Compiler-Aided Methodology for Low Overhead On-line Testing

Ghazaleh Nazarian*, Robert M. Seepers†, Christos Strydis† 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@tudelft.nl
† Dept. of Neuroscience, Erasmus MC, Dr. Molewaterplein 50, 3015 GE Rotterdam, The Netherlands
{r.seepers, c.strydis}@erasmusmc.nl
‡ Dept. of Computer Science and Engineering, Chalmers University of Technology, Rannvagen 6, Goteburg, Sweden
georgig@chalmers.se

***Abstract*—Reliability is emerging as an important design criterion in modern systems due to increasing transient fault rates. Hardware fault-tolerance techniques, commonly used to address this, introduce high design costs. As alternative, software Signature-Monitoring (SM) schemes based on compiler assertions are an efficient method for control-flow-error detection. Existing SM techniques do not consider application-specific-information causing unnecessary overheads. In this paper, compile-time Control-Flow-Graph (CFG) topology analysis is used to place best-suited assertions at optimal locations of the assembly code to reduce overheads. Our evaluation with representative workloads shows fault-coverage increase with overheads close to Assertion-based Control-Flow Correction (ACFC), the method with lowest overhead. Compared to ACFC, our technique improves (on average) fault coverage by 17%, performance overhead by 5% and power-consumption by 3% with equal code-size overhead.**

I. Introduction

The past decade has witnessed the rapid integration of embedded processors in a variety of new applications [1]. On the other hand, the ongoing technology trends of shrinking feature sizes and increasing chip density made processors more susceptible to transient faults [2]. Therefore, reliability becomes a major issue especially for safety-critical embedded systems such as biomedical implants. A large body of research on hardware optimizations for fault detection and fault recovery exists, where hardware is replicated [3] or extended with circuit checkers [4] in order to detect and correct faults. However, such optimizations are not applicable for off-the-shelf processors suitable for many systems.

Alternative techniques for run-time fault detection without special hardware are compile-time software optimizations. The executable code is instrumented with extra instructions to detect program's misbehavior due to hardware transient faults. Such faults can manifest as data errors or Control-Flow Errors (CFE). Experiments on the influence of heavy-ion fault injection on program behavior shows that more than half of the injected faults cause CFEs [5], hence CFEs deserve special attention and will be targeted in this study.

A well-known compiler-assisted technique for on-line CFE detection is Signature Monitoring (SM). In SM, unique signatures are associated with each basic block (branch-free sections of the code). At compile time, the code is instrumented with *set* and *test* assertions. *Set* assertions, used in all Basic Blocks (B-block), calculate and update the runtime signature. *Test* assertions, added at predefined program locations, compare the runtime signature with the associated signatures to verify correct execution. However, redundant assertions introduce overheads. Safety-critical systems such as biomedical implants have extremely low power and memory budgets making high-overhead methods not applicable. Therefore, reducing SM overheads with minimal impact on fault coverage is necessary.

Existing SM techniques do not use application-specific information (e.g., the Control-Flow-Graph (CFG) topology), thus they often have unnecessary assertions. In this paper, novel SM method for on-line CFE detection is proposed which instruments the assembly code based on workload CFG analysis at compile time. Our method is named Selective-Control-Flow-Check (SCFC) because, depending on the CFG topology, a suitable assertion is selected for each B-block. SCFC inserts the lowest number of assertions with low overheads only at critical points of the code while preserving acceptable fault coverage levels. The contributions of this paper are:

- Categorization of existing SM techniques into two classes to highlight pros and cons of each category;
- New software-based method, without extra hardware;
- On average 17% fault coverage improvement over Assertion-based Control Flow Correction (ACFC) with only 2.75% of code-size overhead.

The remainder of this paper is organized as follows: Section II gives an overview and a categorization of previously proposed methods and Section III illustrates two methods representing each category with an example. In Section IV, the targeted error models are introduced. Section V describes SCFC and its advantages over other SM methods. In Section VI the workloads used for experiments, framework for error injection and the results of our evaluation are presented. Finally, the conclusions are provided in Section VII.

II. Related work

One of the major threats to processor reliability are transient faults which can be detected using redundancy. The redundancy can be extra hardware, an extra thread or some additional lines of code in the executed program. Several

978-1-4799-0103-6/13/$31.00 ©2013 IEEE

SM METHODS	SET ASSERTION	TEST ASSERTION	CF-PARAMETERS	CATEGORY
CFCSS [8]	$RS = RS \oplus P_{1i}$ $RS = (RS \oplus P_{1i}) \oplus P_2$	$if RS! = S_i \ br \ error$	$P_{1i} = S_i \oplus S_{pre1}$ $P_2 = \begin{cases} 0000 & \text{in predecessor 1} \\ S_{pre1} \oplus S_{pre2} & \text{in predecessor 2} \end{cases}$	Pred/Succes
ECCA [9]	$RS = P_i + (RS - S_i)$	$RS = \frac{S_i}{RS\%S_i \cdot (RS\%2)}$	$P_i = \prod S_{nxt}$	Pred/Succes
YACCA [10]	$RS = (RS \ \& \ P_{1i}) \oplus P_{2i}$	$If\ (P_{3i}\%RS) \quad error\ ()$	$P_{1i} = S_{pre1} \bar{\oplus} S_{pre2}$ $P_{2i} = S_{pre1} \ \& \ (S_{pre1} \bar{\oplus} S_{pre2}) \oplus S_i$ $P_{3i} = \prod S_{pre}$	Pred/Succes
CCA [11] and CEDA [12]	$RS_1 = S_{1i}$ $RS_2 = S_{2i}$	$if RS1! = S_{1pre} \quad error()$ $if RS2! = S_{2i} \quad error()$	not required	Pred/Succes
ACFC [13]	RS = RS ^ MASK	$if RS! = CONST \quad error()$	not required	Path

