Generic Soft Error Data and Control Flow Error Detection by Instruction Duplication

Moslem Didehban, Hwisoo So*, Prudhvi Gali, Aviral Shrivastava, Kyoungwoo Lee*

Abstract—Transient faults or soft errors are considered one of the most daunting reliability challenges for microprocessors. Software solutions for soft error protection are attractive because they can provide flexible and effective error protection. For instance, nZDC [1] state-of-the-art instruction duplication error protection scheme achieves a high degree of error detection by verifying the results of memory write operations and utilizes an effective control-flow checking mechanism. However, nZDC control-flow checking mechanism is architecture-dependent and suffers from some vulnerability holes. In this work, we address these issues by substituting nZDC control-flow checking mechanism with a general (ISA-independent) scheme and propose two transformations, coarse-grained scheduling, and asymmetric control-flow signatures, for hard-to-detect control flow errors. Fault injection experiments on different hardware components of synthesizable Verilog description of an OpenRISC-based microprocessor reveal that the proposed transformation shows 85% less silent data corruptions compared to nZDC. In addition, programs protected by the proposed scheme run on average around 37% faster than nZDC-protected programs.

Index Terms—Reliability, Transient Faults, Soft Errors, Compiler, Silent Data Corruption.

1 Introduction

Advances in semiconductor technology have brought computer-based systems into virtually all aspects of human life. This unprecedented proliferation of semiconductor-based systems has significantly increased the extent of safety-critical applications i.e., applications with unacceptable consequences of failure. For instance, consider an object identification task in a semi-autonomous vehicle, which is responsible to detect obstacles in the road ahead and issuing emergency break requests after detecting close obstacles. Clearly, errors propagating to the output of such critical tasks can lead to a tragedy. In fact, errors affecting even a single pixel of an image may lead to wrong object classification [2]. Transient faults or soft errors caused by energetic particles, electromagnetic interference, or electrical noises are one of the main sources of hardware malfunctions in miniaturized microprocessors [3].

Traditionally, soft errors were mainly a concern for high-altitude applications (e.g., satellites and airplanes), and hardware redundancy-based solutions have been used to mitigate the problem of soft errors. Due to continued device scaling and higher integration, nowadays soft errors also cause reliability issues for terrestrial applications [4]. Although due to the temporary and untraceable nature of soft errors, it is difficult to precisely quantify their contribution to digital devices' failure rates, they have been considered a first-class suspect in many system-level failure scenarios [5, 6].

Redundancy is the main strategy to cope with the effect of soft errors. In general, the manifestation of errors can be detected by performing computations redundantly and checking the results frequently. Hardware-level error mitigation schemes utilize hardware redundancy and execute redundant computations on different hardware modules. Examples are ARM Cortex-R dual/triple core lock-step microprocessors [7, 8]. On the other hand, software-level solutions are based on the temporal redundant execution of computations on the underlying hardware. While protections offered by hardware-level error-tolerant solutions are hardwired to the hardware, protections offered by software-level solutions are flexible and can be adjusted based on application resilience requirements [1, 9, 10].

In-thread instruction replication is one of the most prominent software-level error protection solutions [1, 9, 11, 12, 13, 14]. These schemes assume that the microprocessor memory hierarchy (including TLBs, Caches, and memory) is ECC-protected (*Error Detection and Correction Code*) and focus on detecting errors on microprocessor core components. For instance, SWIFT [9] transformation replicates the execution of computational instructions, i.e. arithmetic and logical operations, and checks for errors by comparing the values of redundant register operands of **critical instructions**, i.e. memory and control-flow operations before their execution. Recent studies show the effectiveness of instruction replication-based schemes on the commercial off-the-shelf microprocessor in radiation-prone environments[15, 16].

However, most of the existing schemes can detect soft errors that only impact the execution of computational instructions and leave the execution of critical instructions unprotected. Our previous solution nZDC (near Zero silent Data Corruption [1]) improves the error coverage ability of the state-of-the-art instruction duplication-based solutions by protecting the execution of both critical and computational operations. nZDC transformation consists of data-flow and control-flow error detection parts. While nZDC data-flow error detection transformation is generic (can be

- *M. Didehban, P. Gali and A. Shrivastava are with Arizona State University, Tempe, AZ 85287, USA.*
- *H. So and K. Lee are with Yonsei University, Seoul.*

* *Co-corresponding authors*

Manuscript received April 30, 2018; revised August 30, 2018.

applied on all existing instruction set architectures) and effective (detects almost all data flow errors), its control-flow checking mechanism is limited. First, nZDC control-flow transformation is vulnerable to some errors causing unexpected jumps. Second, it relies on ISA support for certain conditional operations. If implemented on an architecture that does not support such operations (i.e. RISC-V[17]), new vulnerabilities will be exposed and its performance overhead increases considerably.

