Low-cost Software Control-Flow Error Recovery

Ghazaleh Nazarian*, Razvan Nane* and Georgi N. Gaydadjiev‡*
* Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology, Mekelweg 4, 2628 CD Delft, The Netherlands
{g.nazarian, r.nane}@tudelft.nl
‡ Dept. of Computer Science and Engineering, Chalmers University of Technology, Rannvagen 6, Goteburg, Sweden
georgig@chalmers.se

***Abstract*—In modern safety-critical embedded systems reliability and performance are two important criteria. In many systems based on off-the-shelf processors software implemented error recovery is the only option to improve the reliability of the system. However, software methods typically introduce large performance overheads. Another important factor in error recovery schemes is the recovery time, especially in systems with real-time requirements. A key observation that helps improve software recovery methods is that only a defined number of locations in the program are susceptible to errors. In this paper we propose a fast software recovery scheme that instruments the program only at locations vulnerable to control-flow errors. We use a systematic bit-flip analysis to identify the exact locations susceptible to control-flow errors in a given program. This helps us to instrument the code with minimal overheads, while maintaining high-level of correct-ability and low recovery times. Our experiments show that using the result of our bit-flip analysis and limiting the code instrumentation to only the susceptible locations improves the efficiency by a factor of 80 when compared to the latest control-flow error recovery methods.**

I. Introduction

In modern processors, along with technology down-scaling and the reduction of the operating voltages, the probability that phenomena such as radiation or crosstalk change the state of a transistor causing a transient fault becomes increasingly higher. Therefore, reliability has emerged as a key factor in embedded processor design. Different reliability optimization schemes are proposed both in hardware and software. Depending on application criticality and requirements, either a scheme in hardware with special circuit checkers or a method in software may be used. For example, safety-critical applications on servers use specialized protecting hardware against transient faults. Such computers are not bundled to a tight power consumption and area. However, applications on embedded processors with limited power and area budgets can not afford extended hardware for reliability protection. Moreover, in off-the-shelf processors the hardware can not be modified and the only remaining solution to protect against hardware transient faults is software optimizations. The target of our work in this paper are such processors for embedded reliable applications.

Transient hardware faults cause data or Control-Flow Errors (CFE) at run time. Data-errors cause an erroneous value in registers or memory and CFEs cause an erroneous execution flow. Since the effect of data-errors and CFEs are different, optimization techniques to protect the application against each of the error types are also different. Software methods instrument the application with extra code (assertions) to detect and recover from data or control-flow errors. CFE detection methods employ signature monitoring inside basic blocks[1] to detect illegal execution flow. Conventional CFE recovery methods use checkpoints at the beginning of the basic blocks in order to restart the execution in case a CFE is detected. Due to the high overhead of using checkpoints for all basic block, some methods select fewer basic blocks to add checkpoints. However, decreasing the number of checkpoints increases the recovery time from the moment an error is detected up to the time the execution is fully recovered. In applications with real-time requirements, the recovery time should be as low as possible. The ideal case for CFE recovery is to know which basic blocks are the source of the error and add checkpoints only to those blocks. Previous research [1] has shown that a significant number of basic blocks are not susceptible to CFEs. Assertions and checkpoints that are added to protect the non-susceptible blocks are unnecessary and, without improving reliability, increase performance overhead.

In this paper, we propose a new CFE recovery method which adds checkpoints only to basic blocks that are susceptible to CFEs. We use the bit-flip analysis scheme proposed in [1]. This framework analyzes the impact of single bit-flips on the control-flow misbehavior and identifies all basic blocks that are the potential destinations of faulty transitions. These basic blocks are the susceptible destinations of erroneous branches caused by a CFE. In this work we extend the previously proposed bit-flip analysis in order to identify the basic blocks where CFEs are initiated from. These blocks are the susceptible sources where the erroneous branch can occur. The result of our extended analysis scheme gives the ordered pairs of susceptible source and destination blocks. Using this information, we instrument the code with necessary assertions and checkpoints to protect susceptible blocks.

The main contributions of this work are:

- A novel fast control-flow error recovery method;
- Framework for identifying the potential source basic blocks where an erroneous branch may stem from;
- Low-cost and effective check-pointing scheme by placing the checkpoints in the identified susceptible source blocks;

[1] branch-free sections of the program

978-1-4673-8035-5/15 $31.00 © 2015 IEEE
DOI 10.1109/DSD.2015.92

- Efficiency metric to rank control-flow error recovery;
- Efficient recovery scheme with short recovery time of 28 cycles and low performance overhead compared to state-of-the-art methods.

The rest of the paper is organized as follows: Next, the motivation behind this work and the required background information are given, including the target fault model and the existing software methods for control-flow error recovery. The detailed explanation over bit-flip analysis and our checkpointing scheme is given in section III. Section IV describes the experimental setup and the results analysis. Finally we present the conclusions and future work.