TABLE I: Set/Test assertions and additional static parameters of signature monitoring methods

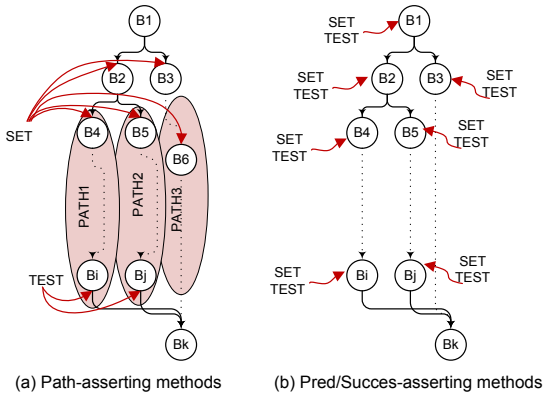


Fig. 1: Two categories of SM techniques

hardware-redundancy methods use a watchdog processor, to compare its results with the main processor [6] [4] and others propose to replicate only parts of the processor such as datapath-components [3]. Saxena et al. [7] propose to use redundant threads for soft-error detection in processors with hardware support for multi-threading. Most of the above techniques need special hardware which is costly and not available in many off-the-shelf-processors. In software (compiler-aided) methods the program code is instrumented with extra instructions to check execution correctness. Well-known methods are: EDDI [14] which duplicates the instructions and checks the consistency between the two versions for data error detection, and signature-monitoring schemes for CFE detection. Control flow Correcting Assertion (CCA) [11], ECCA [9], CFCSS [8], YACCA [10], CEDA [12] and ACFC [13] are all variations of SM. SWIFT [15] is a hybrid method combining CFCSS for CFE and EDDI for data error detection.

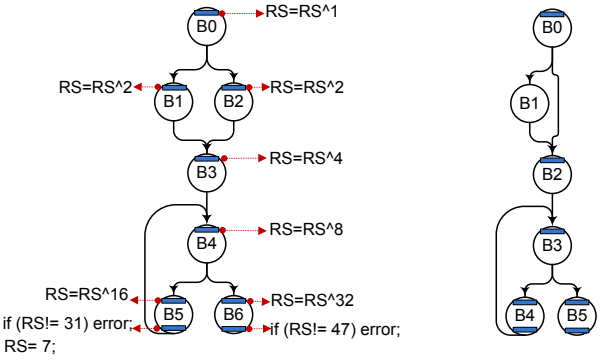
In this paper, we focus only on signature-monitoring detection schemes. In SM techniques, a unique signature is assigned statically to each B-block. In addition to the B-block signatures, there is the *Runtime Signature* (RS) which is generated at runtime. The RS value depends on the program's execution flow and the visited B-blocks during execution. *Set* assertions update the RS with the current B-block signature. *Test* assertions check the correctness of the RS content, to validate execution correctness. *Set* and *Test* assertions are added at specific points of the program (begin/end of B-blocks) at compile time. We divide SM techniques into two categories as depicted in Figure 1: **(a) Path-asserting:** methods which assert if the control-flow path during the execution is correct or not. A *path* consists of two or more B-blocks executed in an uninterrupted sequence; and **(b) Predecessor/Successor-asserting:** methods which assert if the previous (or next) B-block in the execution flow is the correct predecessor (or successor). Predecessor/Successor-asserting methods require more than one assertion per B-block (at least one *set* and one *test*), as shown in Figure1(b). Moreover, in some cases there is a need to save extra information about predecessors/successors (e.g., for B-blocks with multiple predecessors/successors); the so-called Control Flow parameters (CF-parameters). Path-asserting methods decrease the number of assertions per B-block (*test* assertions are needed only in the last B-block of each path) and do not require saving CF-parameters, Figure1(a). However, path-asserting methods depend on the program's CFG and require symmetric topology as will be explained in the next section. Predecessor/successor-asserting is CFG-topology independent.

The most widely used SM schemes are presented in Table I. The table depicts the *set* and *test* assertions, the extra CF-parameters saved at compile time and the category. The signature of each B-block is denoted by an "*i*" subscript (S_i). Predecessor signatures are denoted by "*pre*" subscripts (e.g., S_{pre}), and successors signatures are denoted by "*nxt*" subscript (e.g., S_{nxt}). CF-parameters stored at compile time are indicated with "*P*".

The number of assertions per B-block and CF-parameters have significant impact on memory overhead. Moreover, the number of assertions used per B-block and the assertions complexity determine performance and power overheads. CFCSS, ECCA, CEDA and YACCA are predecessor/successor-asserting methods with two or more complex assertions per B-block and extra CF-parameters. Therefore, these methods have high overheads. CCA is also a predecessor/successor-asserting method, but with simple *set/test* assertions and no extra CF-parameters. Our hybrid method benefits from the advantages of each category by using a combination of assertions from each category in different CFG sections. Among predecessor/successor-asserting methods with high fault-coverage, we use CCA assertions since its assertions are simple and do not require CF-parameters. We use ACFC as the the only available method with path-assertion.