Targeting embedded safety-critical applications, in this work we propose generic-nZDC (gZDC), which is a combination of nZDC data-flow error detection mechanism with an inverted branch mechanism [18] and a generic control-flow error protection ([14]) solution. We also advise a novel coarse-grain main-redundant instruction scheduling policy and an asymmetric control-flow signature scheme to provide protection against hard-to-detect control-flow errors. To evaluate the effectiveness of our scheme, we performed around half a million transient fault injection experiments on a synthesizable Verilog description of an OpenRisc-based embedded microprocessor. Experimental results show that the gZDC-protected programs suffer from 85% less silent data corruption compared to nZDC-protected programs and their execution time is 37% faster.

2 Related Work

Software-level redundancy can be applied on different software abstract levels ranging from coarse-grained process-level [19, 20, 21] and thread-level [22, 23, 24, 25, 26] redundancy to fine-grained instruction-replication schemes [1, 9, 11, 12, 27]. In addition, several researchers [18, 28, 29, 30, 31, 32, 33] proposed hybrid (hardware and software) schemes. In this work, we focus on pure software-level fine-grained schemes because they can potentially provide a high degree of error coverage and such solutions do not require any modifications in the underlying operating system or microprocessor.

Instruction-level soft error detection schemes work based on inserting redundant assembly-level instructions in the original application's code and checking the results of main and redundant computations for error detection [1, 9, 11, 12, 14, 27]. Such solutions should make sure that the target application's data flow and control flow take place as expected. Data flow protection concerns the correct execution of memory write instructions while control flow protection focuses on executing the right memory write operations in the correct order.

2.1 Data-flow protection

Error **D**etection by **D**uplicated **I**nstructions (EDDI Oh et al. [27]) is one of the earliest work on the instruction duplication-based error detection area. EDDI partitions available architectural registers and memory address space into two halves and duplicates instructions during the compilation. For each main instruction, EDDI issues a shadow instruction with the same opcode as the original ones but uses a different set of registers. Duplicated memory (write/read) instructions perform memory operations redundantly on different memory locations. Error detection takes place by checking the results of store and branch instructions register operands immediately before their execution. Figure 1(b) shows the EDDI transformation corresponding to the original code shown in Figure 1(a)[1]. In the Figure, the main instructions are shown in bold. EDDI error checking instruction (Underlined instruction) compares the results of computations (value of `R4` and `R4*` registers) before writing them to the memory. If any error affects the execution of computational instructions (`ADDs and MULs`) and leads to a discrepancy between store value redundant registers (`R4 and R4*`) the error will be detected. However, since there is only one instance of store address register (`SP`) which is used in both redundant store instructions, the store address register is a single point of failure.

Reis et al. [9] proposed SWIFT (**S**oft**W**are **I**mplemented **F**ault Tolerance), which is composed of several optimizations on EDDI transformation. The authors of SWIFT eliminate the need for memory partitioning by arguing that the memory subsystem can be protected by ECC. As a result, SWIFT transformation (shown in Figure 1(c)) only duplicates arithmetic instructions and inserts checking operations for both value and address register operands of memory write instructions. To maintain an error-free and consistent input replication on memory load operations, SWIFT checks for errors in the address register operand of memory read instructions and copies the loaded value into the corresponding redundant register immediately after the execution of the load. Furthermore, many proposals also presume ECC-protected memory and improve the performance overhead of SWIFT by taking advantage of hardware-detected errors and rare hardware events (Shoestring[11]) or selective replication ([12, 35]).

In our previous work nZDC [1], we investigated the error detection capability provided by existing instruction duplication error detection (without memory replication) schemes. We observed that since such solutions do not replicate memory write instructions, any error directly affecting store operations remains undetected. In fact, since prior solutions use almost half of the user-available registers for redundant computations, they increase register allocator pressure to generate more spill (load and store from/to the stack) codes which leads to more vulnerable operations. To address this issue, we proposed loading the written value back from the memory and checking it against the redundant computed value (Figure 1(d)).

2.2 Control-flow protection

There are two types of control flow errors: (1) unwanted jump errors which cause unexpected jumps from an arbitrary location in the program's memory space to another. In this case the control-flow of the program changes in a way that is not permitted in its control flow (CF) graph. Examples of soft errors causing unwanted jumps are errors on PC/nPC registers, errors on target address fields of branch instructions, errors altering the opcode of a non-branch instruction to a branch/jump, and errors on the address field of a branch target address buffer structure. (2)

1. Note that to avoid confusion in the implementation of the EDDI scheme, we use the exact sample code provided in [34] paper written by the authors of the original EDDI paper.