II. Background information

A. Fault Model

The target fault model in our work is single bit transitions, which are due to events such as crosstalk or radiation. As investigated in [2], multiple bit transitions cause the same misbehavior as single bit flips. Therefore, in this work we consider only single bit transitions. Single bit transitions may lead to Data-Error or CFE. Data-errors cause an erroneous value in registers or memory. CFEs cause an erroneous execution flow. It should be noted that an error in the condition of a conditional-branch is a data-error and is detectable with specific detection methods for this fault category. CFEs may occur due to three main reasons: 1) branch creation; 2) branch deletion and 3) error in the branch destination. Branch creation may occur if a non-branch instruction converts to a branch and branch deletion may happen if a branch instruction converts to a non-branch instruction. The probability of transforming a non-branch opcode to a branch and vice versa due to a single bit transformation is extremely low and depends on the opcode coding of the instruction set architecture. It is important to note that branch creation may also happen due to bit-flips in the program counter. However, the probability of CFE occurrence due to single bit-flip in the program counter is relatively low as it is a small circuitry compared to the rest of the processor components. For these reasons, in the recent works the main cause of CFEs is considered to be erroneous branch instruction destinations [3]. In this work, also, we target CFEs caused by faulty bit-flips in branch instructions destinations. Since the first two categories of CFE have very low probabilities, adding high-overhead software assertions for detecting such errors in embedded systems with high-performance requirements is not efficient.

B. Related Works

Proposed methods for detection and recovery of transient faults exploit some form of redundancy as redundant hardware, redundant process and redundant thread or some additional instructions in the executed program to detect the faults and checkpoints to roll back the execution and recover from the detected fault. Hardware redundancy, either if hardware is replicated [4] [5] or extended with circuit checkers [6] has the drawback of being costly and not being applicable in many off-the-shelf-processors. Methods using redundant threads [7] or process-level redundancy [8] for reliability optimization also need parallel hardware resources which may not be available in older processors used in current systems. Several software optimization methods to improve reliability instrument programs with additional code to check run-time program execution. Since the effect of data-errors and CFEs are different, the detection mechanisms proposed for each error type are also different and target only one of the two error types. Data error detection methods, such as EDDI [9], add a duplicated version of instructions and a corresponding compare checking the consistency between the two versions. CFE detection methods have a unique signature associated to each basic block. At runtime *set* assertions update *runtime Signature* (S) to the current basic block signature and *test* assertions check the correctness of the S content to validate control-flow correctness.

Several software optimization methods for CFE detection exist: CCA [10], ECCA [11], CFCSS [12], YACCA [13], CEDA [14], ACFC [15], Abstract Control Signatures (ACS) [3] and SWIFT [16]. SWIFT is a hybrid method combining CFCSS for CFE and EDDI for data error detection. All the mentioned methods add *set* assertions to all basic blocks to update the runtime signature along the control-flow path. However, depending on the category of the detection method *test* assertions are added in predefined locations of the program. CFE detection methods can be divided into two main categories: *path-asserting* and *predecessor/successor-asserting* methods. A path-asserting method adds *test* in one B-block per control-flow path[2] to assert correct path execution. Predecessor/Successor-asserting methods add *test*s in all B-blocks to check if the previous (or next) B-block in the execution flow is the correct predecessor (or successor). CFCSS, ECCA, CEDA, YACCA and CCA represent predecessor/successor-asserting methods with high fault coverage and high overhead. ACFC and ACS add one *test* assertion for group of basic blocks and are categorized as path-based methods. Predecessor/successor assertions are also categorized into two groups: 1) methods with incremental signatures update; 2) methods with local signature updates. At incremental signature updating, the global signature content at each basic block is dependent on all set assertions in the predecessor basic blocks along the execution path. Sample methods for incremental signature update are CEDA, CFCSS and YACCA. On the other hand, local signature updates set the signature at the current basic block independent of global signature content. ECCA and CCA are examples of methods with local signature update. The shortcomings of local signature updates are high overheads and low fault-detection capability for basic blocks with multiple predecessors. A recently proposed detection method with local signature update is FCFC [1] which does not have the downside of not detecting errors in basic blocks with multiple predecessor.

In [17], the authors investigate that the majority of transient faults can either be ignored (because they do not ultimately propagate to user-visible corruptions at the application level)

[2]a group of B-blocks executed in an uninterrupted sequence

or are easily masked by lightweight symptom-based detection. They use compiler analysis to find high-value portions of the application code that are both susceptible to soft errors and statistically unlikely to be covered by the timely appearance of symptoms. They protect these portions of the code with instruction duplication. Their solution offers optimization for the coverage and performance trade-off for detecting data errors. Another work that analyzes the effect of single-bit flips on the CFEs and finds the potential faulty destinations of erroneous branches was proposed in [1]. In this work, the instrumentation for CFE detection is limited to the identified susceptible basic blocks.