III. Method Comparison

In this section ACFC and CCA are explained and compared using an example. In ACFC one *Set* assertion is used at the



(a) Sym. CFG with ACFC assertions (b) Asym. CFG

Fig. 2: Sym. CFG with ACFC assertions and an Asym. CFG

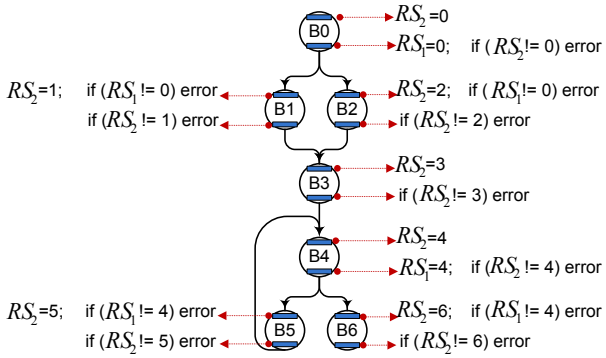


Fig. 3: CFG with CCA assertions

beginning of each B-block, and a *Test* assertion is used at the end of each possible control-flow path. Control-flow paths are guarded by RS. Each B-block in a control-flow path is represented by one bit in the associated RS. The Set assertion (shown in Table I) in each B-block sets its corresponding bit in the RS, by a bitwise XOR between RS and the B-block MASK. The MASK of a B-block has a value with "1" at the bit position corresponding to the B-block and all other bits set to "0". The *Test* assertion compares the runtime signature value with the CONST representing error-free control-flow path. The CONST of a control-flow path has a value with only the bits corresponding to B-blocks in the control-flow path set to "1". Since the *Set* assertion uses bitwise XOR, if a B-block is executed an even number of times, its bit in RS will be reset. To overcome this, ACFC suggests to test and restore RS at the end of each loop iteration. RS is restored using the CONST of the control-flow path before entering the loop.

Figure 2(a) shows the ACFC scheme applied to an example with simple CFG topology. In the depicted CFG, the possible paths without considering the in-loop B-blocks are {B0, B1, B3, B6} and {B0, B2, B3, B6}. ACFC sets the corresponding bits of RS in each B-block and in the last B-block of the path (B6) compares RS content with CONST of the path which is 47 (101111'b). It should be noted that B1 and B2 are multi-path B-blocks that show up in the CFG representing an if-else program statement. ACFC uses the same RS bit for multi-path B-blocks (bit #2 in Figure 2), because at runtime only one of them is executed. The path in the loop is {B4, B5} which uses the RS 4th and 5th bits. In the last B-block (B5) RS is compared with 31 (011111'b) and restored with the CONST value before the loop, which is 7 in our example.

ACFC (a path-asserting method) is designed to be used only in symmetric CFG topologies. The inefficiency of this method with asymmetric topologies is explained with an example. Figure 2(b) shows an asymmetric CFG in which B1 may or may not be executed at runtime. For this reason, ACFC can not assert B1 execution by *setting* its RS bit and *testing* RS content at the end of the path. All programs with *if* statements without an *else* part produce asymmetric topologies. To cope with this problem, the method requires a dummy *else* statement to make the CFG symmetric. As a result, this solution increases the overheads. Moreover, ACFC extracts the CFG from and instruments directly the program source-code. However, the CFG of the program does not remain the same at different phases of the compilation before the executable binary is produced. In the majority of cases, the final CFG topology of the code is asymmetric and more complicated than the symmetric topologies assumed by ACFC.

Another shortcoming of ACFC is related to loops: At the end of each loop RS has to be restored with the control-flow-path value at the loop beginning. As a consequence, a CFE from a B-block outside the loop to the *restore* statement at the end of the loop can not be detected. As an example in Figure 2(a) an erroneous jump from B0 to the last statement of B5 ($RS = 7$) can not be detected using ACFC.

CCA is the simplest among the predecessor/successor-asserting methods. Figure 3 depicts the same CFG instrumented with CCA assertions. A pair of set RS_2 (at the beginning of the B-block) and test RS_2 (in the end of the B-block) guard B-block uninterrupted execution and detect erroneous jumps to/from mid of the B-block. A pair of set RS_1 (in the end of the predecessor B-block) and test RS_1 (at the beginning of the current B-block) checks the predecessor correctness. In case inconsistencies are detected an error recovery routine will be called. CCA requires 4 assertions for almost all B-blocks, causing significant overheads in terms of performance and memory. Another shortcoming of CCA is its inability to detect errors in B-blocks with multiple predecessors.

IV. Error Model

In this paper, we target Single-Event Upset (SEU) as our fault model. SEUs are typically caused by electro-magnetic radiation or wire crosstalk and usually result in single bit flips in different parts of the system: memory (data or code segment), buses (data or address), functional units or control logic, to name a few. Hardware faults lead to software errors. In this paper, *faults* refer to malpractices in hardware and *errors* refer to the effects caused by those faults in the software. The effect of SEUs on program instructions can be a change in instruction's address, opcode or its operands, which results in two conceptually different error types:

- *Data errors*: Faulty change of instruction opcode or operands (excluding control-flow instructions), will cause erroneous data in the memory or registers;