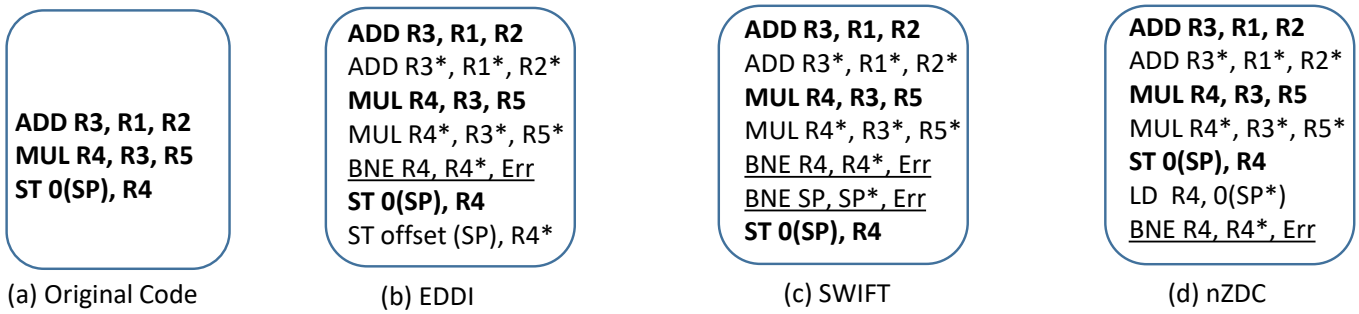

Fig. 1. Data-flow transformation for EDDI, SWIFT, and nZDC schemes. Main instructions are bold, and registers of redundant instructions are noted by an asterisk. EDDI only checks for errors in the value register of store operations (R4 and R4*) and duplicates memory subsystem. SWIFT transformation does not duplicate the memory subsystem and checks for errors in both address (SP and SP*) and value registers (R4 and R4*) of store instructions. nZDC transformation improves the coverage of SWIFT by checking for errors in the result of the store instruction.

Wrong direction control-flow errors which alter the direction of a branch, i.e., from taken to not-taken or vice-versa. These types of errors can be caused by errors affecting the computation of compare instruction register operands, the opcode of original compare and branch instructions, or even program status flag registers.

A popular solution for unwanted jumps is to embed some predefined signatures in the code and dynamically check their value. **C**ontrol-**F**low **C**hecking by **S**oftware **S**ignatures (CFCSS Oh et al. [36] is a well-known signature monitoring approach to prevent unwanted jump errors. The main idea of CFCSS is to assign unique values (called signatures) for each basic block and insert the signature updating instructions at the start of every basic block. This update is based on the signatures of the current basic block and the previous basic block. Errors that alter the control-flow and skip the execution of these signature updating operations will be detected if they cause a discrepancy between dynamically computed signatures and the statically assigned ones. However, recent investigations demonstrate that if applied without other error protection schemes, existing control flow detection techniques not only impose significant performance degradation but in many cases also increase the total program's execution vulnerability against soft errors ([37, 38, 39]).

SWIFT transformation adopts the concept of CFCSS with duplicated data-flow schemes. Instead of verifying the operands of comparison operations, it duplicates the compare instruction with shadow registers. While the result of the original compare instruction is used for the real branch instruction, SWIFT transformation utilizes the result of shadow compare instruction to update signature registers. If the result of shadow compare instruction leads to a taken branch, SWIFT updates the signature registers by the difference between the signatures assigned to the present basic block and the target basic block. If the branch instruction is not taken, SWIFT updates the signature registers regardless of the result of the shadow compare instruction. However, SWIFT transformation fails to detect wrong direction control flow errors that alter the direction of the branch from taken to not taken.