Recovery methods use a detection mechanism to detect first the fault and then use checkpoints to roll back in order to recover the execution. Traditional methods place checkpoints at critical points of the program [18]. However checkpoints impose a high performance overhead. Previous works have tried to find the optimum locations to add checkpoints in order to avoid excessive performance overhead while limiting the recovery time [19] [20]. These studies use a mathematical model to compute the optimal checkpointing intervals. In the proposed solutions the performance is traded off for the recovery time. A recently proposed lightweight checkpointing method, uses static analysis of applications to find code regions without Write-After-Read dependencies, the so called idempotent regions [21]. These lightweight checkpoints save only the registers state at the beginning of the idempotent regions. The starting address of the idempotent region is saved to roll-back the execution in case an error is detected. This lightweight checkpointing imposes low runtime performance overhead. However, the recovery time from the moment that an error is detected until the moment that the execution is recovered to the location where the error has occurred is dependent on the size of idempotent regions. Larger idempotent regions result in low performance overhead but higher recovery time. One of the recently proposed methods is ACCE [22]. It does not use checkpoints due to the high performance cost. The authors propose a roll-back mechanism using a global error-handler and recovery routines for each function in the compilation unit. To the best of our knowledge ACCE provides the fastest recovery mechanism among the previously proposed recovery techniques. The global error-handler is responsible to determine from which function the error has initiated. After it is determined which function is the source of the error, the recovery routine of that function finds the basic block in which the error has occurred and the execution is rolled-back to the start of that basic block. The recovery time in this mechanism is equal to the total number of cycles for executing the global error-hander and the corresponding recovery-routine. One shortcoming of ACCE is that the data is not restored after the roll back. In order to restore data after roll-back the authors suggest to use data duplication mechanism. But the cost of data duplication and comparison is not lower than using checkpoints.

C. Motivation

A significant percentage of transient faults causing CFEs lead an execution outside the program boundary scope. Since CFE detection and recovery methods instrument the program within its memory boundary, errors outside this boundary are not recoverable with currently existing software program instrumentation techniques. Recovery from such errors is only possible with the help of the operating system to detect the error as segmentation fault and re-executing the program from the beginning. However, nowadays there exist many tiny embedded processors without an operating system. CFEs which lead the execution into an erroneous destination inside the program boundary can be detected and recovered by software methods that instrument the program. Current CFE detection and recovery methods add assertions to all basic blocks of the program. Recovery methods that use checkpoints to save the processor's state at specific points of the program, trade-off the recovery time for performance. Placing less checkpoints and dividing the program in larger sections reduces the high cost of checkpoints, but increases the recovery time in case an error is detected. An important observation by one of the previous works is that many of the program basic blocks are not realistic destinations of the erroneous branches caused by single bit-flips in the destination of control instructions [1]. We use this observation to implement an effective CFE detection and recovery scheme with low performance overheads and low recovery time. In this scheme, our main motivation is to limit the CFE detection and recovery instrumentation only to susceptible basic blocks. The checkpoints, which are the main cause of high performance overheads in CFE recovery schemes, should only protect basic blocks that are potential destinations of CFEs.

D. Assessing software error recovery methods

The three crucial factors that a software recovery technique should have are correct-ability, low recovery time and high performance. Software recovery methods instrument the program with assertions and checkpoints. High number of assertions and checkpoints located in short intervals improve the correct-ability and the recovery time of the method. However, it has a negative impact on the performance. Therefore, in order to asses a software recovery technique the three of these metrics should be considered at the same time. For this reason we introduce a new metric to quickly assess software recovery methods for a given workload. We define Recovery Efficiency Factor (REF) that depends on performance-overhead[3], the number of execution cycles from the moment an error is detected until the moment it is recovered and the percentage of correct outputs among a defined number of program samples with injected faults:

$$REF = \frac{correct.output.ratio}{Performance.overhead * recovery.cycle.count}$$

REF is a suitable metric for evaluating reliability optimization techniques in real-time and high-performance embedded

