore [38], analyzes symptoms including memory exceptions, branch mispredicts, and cache misses. mSWAT [18] presented a solution which detects anomalous software behavior to provide a reliable system. It requires special simple hardware detectors to detect faults. These techniques are orthogonal to ACS, as they rely on specialized hardware. If available, they can be leveraged along with ACS to increase the number of faults detected under HWDetects category.

A category of previous works related to control flow protection are watchdog processor based solutions [22]. The general idea of these techniques is to have a watchdog processor, along side the main processor, that monitors and checks the program executing on the main processor. These solutions rely on the availability of watchdog processor and in some cases even propose specific changes to the watchdog processors. A variety of watchdog based solutions [21, 25, 36] are proposed in literature by modifying some aspect (e.g., changing the type of signatures) of the technique. Some recent solutions also suggest the idea of distributed checking in the core for various components. Argus [24], for example, relies on a series of hardware checker units to perform online invariant checking to ensure correct application execution (data flow as well as control flow). Argus achieves very low overhead by adding extra hardware. In comparison to these techniques, ACS targets COTS components and does not require any hardware changes.

An interesting approach to soft error reliability is using Redundant Multithreading (RMT). AR-SMT [34] introduced the idea of RMT on SMT cores; The work is done by a leading thread, and the

trailing thread checks for the correctness. Subsequent works [14, 31] in this category have tried to reduce the overhead due to RMT. All these techniques come with the overhead of running an extra thread which executes a skeleton of the original program.

Another compiler assisted solution for control flow checking uses extra hardware to minimize the overhead [19]. It requires compiler, as well as hardware changes. Ours is a software-only approach to produce protected programs.

There is a large body of related work in Control Flow Integrity (CFI) [1] for computer security against external software attacks. CFI works by making sure that all the control transfer occur as determined by the static CFG. The failure model targeted by CFI schemes is very different from soft errors failure mode. In CFI, constant destinations (direct branches ) are statically verified and while computed (dynamic branches) are verified for correct destination by instrumenting the code. Soft errors can affect the direct as well as indirect branches and hence CFI, as is, is not directly applicable for soft errors. Though direct branches can also be protected in a manner similar to dynamic branches, but the already high overhead (20%-60% for dynamic branches only) would become prohibitive.

Path profiling [4] finds the execution count of a path in a Directed Acyclic Graph (DAG). It is a related problem to our work and gives an unique number for each path in a DAG. However, we want to have a balanced path length along with information about edges in the path to insert balancing increments. This can not be obtained with path profiling. Moreover, usually profiling is created with training inputs but later the program might be executed with a different set of inputs. In ACS, we need the correct path length with the current inputs a program is executing. Therefore, the data produced by off-line profiling can not be used in ACS.

## 7. Conclusions

The ever increasing desire to create powerful and efficient microprocessors, with each successive new generation, has led to the use of increasingly smaller transistors into these devices. Aggressive scaling makes transistor devices more susceptible to transient faults. To tackle the problem of control flow protection at minimal performance overhead, we have proposed Abstract Control flow Signatures (ACS). ACS achieves its efficiency by working at a coarse-grain level than the previously proposed signature based techniques and also by simplifying signature updates in each basic block. ACS reduces performance overhead, on average, from 75% down to 11% while maintaining the similar level of fault coverage in comparison to a previously proposed approach (CFCSS [29]).

## 8. Acknowledgements

The authors would like to thank the shepherd and the anonymous reviewers for their constructive comments and suggestions for improving this work. This research is supported by the National Science Foundation under grant CCF-0916689 and STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

## References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.*, 13(1): 4:1–4:40, Nov. 2009. ISSN 1094-9224.

[2] A. Aho, M. Lam, R. Sethi, and J. Ullman. *Compilers: principles, techniques, and tools*, volume 1009. Pearson/Addison Wesley, 2007.

[3] Z. Alkhalifa, V. Nair, N. Krishnamurthy, and J. Abraham. Design and evaluation of system-level checks for on-line control flow error detection. *TDPS*, jun 1999.

[4] T. Ball and J. R. Larus. Efficient path profiling. In *ACM/IEEE Micro*, 1996.

[5] W. Bartlett and L. Spainhower. Commercial fault tolerance: A tale of two systems. *In TDSC*, pages 87–96, 2004.

[6] D. Bernick, B. Bruckert, P. D. Vigna, D. Garcia, R. Jardine, J. Klecka, and J. Smullen. Nonstop advanced architecture. In *DSN*, pages 12–21, June 2005.

[7] N. Binkert et al. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2), Aug. 2011.

[8] E. Borin, C. Wang, Y. Wu, and G. Araujo. Software-based transparent and comprehensive control-flow error detection. In *CGO*, 2006.

[9] M. Chu, K. Fan, and S. Mahlke. Region-based hierarchical operation partitioning for multicluster processors. In *PLDI*, pages 300–311, June 2003.