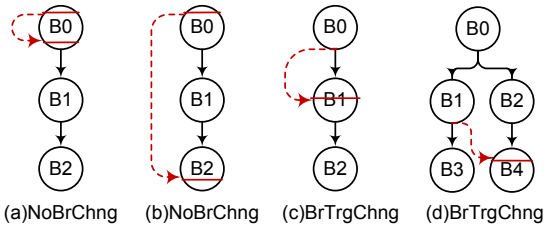


Fig. 4: Targeted control-flow errors

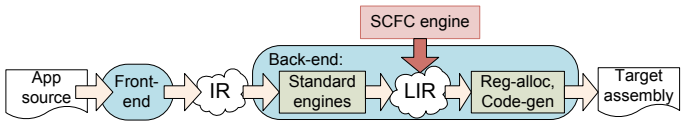


Fig. 5: CoSy framework

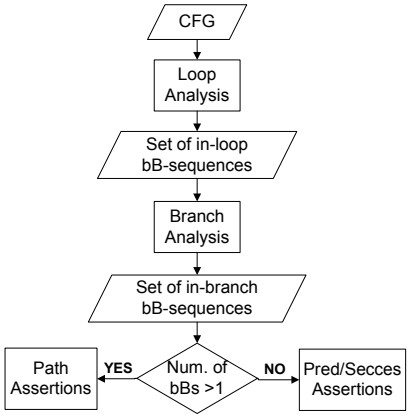


Fig. 6: CFG processing and SCFC instrumentation

- *Control-flow errors (CFE)*: Faults changing the opcode or operand of control-flow instructions or converting a non-control-flow instruction opcode to a control flow one causing incorrect execution sequence.

Since CFEs occur often and have a large impact on program behavior, we target this error type in our study. Figure 4 shows how CFEs affect program's execution. *NoBrChng* are errors in which a non-branch instruction is changed to a branch instruction. The consequence of NoBrChng is an erroneous jump from the **middle** of a B-block to the end of the same B-block (Figure 4(a)) or to another B-block in the CFG (Figure 4(b)). In *BrTrgChng* error type the operand of a branch instruction is changed. A BrTrgChng error causes an erroneous jump from the **end** of a B-block to a random location (Figures 4(c) and 4(d)). The case when an error cause a branch instruction change to a non-branch will either behave as BrTrgChng or will not result in a CFE[1].

V. SCFC IMPLEMENTATION

Considering the high number of assertions of predecessor/successor-asserting methods and the shortcomings of path-asserting methods with asymmetric CFG topologies, there is a need for an SM technique with a reasonable number of assertions which is usable in asymmetric CFG topologies. In addition, the optimizations should be applied at a point (in terms of compiler passes) where the final CFG topology is available. For this reason, we implemented SCFC using the intermediate representation used by the CoSy back-end (LIR in Figure 5) which has the CFG of the final assembly code. In this section, first the development framework is introduced followed by SCFC explanation and an example using the CFG of a realistic workload.

A. Framework for compile-time optimizations

CoSy is a modular framework specially developed for simplifying compiler design and optimization [16]. Figure 5 shows the framework constituted by different modules (so called engines) responsible for different compilation tasks, e.g, register-allocation, scheduling, etc. Additional compiler optimizations can be implemented as a new engine. The proposed SCFC optimization is implemented as an engine. In the compiler generated by CoSy, first the front-end generates an Intermediate Representation (IR) of the program. After the IR is processed by a group of optimization engines, the compiler back-end transforms the IR to Lower-Intermediate-Representation (LIR). This version is closer to the assembly code and represents closely the final CFG topology. Therefore, our SCFC engine manipulates the LIR. Moreover, since the SCFC engine is added after the scheduler engine, all required extra assertions will not be relocated or optimized.

B. SCFC optimization

By analysis of a program's CFG, all available paths (sequence of two or more B-blocks executed sequentially) and the lonely-blocks[2] are identified. SCFC processes the workload's CFG and determines the B-blocks in loops, B-blocks in different control-flow paths and lonely blocks. SCFC uses two assertion types based on the CFG analysis. Figure 6 depicts the flowchart of the SCFC algorithm. The algorithm consists of three steps; (1) *loop analysis*, (2) *branch analysis* and (3) *adding assertions*. At the first step, B-blocks residing in each loop are extracted and saved as separate sequences and B-blocks outside loops in a single sequence. At the second step, all sequences are processed to extract B-blocks which are in different paths of the control flow due to conditional branches. Third, the B-block sequences of the second step are used to decide on the assertions type required for each B-block. *Path-assertion is applied for B-blocks sequences with lengths of 2 or more. Predecessor/successor-assertions are used for all lonely-blocks.*

Path-*set*-assertions are used at the beginning of each B-block in paths and only one path-*test*-assertion is required at the end of the last B-block of the same path. Predecessor/successor-*set*-assertions are used for the predecessor B-block of a lonely-block and the corresponding *test* assertions are added at all lonely-blocks. The working of the two SCFC assertion categories is as follows.

[1]This will potentially cause a data error not targeted here.

[2]B-blocks that can not be grouped in any path.