[3]the percentage of additional clock-cycles in the instrumented program

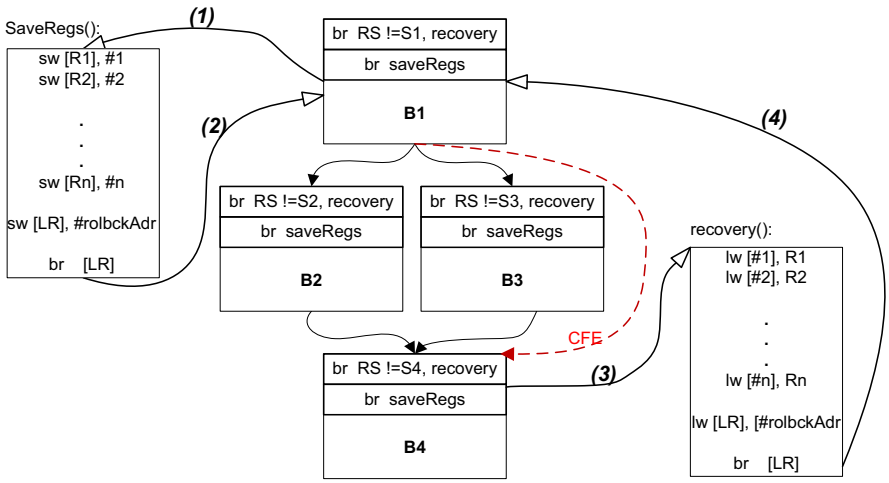


Fig. 1: Recovery flow

systems. In order to explain how REF distinguishes between a weak and a strong recovery method, we use an example of two methods with different recovery capabilities. An example of a poor recovery method causes 100% performance overhead while the correct output ratio is only 1% and the recovery cycle count is 100 cycles. The calculated REF for such a method is $(1\%)/(100\% * 100) = 0.0001$. On the other hand, an example of good recovery method has 1% performance overhead with 100% correct output ration and 10 cycles of recovery latency. The calculated REF for this method would be $(100\%)/(1\% * 10) = 10$. These examples that are opposite cases show that for real-time and high-performance systems a weak recovery method has a lower REF than a good recovery method.

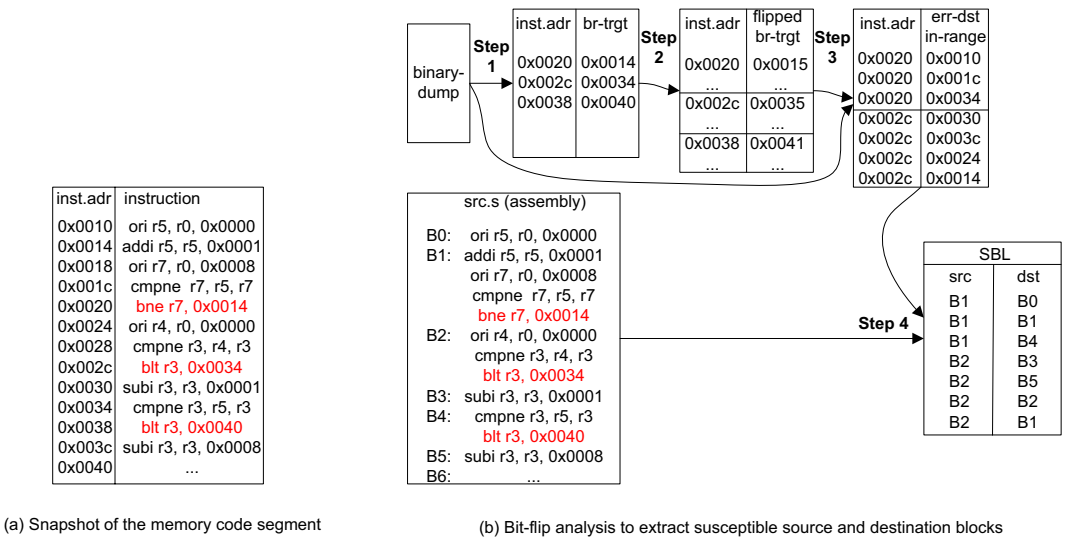
III. Overview of Fast Recovery with Workload Specific Checkpoints

In our proposed recovery method, at compile time instructions are added to the program in order to detect and recover from the error. Complete recovery from CFEs is accomplished using Workload Specific Checkpoints (WSC). The error detecting instructions are executed during the normal flow of the program. However, the instructions added for error recovery are executed only when an error is detected and the execution control is transferred to the special function for error recovery. The goal of our scheme is to have an efficient recovery scheme with low recovery time, low performance overhead and high fault coverage. To have the lowest possible recovery time, we need to provide an arrangement that the recovery process is done at the smallest possible granularity, which is basic block level. Moreover, to have low performance overhead, we need to add error detecting and recovery instructions only to the necessary locations of the program that are the potential basic blocks where CFEs can occur. It is important to note that not all CFE detection methods have the flexibility to instrument only a selected number of basic blocks [1]. Therefore, in order to be able to fulfill the second requirement, we need to use a flexible error detection method such as FCFC [1]. FCFC is an efficient CFE detecting method with the flexibility to assert the correct execution flow by adding instructions only to the required subset of basic blocks. In what follows, the three important aspects of the proposed recovery method are explained.

A. Fast Recovery Scheme

In order to decrease the amount of time between the moment a CFE occurs and the moment the program is recovered, we implement the detection and recovery processes at basic block level. FCFC fault detecting instructions are able to detect CFEs immediately after occurrence in the faulty target block. Moreover, our proposed recovery method is arranged in a way that the execution control will be transferred right to the basic block that was the source where the CFE occurred.