[10] S. Feng, S. Gupta, A. Ansari, and S. Mahlke. Shoestring: Probabilistic soft-error reliability on the cheap. In *ASPLOS*, Mar. 2010.

[11] S. Feng, S. Gupta, A. Ansari, S. A. Mahlke, and D. I. August. Encore: low-cost, fine-grained transient fault recovery. In *MICRO*, pages 398–409, 2011.

[12] B. T. Gold, J. C. Smolens, B. Falsafi, and J. C. Hoe. The granularity of soft-error containment in shared memory multiprocessors. *IEEE Workshop on SELSE*, 2006.

[13] O. Goloubeva, M. Rebaudengo, M. Sonza Reorda, and M. Violante. Soft-error detection using control flow assertions. In *DFT*, pages 581 – 588, nov. 2003.

[14] M. Gomaa and T. Vijaykumar. Opportunistic transient-fault detection. In *ISCA*, pages 172–183, June 2005.

[15] D. S. Khudia, G. Wright, and S. Mahlke. Efficient soft error protection for commodity embedded microprocessors using profile information. In *LCTES*, pages 99–108, New York, NY, USA, 2012. ACM.

[16] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–86, 2004.

[17] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert. Statistical fault injection: quantified error and confidence. In *DATE*, pages 502–506, 2009.

[18] M. Li, M. Pradeep, R. S. Sahoo, S. Adve, V. Adve, and Y. Y. Zhou. Swat: An error resilient system. In *IEEE Workshop on SELSE*, pages 8–13, 2008.

[19] X. Li and J.-L. Gaudiot. A compiler-assisted on-chip assigned-signature control flow checking. In *Advances in Computer Systems Architecture*, volume 3189 of *LNCS*, pages 554–567. Springer Berlin, 2004.

[20] X. Li and D. Yeung. Application-level correctness and its impact on fault tolerance. In *HPCA*, pages 181–192, Feb. 2007.

[21] D. Lu. Watchdog processors and structural integrity checking. *IEEE Transactions on Computers*, C-31(7):681 –685, july 1982.

[22] A. Mahmood and E. J. McCluskey. Concurrent error detection using watchdog processors-a survey. *IEEE Trans. Comput.*, 37(2):160–174, Feb. 1988.

[23] T. May and M. Woods. Alpha-particle-induced soft errors in dynamic memories. *IEEE Transactions on Electron Devices*, 26(1):2–9, Jan. 1979.

[24] A. Meixner, M. Bauer, and D. Sorin. Argus: Low-cost, comprehensive error detection in simple cores. *IEEE Micro*, 28(1):52–59, 2008.

[25] T. Michel, R. Leveugle, and G. Saucier. A new approach to control flow checking without program modification. In *FTC*, pages 334 –341, jun 1991.

[26] S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufmann Publishers, 1997.

[27] S. Mukherjee. *Architecture Design for Soft Errors*. Morgan Kaufmann, 2008.

[28] S. S. Mukherjee, C. Weaver, J. Emer, S. Reinhardt, and T. Austin. A systematic methodology to compute the architectural vulnerability factors for a high performance microprocessor. In *MICRO*, pages 29–42, Dec. 2003.

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

[30] N. Oh, P. Shirvani, and E. McCluskey. Error detection by duplicated instructions in super-scalar processors. *Reliability, IEEE Transactions on*, 51(1):63–75, 2002.

[31] S. K. Reinhardt and S. S. Mukherjee. Transient fault detection via simulataneous multithreading. In *Proc. of the 27th ISCA*, pages 25–36, June 2000.

[32] G. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. SWIFT: Software implemented fault tolerance. In *CGO*, pages 243–254, 2005.

[33] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, D. I. August, and S. S. Mukherjee. Software-controlled fault tolerance. *ACM TACO*, 2(4):366–396, 2005.

[34] E. Rotenberg. AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. In *International Symposium on Fault Tolerant Computing*, 1999.

[35] P. Shivakumar, M. Kistler, S. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *DSN*, pages 389–398, June 2002.

[36] R. Vemu and J. Abraham. Ceda: Control-flow error detection using assertions. *IEEE Transactions on Computers*, 60(9):1233 –1245, sept. 2011.

[37] R. Venkatasubramanian, J. Hayes, and B. Murray. Low-cost on-line fault detection using control flow assertions. In *IOLTS 2003.*, july 2003.

[38] N. J. Wang and S. J. Patel. ReStore: Symptom-based soft error detection in microprocessors. *In TDSC*, 3(3):188–201, June 2006.

[39] N. J. Wang, J. Quek, T. M. Rafacz, and S. J. Patel. Characterizing the Effects of Transient Faults on a High-Performance Processor Pipeline. In *DSN*, June 2004.

[40] J. F. Ziegler and H. Puchner. *SER-History, Trends, and Challenges: A Guide for Designing with Memory ICs*. Cypress Semiconductor Corp., 2004.