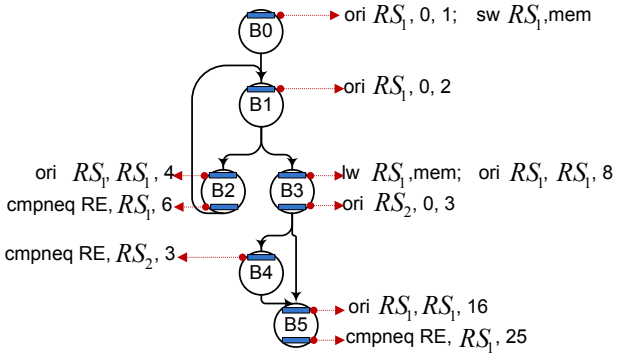


Fig. 7: CFG with SCFC optimization

1) SCFC-Path-assertions: The *set* assertion is an OR instruction with an immediate representing the B-block MASK (ori RS_1, 0, MASK). The *test* assertion is *cmpneq* instruction to compare the contents of RS_1 with the path CONST (cmpneq RE, RS_1, CONST). RE is a restricted register from the processor register-file that is used to hold the results of fault detection.
2) SCFC-pred/succes-assertions: We use an OR instruction as *set* assertion to set RS_2 to the signature of the B-block ($oriRS_2, 0, Sig_B$). The *test* assertion is an instruction comparing RS_2 contents with the predecessor signature ($cmpneqRE, RS_2, Sig_{preB}$). If there is an inconsistency RE is updated.

Path-assertions usage is limited to sequences with at least two B-blocks, because if they are applied to lonely-blocks, two assertions are used at each B-block which has the same code-size overhead as predecessor/successor-assertions. While predecessor/successor-assertions use one runtime signature for all B-blocks of the routine, path-assertions occupy one bit of the run-time signature per B-block until the end of the routine, therefore depending on the number of B-blocks lengthy runtime signatures can be expected.

Figure 7 shows a subgraph from the Checksum (*CSUM*) benchmark CFG instrumented with SCFC assertions. The first step of CFG processing produces the following B-block sequences: {B1, B2} and {B0, B3, B4, B5}. From the sequences above, the second step of CFG processing generates the following set of B-block sequences: {B1, B2}, {B0, B3, B5} and {B4}. B4 is the only lonely-block and is instrumented with intra-block predecessor/successor-assertions. The intra B-block predecessor/successor-assertions for B4 are a set assertion (ori RS_2, 0, 3) at the end of the predecessor block (B3) and a test (cmpneq RE, RS_2, 3) in the begin of B4. B-blocks in sequences {B1, B2} and {B0, B3, B5} are protected using path-assertions. Path-asserting *set*s are added to each B-block of these two paths and path-asserting *test* is added only in the two last B-blocks; B2 and B5. {B0, B3, B5} is the main control-flow path and {B1, B2} is a loop path inside the main path. Execution flow at B1 can continue to B3 in the main path or can enter into the loop-path. Therefore, before the execution flow reaches this point, the content of the path-asserting runtime signature (RS_1) should be stored and retrieved in the next B-block after the loop path. To store *sw* RS_1,*mem* instruction in B0 stores RS_1 content to memory and to retrieve *lw* RS_1,*mem* retrieve RS_1 contents after the loop. RS store and retrieve are needed for while-loop and nested-if-statement structures. This is due to the fact that these statements can cause paths that may or may not be accessed during the execution flow.

Advantages: (1) Since SCFC instruments the program assembly code, there is no interference with other compiler optimizations making it more precise than ACFC and CCA (both working at source-code level). (2) Compared to CCA, SCFC always reduces the total number of assertions. For instance for the depicted CFG in Figure 7, CCA adds 17 assertion statements while SCFC uses only 9. (3) Other than the ACFC method, SCFC does not require *restoring* RS at loop boundaries, therefore overheads are reduced. (4) Our method can instrument programs with symmetric and asymmetric CFG topologies while ACFC requires symmetric topologies. In the example CFG of Figure 7, ACFC adds 8 extra statements (almost equal to SCFC) for *set*, *test* and *restore* of RS, without guaranteeing correct execution of B4.

Limitations: The result of the comparison in *Test* assertions are written into a restricted register, reserved for *Test* assertions only. We are aware that register-file reduction slightly increases processor register pressure. This may result in higher number of load/store instructions due to register spill to memory. The overheads in terms of power and performance, presented next, include the contribution of these extra load/store instructions. The fault detection framework used in this work checks the value of the restricted register in the simulated traces (our simulator is explained in the next section), to detect fault occurrences. In practical SCFC implementations, in case of a fault, a dedicated fault-recovery routine will be invoked. Fault-recovery techniques, however, are beyond the scope of this paper.

VI. Experimental Setup and Results

In this section the workloads-under-test, our experimental setup and SCFC simulation results are presented. We evaluate our proposal and compare it against ACFC and CCA in terms of performance, code-size and power-consumption overheads in addition to fault coverage.

A. Workloads

We experimented using a subset of ImpBench [17]. ImpBench is a benchmark suite with four categories of low-power applications typical for biomedical implants. The three categories represent generally used applications in embedded biomedical systems: *compression*, *encryption* and *data-integrity*. Within the compression and encryption categories, after profiling, programs have been found to exhibit similar CFG characteristics. Therefore, from each category we have chosen a single representative benchmark with the smallest CFG, i.e. FINNISH from compression and RC6 from encryption. Small CFGs are chosen to reveal the highest overheads of the methods for each application category. In larger programs, the overhead of adding assertions is expected to be lower. An exceptional case is the data-integrity category, in which the two benchmarks (CRC and CSUM) have very different CFG types. Therefore, we used both in our experiments. It is important to note that we have not used programs of the *real-application*

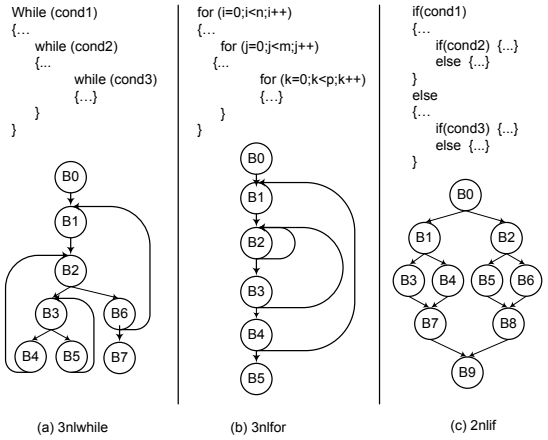


Fig. 8: Control flow dominated test kernels

category, as they are very specific for biomedical embedded systems only. We also used a set of control-flow dominated synthetic kernels to investigate the worst-case overheads.