Figure 1 shows how the recovery scheme transfers the control immediately to the block before CFE occurrence. As depicted in the figure, two statements are added in the beginning of each basic block to provide recovery. The first statement in the basic blocks (*br RS!=Sig, recovery*) is the test assertion of CFE detection scheme. In this figure for the matter of readability the statements related to set assertions of detection scheme are not shown. In case the test assertion finds a mismatch between the runtime signature content and the signature value it should have at the current basic block the recovery function is called. The second statement is a simple function call that calls saveRegs function. Calling saveRegs function will transfer the execution to this function. Moreover, it saves the return address (which is the original start address of the basic blocks before adding the recovery statements) into the Linked-Register (LR). In the figure a CFE occurrence is depicted by the dashed edge from B1 to B4. Also the steps of code executions that leads to recovery from the depicted CFE are shown. The first step, before the CFE occurs in the beginning of the source basic block where CFE will stem from, the control transfers to saveRegs function. This function



(a) Snapshot of the memory code segment

(b) Bit-flip analysis to extract susceptible source and destination blocks

Fig. 2: Bit-flip analysis scheme illustration

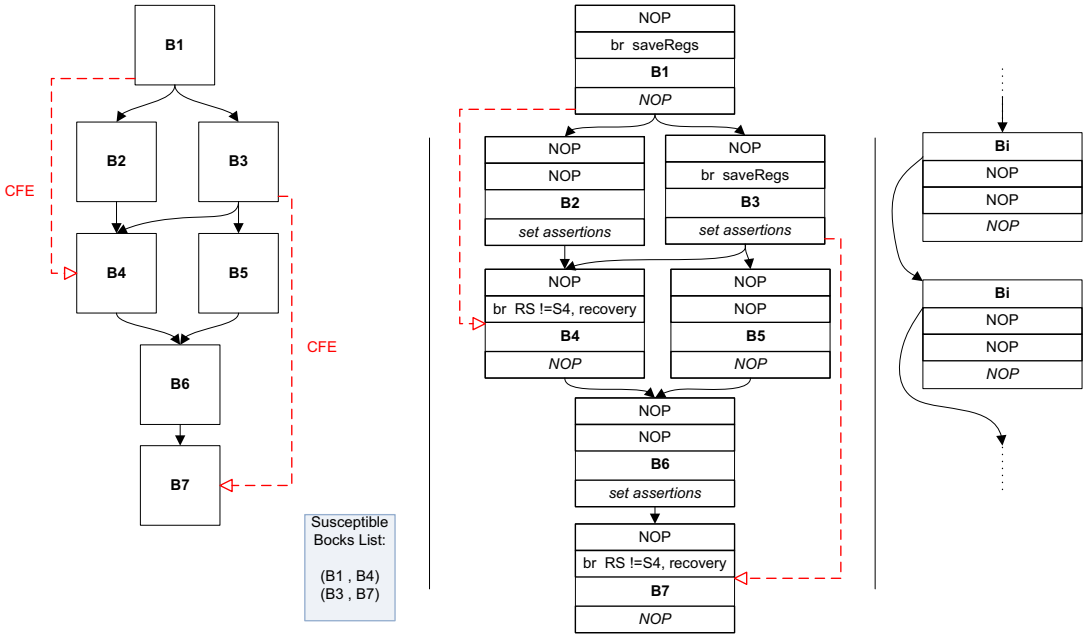
acts as a checkpoint and saves the contents of all registers in the register file. Moreover, it saves the content of the linked register that holds the start address of the basic block. This address is the address where the execution should roll-back to, when the error is detected. In the second step, the execution is transferred back to B1. After CFE occurs, the test statement in B4 detects the error and calls the recovery function, shown as step 3. The recovery function restores the saved contents of the register file in the previous checkpoint (which is at B1) and loads the roll-back address (which is the start address of B1) into the linked register. Finally, in the fourth step, the execution rolls back to the beginning of B1.

It is important to notice that this recovery scheme has minimal recovery time, since the extra code that is added to transfer the execution control back to the point before CFE occurrence is minimal. The recovery steps in a similar recovery method (ACCE) [22] consist of four function calls that have in total higher number of recovery code compared to our scheme. Compared to ACCE, which is also at basic block level, the added amount of recovery code in our method is less than half. Another disadvantage of ACCE is that it does not support full recovery from CFEs. This is due to the fact that ACCE does not save and restore the modified data in the register file due to re-execution of the rolled-back basic block. In order to provide data integrity and full recovery from CFEs, ACCED is proposed where the data values are protected by having a duplicated version and comparison between the two versions. This is a very expensive solution for data restoration.

B. Efficient Checkpoints at Identified Susceptible Blocks

Instrumenting all basic blocks with checkpoints as described above is too costly. Fortunately, previous studies about the effect of single-bit transitions on the control-flow mis-behavior has shown that not all basic blocks are susceptible to CFEs [1]. In other words, only a number of basic blocks in the CFG are susceptible to CFEs and need to be protected by assertions and checkpoints.