Each synthetic test kernel implements a special CFG topology case. They are designed to represent a set of CFG topologies with many nested control statements. Possible statements are *do-while*, *while-do* loops, *for-loops* and *if-then-else* constructs. *For-loops* are a variant of *do-while loops* with an initialization statement at the beginning and an increment at the end, resulting in the same CFG-structures. Therefore, in our test kernels, we do not consider do-while-loops. Our test programs are constructed using different combinations of *for-loops*, *do-while* and *if-then-else* statements.

The maximum nesting level in our test kernels is 3. This level has enough complexity to evaluate our method while it still allows running 1000 instances on the simulator (presented in the next sub-section). For the three control statements and nesting-level 3, a total of 27 combinations (CFG topologies) exist. Out of these, we build test programs for the three topologies that represent the worst-case scenarios (highest number of required assertions), depicted in Figure 8. *3nlfor* has three nested levels of *for-loops* where the body of the loops contains a simple addition or subtraction statement. *3nlwhile* has nested *while-loops* with simple body statements. *2nlif* is composed of two nested levels of *if-then-else* statements.

B. Experimental Setup

In our experiments we use an embedded 32-bit RISC processor. Our method however can be re-targeted to any arbitrary processor as it modifies only the intermediate code representation. Workload binaries generated by our customized compiler are evaluated using Synopsys Processor Designer cycle-accurate simulator [18]. In what follows, our target architecture and the error-injection method are described.

Target architecture: Our target architecture is a basic, 32-bit, five-stage, in-order RISC processor which has, other than a forwarding unit, no advanced micro-architectural features. This processor has similar load/store-based ISA as any ARM processor. The only significant difference is the higher number of registers of ARM processors. Therefore, the overheads of

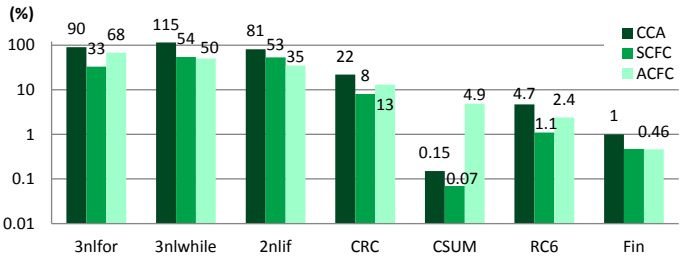


Fig. 9: Performance overheads

the SCFC instrumentation in a standard processor (such as ARMv7m) are expected to be lower than our target processor.

Error injection: NoBrChng and BrTrgChng errors discussed in section IV are emulated using a special error-injector instruction and a Linear-Feedback-Shift-Register. These errors demonstrate hardware faults effect on the control-flow of the running program. The special error-injector instruction and the Linear-Feedback-Shift-Register (LFSR) are implemented in the simulator. The error-injector instruction is added in the beginning of the program-under-test along with a random value (generated by RANDOM linux command) as its operand. The random value is used as a LFSR seed and determines the trigger time of the error. After the number of cycles specified by the trigger time has elapsed, a NoBrChng/BrTrgChng error is generated. In NoBrChng error, the opcode of a non-branch instruction is changed to a branch instruction with a random value as its operand. BrTrgChng error changes the operand of a branch instruction to a random value. The corresponding random values for branch operands are generated by the LFSR. The polynomial used to generate our pseudo random numbers is: $x^{32} + x^{31} + x^{29} + x + 1$ [19].

C. Experimental Results

We investigate static memory (code-size), performance and power overheads of the SCFC technique for the selected ImpBench workloads. To study the worst-case overheads for our method and compare them to those of ACFC and CCA, we also use the three synthetic test kernels introduced earlier. Normally, real code contains more than one statement in its control-statement bodies resulting in lower net overheads due to adding the extra *Test* and *Set* assertions. The performance of each method is estimated using the total number of required cycles for completing the program in the simulator. Performance and static memory overheads of the three optimization methods are shown in Figures 9 and 10. The overheads of each method are calculated with respect to an unprotected (for CF errors) version of the program.

As expected, the performance and static memory overheads of SCFC are lower than those of CCA. A peculiar case of obtained overheads is seen in *3nlfor*, *CRC*, *CSUM* and *RC6* workloads where SCFC has lower overheads than ACFC. For the *3nlfor* kernel the reason is that its CFG topology (depicted in Figure 8(b)), has only one main control-flow path from the first B-block to the last one while SCFC adds instructions for restoring RS only in cases where the control-flow may diverge from the main path of the execution (if there are multiple control-flow paths between the first and last B-block).

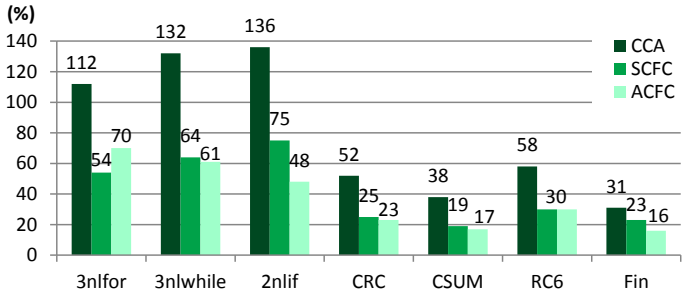


Fig. 10: Static memory overheads

Instruction category	Current range[mA] $0.5\mu m$	Power range[mW] $0.5\mu m$	Power range[mW] $90nm$
Arith./logic	172-179	567.6-590.7	36.03-37.50
Load	185-192	610.5-633.6	38.76-40.22
Store	169-175	557.7-577.5	35.40-36.66

TABLE II: Power model of the ISA

Therefore, SCFC instructions for restoring RS are not required in *3nlfor*, while ACFC adds *reset* statements for restoring the RS value at the end of all loops. As a consequence the imposed static memory and performance overheads of ACFC in *3nlfor* are bigger than those of SCFC.

Considering all workloads, on average[3] SCFC has similar static memory overhead to ACFC. With respect to performance, SCFC has 5% less overhead compared to ACFC, on average. The main reason for this seemingly unexpected reduction in performance overhead is the asymmetric topologies of some of the benchmarks. More precisely, *CRC*, *CSUM* and *RC6* workloads have asymmetric CFGs while ACFC requires symmetric topologies (as explained in section III). ACFC adds dummy *else*s to complement all *if-statements* and balance the CFGs. Dummy *else*s cause extra branches and impact performance and power results. As a result, for these benchmarks, ACFC has higher performance overhead than SCFC. Since *CSUM* is small, dummy *else* statements cause higher overheads.

For estimating the power consumption, we use the same approach as presented in [20]. The authors there used a model for calculating power consumption of programs given by Tiwari et al. in [21]. In this model, for each instruction category a current-value range is specified. The power of each category is calculated by multiplying this value with the supply-voltage ($P = I * V$). The numbers reported in [21] are for $0.5\mu m$ technology with 3.3 V supply-voltage. In order to estimate power consumption in our experimental processor (with $90nm$ technology), we scale down the numbers found in [21]. In Table II we show the numbers for $0.5\mu m$ technology and our projected versions for $90nm$ technology. The scaling factor from $0.5\mu m$ to $90nm$ is determined using the dynamic power consumption ($P_{dyn} = \alpha f C V^2$) and CMOS transistor capacitance ($C = (\epsilon)S/d$) equations. In the former, α is the activity factor (we assume it is not affected by the technology), C is the overall switching capacitance, V is the supply voltage and f is the operating frequency. In the latter equation, ϵ is the dielectric constant of the oxide material, S is the oxide surface (length multiplied by width) and d is the thickness of the oxide. Assuming similar oxide material (SiO_2) for the two technologies (constant ϵ) and similar gate widths (W_{ox}), the simplified power scale-down factor becomes:

$$s = \frac{P_{dyn}(0.5\mu m)}{P_{dyn}(90nm)} = (V^2_{0.5\mu m}/V^2_{90nm})(\frac{L_{0.5\mu m}}{L_{90nm}})(\frac{d_{90nm}}{d_{0.5\mu m}})$$

With typical values for $0.5\mu m$ technology of $V = 3.3V, L = 0.5\mu m, d = 8nm$ and for $90nm$ technology, $V = 1.2V, L = 90nm, d = 3nm$ the scale-down factor becomes 15.75. By multiplying the accumulated number of instructions in each category by the corresponding power-value we have estimated the overall power consumption of each benchmark for all three cases under study. The power consumption overhead of each method is calculated with respect to the base-line version with no compiler optimizations. For simplicity, we have assumed that the static power has similar contribution to the total power consumption for all binaries, optimized and baseline. We are aware that our estimations are rough, but since we study the differences between the three methods, we expect that any offsets in the estimated numbers compensate each other. Figure 11 depicts the power overheads and shows that SCFC binaries exhibit lower power overheads than ACFC for the 3nlfor kernel, CRC, CSUM and RC6 benchmarks. The high power overhead of ACFC for CSUM is due to the asymmetric topology of CSUM and its relatively small size, as explained above. On average, SCFC has 3% lower power overhead as compared to ACFC.

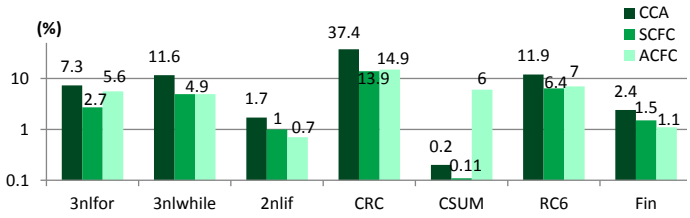


Fig. 11: Power overheads

Figure 12 depicts the fault coverage results of 1000 simulation runs for the two error types discussed in section IV. To determine the fault coverage, we run the benchmarks using the smallest data sets possible, as the contribution of the input size with respect to the fault coverage is negligible. Each bar group shows the results for one error type and one benchmark. The last bar group depicts the average fault coverage of each method for all cases. As expected, SCFC has always higher fault coverage than ACFC. On average, the fault coverage of SCFC is 17% higher compared to ACFC. The exceptional case of higher error coverage of SCFC for *CSUM* benchmark compared to CCA is due to the peculiar CFG topology of *CSUM*. *CSUM* has two loops containing B-blocks with multiple predecessors. A significant portion of the execution time is spent in the for-loops and a high number of errors are injected in the corresponding B-blocks. However, since these B-blocks have multiple predecessors, errors are not captured by CCA. On the contrary, SCFC does cover B-blocks with multiple predecessors, that results in improved error coverage for the *CSUM*.