In order to minimize the imposed performance overhead by checkpoints, we use the bit-flip analysis framework proposed in [1]. This framework gets the program binary dump and assembly as input and generates a list containing all susceptible blocks that are the potential destinations of CFEs due to single-bit transitions. In order to use this information to limit the number of checkpoints, we also need to know the susceptible source basic blocks that the CFEs can stem from. We have easily extended the framework to add this information to the generated list. Figure 2 shows a schematic view of the extended framework to extract susceptible source and destination blocks. An example snapshot of the memory code segment is depicted in figure 2.a and the steps for extracting the list of susceptible blocks for this part of the code segment is illustrated in figure 2.b. At the first step, all branch targets (br-trgt) and the branch instruction addresses (inst.adr) are extracted. In the second step a set of XORs with MASKs, generate flipped branch target addresses. For instance "addr XOR 0001" flips the first bit of the target addresses and generates a realistic erroneous address. The resulting flipped addresses are saved in flipped-br-trgt. In the figure the first bit flipped-target is shown. In the third step, a simple script compares each of the erroneous addresses in flipped-br-trgt file to the extracted instruction addresses within the program scope. The result of the comparison at this step is a list of potential erroneous target addresses (susceptible to be the target of CFEs) within the program scope and is saved into err-dstin-range file. Finally in the fourth step, the corresponding source and destination basic blocks of the susceptible target addresses (in err-trgtsin-range file) are extracted. To extract the susceptible basic blocks, we compare the susceptible instructions offset to the corresponding code section in the assembly. Having basic block labels in the assembly file, the



(a) CFG sample with potential CFEs and the generated susceptible block list using bit-flip analysis framework

(b) Instrumentation for CFE recovery only to protect the susceptible blocks

(c) A non-susceptible block with rescheduled NOPs

Fig. 3: Instrumentation and checkpoints in susceptible blocks

susceptible basic block labels are easily extracted and saved in the SBL file.

Figure 3.a shows a sample CFG, corresponding to another example code, with potential erroneous CFE edges and the output list of the bit-flip analysis framework containing pairs of susceptible source and susceptible destination blocks. In this sample execution flow graph, the potential CFEs are depicted by the dashed edges from B1 to B4 and from B3 to B7. Our framework identifies the potential destinations of the erroneous branches due to single bit-flip transitions in the branch instructions targets and generates the susceptible block list. The result of the bit-flip analysis is the list containing two pairs of susceptible blocks as depicted in Figure 3.a. The first elements are the susceptible source blocks, B1 and B3, and the second elements in the block pairs are susceptible destinations, B4 and B7.

Figure 3.b depicts the same sample CFG, which is instrumented with CFE detecting assertions and checkpoints only in necessary locations using the susceptible blocks list information. Only the susceptible source blocks need to have a call to the saveRegs function to checkpoint the register file contents. In the sample graph only B1 and B3, which are susceptible source blocks, have the call to saveRegs function. Accordingly, only susceptible destination blocks need to have test assertions to check if the basic block is reached through a valid control-flow or due to a CFE occurrence. In the example graph only B4 and B7 contain *"br RS!=Sig, recovery"* statement, which is the test assertion. The predecessors of susceptible destination blocks are the only blocks that should have set assertions to update the runtime signature to a valid signature. These blocks in the sample CFG are B2, B3 and B6.

Assertions and checkpoints in non-susceptible blocks are replaced with NOP instructions. In order to impact the performance, we reschedule the replaced NOPs after the branch instruction at the end of the basic block. As a result, at runtime the replaced NOP instructions are not executed. Consequently, the recovery instrumentations does not impose any performance overhead in non-susceptible blocks.

IV. Experimental Setup and Results

To investigate the proposed recovery scheme, we compare it to a recent state-of-the-art work to show its effectiveness in removing unnecessary assertions and checkpoints in non-susceptible blocks. We have implemented and optimized a compiler using CoSy compiler development framework [23]. The generated compiler targets a basic, 32-bit, five-stage, in-order RISC processor. This processor is the template processor available in Synopsys Processor Designer simulator. It has no advanced micro-architectural features but has similar load/store-based ISA as any ARM processor. The only significant difference is the higher number of registers in the ARM case. As a result in such processors the additional instructions used for detection and recovery will cause lower register pressure. Therefore, the overhead of our instrumentations in a standard processor (such as ARMv7m) is expected to be lower than presented here. We have implemented compiler passes for our proposed recovery method and ACCE. We use a representative set of workloads from Mibench [24]. For each workload three different binaries are generated; the

Workloads	ACCE			Recovery with WSC		
	correct output	not recoverable errors		correct output	not recoverable errors	
		wrong output	out of program boundary		wrong output	out of program boundary
basicmath	90	125	785	259	303	434
qsort	83	47	871	376	94	531
pbmsrch	67	84	850	333	159	509
sha	58	58	885	318	218	465
dijkstra	87	28	886	409	124	468
CRC	76	32	893	313	102	586
Average	76.83	62.33	861.66	334.66	166.66	498.83

TABLE I: Categorization of programs output in 1000 program runs with random control-flow errors

original binary without any optimization, binaries compiled with ACCE optimization pass, and binaries compiled with our recovery pass. To get the error coverage of each optimization scheme we inject CFEs (by flipping single bits in the branch instruction operands) into the respective binary and inspect the number of detected errors. In what follows, first we explain our error injection framework and second we analyze the results.