[3]We use median, since data is not normally distributed.

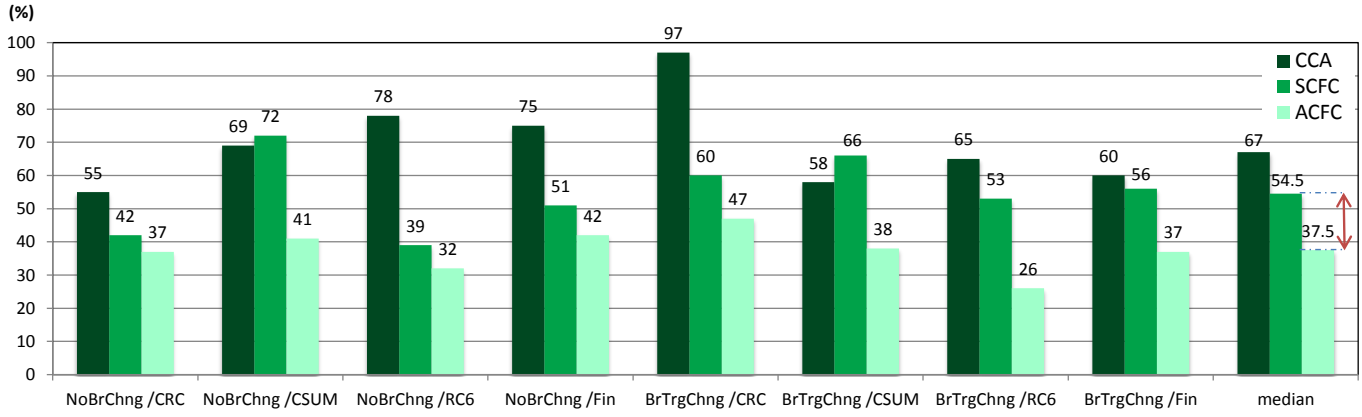


Fig. 12: Fault coverage comparison between ACFC, CCA and SCFC for the error set.

VII. Conclusions

In this paper we presented a novel technique for application-specific control-flow fault detection. The proposed SCFC technique is a workload-aware, hybrid combination of methods from both SM categories: path-asserting and predecessor/successor-asserting. In SCFC, the program CFG is analyzed to detect control-flow paths and lonely B-blocks. B-blocks in the control-flow paths are guarded using the path-asserting method. Lonely-blocks are instrumented with predecessor/successor-asserting checks. SCFC has been validated on a simple RISC processor for a set of commonly used biomedical workloads and three synthetic kernels. The results of our evaluation show significant overhead reduction compared to CCA. In respect to ACFC –the method with the lowest overheads– SCFC improves (on average) fault coverage by 17%, performance overhead by 5% and power-consumption by 3% with equal code-size overhead. It is important to note that the reported results are worst-case scenarios and in real systems the overheads will be considerably lower.

Acknowledgment

This work is supported by the STW Research organization. Many thanks are due to Bryan Olivier from ACE B.V. support team for his valuable help and Vlad M. Sima for his comments.

References

[1] W. Wolf, "Embedded systems and hardware-software co-design: Panacea or pandora box?" in *30th Conf. on Design Automation*, June 1993, p. 308.

[2] D. Zhu, "Energy management for real-time embedded systems with reliability requirements," in *Proceedings of International Conference Computer-Aided Design, ICCAD*, November 2006, pp. 528–534.

[3] 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.

[4] 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.

[5] U. Gunneflo, J. Karlsson, and J. Torin, "Evaluation of error detection schemes using fault injection by heavy-ion radiation," in *Proceedings of Nineteenth International Symposium on Fault-Tolerant Computing, 1989. FTCS-19*, June 1989, pp. 340–347.

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

[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] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] N. Oh, P. P. Shirvani, and E. McCluskey, "Error detection by duplicated instructions in super-scalar processors," in *IEEE Trans. on Reliability*, march 2002, pp. 63–75.

[15] 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.

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

[17] C. Strydis, C. Kachris, and G. N. Gaydadjiev, "Impbench: A novel benchmark suite for biomedical, microelectronic implants," in *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation, 2008. SAMOS 2008.*, July 2008, pp. 82–91.

[18] Synopsys processor designer. [Online]. Available: http://www.synopsys.com/Systems/BlockDesign/processorDev/Pages/default.aspx

[19] M. George and P. Alfke, "Linear feedback shift registers in virtex devices," www.xilinx.com, 2007.

[20] A. Parikh, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin, "Instruction scheduling based on energy and performance constraints," in *Proceedings of IEEE Computer Society Workshop on VLSI*, 2000, pp. 37–42.

[21] V. Tiwari and M. T.-C. Lee, "Power analysis of a 32-bit embedded microcontroller," in *Proceedings of Design Automation Conference, ASP-DAC '95/CHDL '95/VLSI '95.*, September 1995, pp. 141–148.