A. Realistic Fault Injection

Error injection: We have implemented a runtime error-injection mechanism in the Synopsys Processor Designer simulator. A special instruction is designed and included in the processor instruction set architecture to inject an error at a random execution cycle. The error-injector instruction is added in the beginning of the program-under-test. A random value, generated by *random* linux command, is given as the operand of the error-injector instruction. The random value determines the number of execution cycles that should pass before the occurrence of the error. A dedicated flag for error injection is set after the given random number of cycles. When this flag is set, the first branch instruction in the execution path will be corrupted and hence CFE initiated. The latter is achieved by flipping a single random bit of the chosen branch instruction operand. Changing a single bit at a time simulates a realistic error injection environment as single bit flips are the most probable faults in real systems [2].

B. Results

Table I shows the result of fault injection for one thousand runs of the Mibench workloads instrumented with ACCE and our recovery scheme. The selected set of Mibench workloads is comprehensive to exemplify the advantages and the limitations of both methods. In the table the programs behavior to the injected CFE is categorized into three columns: correct output, wrong output and execution out of program boundary. The errors that are not recoverable are the ones that cause wrong output and the ones that lead the execution outside the program boundary. In one thousand execution runs of the workloads with control-flow error, ACCE recovers on average 76.83 of the cases, while the average recovery number of our scheme is 334.66. The main reason for the higher recovery number of our scheme compared to ACCE is due to the fact that in our scheme register file content is saved at the checkpoints in only the necessary blocks and restored when needed. In [22] the number of faults causing wrong output for ACCE is reported, but unfortunately the number of faults

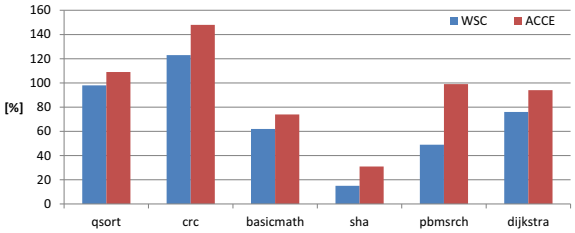


Fig. 4: WSC and ACCE performance overheads

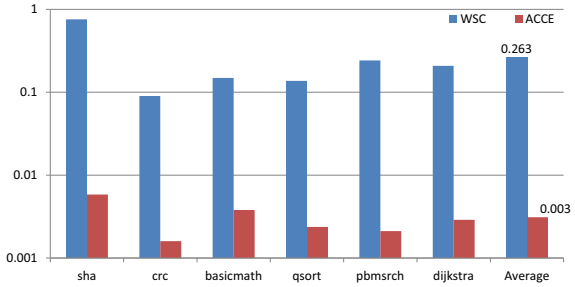


Fig. 5: REF factors of our method and ACCE

causing out-of-boundary execution is not reported explicitly. Our implemented ACCE results, compared to our recovery, show that a higher number of injected faults lead the execution out of the program boundary. The two reasons for the higher number of such executions is that, first, in this work we target branch-destination corruption fault model that has a high probability to lead the execution out of the program scope. Second, ACCE recovery code introduces many new branch instructions that become themselves the target of the injected faults and cause out of boundary execution.

The performance overhead imposed by the extra instructions of each recovery method (ACCE and our method) compared to the baseline program binaries (which do not have any optimizations) is illustrated in Figure 4. The high performance overhead imposed by both methods for *qsort* and *crc* workloads is due to the fact that they are tiny programs and the extra assertion instructions added by the recovery schemes cause higher overheads in small programs. We deliberately selected these two workloads to investigate the worst-case-scenario.

In the chart of Figure 5 the normalized recovery efficiency factor of ACCE and our method are presented for the set of workloads. The efficiency factor (REF) of ACCE is calculated using its minimum recovery time. In ACCE instrumentation,

the recovery time depends on the number of basic blocks in the functions and whether the erroneous branch destination is inside the same function or it targets a basic block in another function. If it is inside the same function the recovery time is shorter and if the faulty target block belongs to another function the latency is higher. The minimum recovery time in ACCE is when the CFE initiates in the first basic block of the function and targets a faulty block in the same function. In ACCE implementation for our target processor, the minimum recovery time is 32 cycles. This is the minimum recovery time just to roll back the execution to the correct point, but without restoring the correct data of the modified registers. The constant recovery time in our method including the restoration of the correct data content of the register file is 28 cycles. As depicted in the chart, REF number of our scheme is about 80 times higher than REF number of ACCE.

V. Conclusion

In this paper we have introduced a lightweight, low-latency CFE recovery method with checkpoints only in the susceptible source basic blocks. Our proposed recovery scheme is able to detect the CFE and roll back the execution to the beginning of the basic block where the CFE has occurred with a latency of only 28 cycles. We have modified a previously proposed bit-flip analysis framework to identify the susceptible sources of CFE in the program. This information is used to limit the checkpoint locations only to the susceptible source blocks and decrease the imposed overhead dramatically. In order to assess our recovery method fairly we have to consider the three metrics of correctability, performance and recovery time. For this reason we have introduced the recovery efficiency factor that is calculated based on all three crucial metrics. Comparing our scheme to a well known recovery scheme (ACCE), our method is more efficient by a factor of eighty. The main reason behind the efficiency of our proposed method is that the instrumentations for detection and recovery are added exactly at the identified vulnerable spots of the program.

References

[1] G. Nazarian, D. G. Rodrigues, lvaro F. Moreira, L. Carro, and G. Gaydadjiev, "Bit-flip aware control-flow error detection," in *23rd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, March 2015, pp. 215–221.

[2] F. Ayatolahi, B. Sangchoolie, R. Johansson, and J. Karlsson, "A study of the impact of single bit-flip and double bit-flip errors on program execution," *Computer Safety, Reliability, and Security Lecture Notes in Computer Science*, vol. 8153, pp. 265–276, 2013.

[3] D. S. Khudia and S. Mahlke, "Low cost control flow protection using abstract control signatures," in *Proceedings of LCTES*, June 2013, pp. 3–12.

[4] T. S. Ganesh *et al.*, "Seu mitigation techniques for microprocessor control logic," in *Proceedings of the Sixth European Dependable Computing Conference (EDCC'06)*, 2006, pp. 77–86.

[5] A. Mahmood and E. J. McCluskey, "Concurrent error detection using watchdog processors-a survey," in *IEEE Trans. on Computers*, 1988, pp. 160–174.

[6] F. A. Bower, D. J. Sorin, and S. Ozev, "A mechanism for online diagnosis of hard faults in microprocessors," in *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO05)*, November 2005, pp. 197–208.

[7] N. Saxena and E. J. McCluskey, "Dependable adaptive computing systems the roar project," in *Proceedings of International Conference on Systems, Man, and Cybernetics*, 1998, pp. 2172–2177.

[8] A. Shye, T. Moseley, V. Reddi, J. Blomstedt, and D. Connors, "Using process-level redundancy to exploit multiple cores for transient fault tolerance," in *International Conference on Dependable Systems and Networks, DSN*, June 2007, pp. 297 – 306.

[9] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Error detection by duplicated instructions in super-scalar processors," *IEEE Trans. on Reliability*, vol. 51, no. 1, pp. 63–75, March 2002.

[10] G. A. Kanawati, V. S. S. Nair, N. Krishnamurthy, and J. A. Abraham, "Evaluation of integrated system-level checks for on-line error detection," in *Proceedings of IEEE International Computer Performance and Dependability Symposium*. IEEE, September 1996, pp. 292–301.

[11] Z. Alkhalifa, V. S. Nair, N. Krishnamurthy, and J. Abraham, "Design and evaluation of system-level checks for on-line control flow error detection," *IEEE Trans. on Parallel and Distributed Systems*, vol. 10, no. 6, pp. 627–641, June 1999.

[12] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Control flow checking by software signatures," *IEEE Trans. on Reliability*, vol. 51, no. 1, pp. 111–122, March 2000.

[13] O. G. ad M. Rebaudengo, M. S. Reorda, and M. Violante, "Soft-error detection using control flow assertions," in *18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*. IEEE, November 2003, pp. 581–588.

[14] R. Vemu and J. Abraham, "CEDA: Control-flow error detection using assertions," *IEEE Trans. on Computers*, vol. 90, no. 9, pp. 1233–1245, September 2011.

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

[16] G. A. Reis *et al.*, "Swift: software implemented fault tolerance," in *Proceedings of International Symposium on Code Generation and Optimization,CGO*, March 2005, pp. 243–254.

[17] D. S. Khudia, G. Wright, and S. Mahlke, "Efficient soft error protection for commodity embedded microprocessors using profile information," in *Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*, May 2012, pp. 99–108.

[18] N. S. Bowen and D. K. Pradhan, "Processor- and memory-based checkpoint and rollback recovery," *Computer*, vol. 26, no. 2, pp. 22–31, February 1993.

[19] D. Szentivanyi, S. Nadjm-Tehrani, and J. M. Noble, "Optimal choice of checkpointing interval for high availability," in *Proceedings of Pacific Rim International Symposium on Dependable Computing*, December 2005.

[20] G.-L. Park, H. Y. Youn, and H.-S. Choo, "Optimal checkpoint interval analysis using stochastic petri net," in *Proceedings of Pacific Rim International Symposium on Dependable Computing*, 2001, pp. 57 – 60.

[21] S. Feng, S. Gupta, A. Ansari, S. A. Mahlke, and D. I. August, "Encore: low-cost, fine-grained transient fault recovery," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, December 2011, pp. 398–409.

[22] R. Vemu, S. Gurumurthy, and J. Abraham, "Acce: Automatic correction of control-flow errors," in *Int. Test Conference*, 2007, pp. 1–10.

[23] Cosy compiler. [Online]. Available: http://www.ace.nl/compiler/cosy

[24] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *International Workshop on Workload Characterization*, 2001, pp. 3–14.