To protect the execution of control flow instructions and detect wrong-direction errors, nZDC scheme duplicates compare operations and double checks for the direction of branch instructions (Figure 2(b)). To accomplish compare operation duplication, nZDC transforms all *cmp* instructions to *subs* instructions (marked as Z3 and Z6 in Figure 2(b)). nZDC research targets ARM-v8 ISA which in *subs* operation behaves as same as *cmp* but also preserves the subtraction result. nZDC control-flow transformation assigns two specific registers named CDR and CCR for the result of the redundant *subs* instructions. nZDC does not simply compare CDR and CCR registers for error detection. Instead, it executes a series of conditional invert and xor operations for each of these control flow registers to detect errors affecting the execution of a branch operation. Furthermore, to detect the manifestation of unwanted jumps, nZDC control-flow assigns static signatures to each program basic blocks (#sigBB0, #sigBB1, and #sigBB2 in Figure 2(b)) and encodes (by performing Xor operation) the value of these static signatures in the computations of CDR and CCR registers. In a fault-free run and a taken branch case, CCR register first sets to the subtraction of compare instruction operands values (Z6), then its value gets xor-ed by the destination basic block register (Z7). And in the destination basic block, CCR register gets xor-ed by the destination basic block signature again. At this point, CCR value is equal to its value in the source basic block before being xor-ed. Then it gets xnor-ed by the CDR register that has the bitwise-inverted value of CCR register. Therefore, CCR register should always hold the value of zero before its next update. Otherwise, error detection flag will be raised (Z5). Note in a fault-free run at the beginning of each basic block the value of CDR register is equal to the inverted value of CCR and the result of their xnor (Z2) operation is always zero. The only difference in the case of a not taken branch is that CDR register gets xor-ed by the xor-ed value of the static signature of the taken basic block and not taken basic block.

2.2.1 Drawbacks of nZDC control-flow transformation

1) Vulnerability. In nZDC control flow transformation during the execution of each basic block body (shown by the vertical red line in Figure 2(b)) the value of the CCR register is Zero and the CDR register is dead. That implies any unwanted jump error that changes the control flow from inside the body of a basic block into the body of another basic block remains undetected by nZDC control flow error detection instruction (Z5 in Figure 2(b)) as far as the erro-

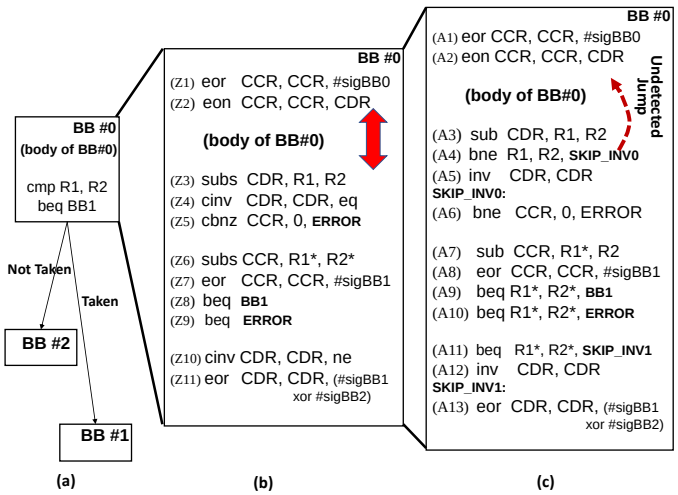

Fig. 2. (a) Original code consists of basic block computations and a compare and branch instruction, (b) nZDC control-flow transformation inserts 9 extra instructions to protect the execution of the original compare and branch instructions on an ISA with conditional instructions, and (c) nZDC control-flow protection requires 11 extra instructions on architecture without conditional codes (e.g. RISC-V) and also introduces new undetected errors.

neous jump doesn't cause any mismatch between original compare instruction main and shadow register operands.

2) Code Bloat. nZDC requires many extra instructions to protect one branch instruction. For each original compare/branch operation, nZDC control-flow transformation inserts 10 extra instructions only for control-flow error detection as shown in Figure 2(b).

3) Portability. nZDC requires conditional arithmetic or bit manipulation instructions, such as Z3 and Z6 in Figure 2(b). Therefore, architectures (like RISC-V[2]) that do not support conditional instructions cannot implement the original nZDC without additional transformation. Figure 2(c) shows nZDC control flow transformation on an architecture that does not support conditional codes. To implement the functionality of the conditional codes, two extra branches (A4 and A11) are introduced.

These extra instructions to port nZDC into architectures without conditional instructions exacerbate the vulnerability and code bloat problems of nZDC. First, extra instructions naturally exacerbate the inefficiency of nZDC control-flow protection since it requires extra instructions to perform the roles of the unsupported instructions. In addition, these instructions expose new vulnerabilities. For instance, consider an error on the functional unit responsible for effective address calculation of the first added branch instruction (A4). Assume the error causes an unwanted backward jump to the instruction A1 or A2 or anywhere in the body of the BB0 (dashed arrow in Figure 2(c)). Such error leads to extra computations and remains undetected. However, if we consider the same error on the branch instruction Z8 in Figure 2(b), nZDC detects the error since the update on the CCR register has happened before the erroneous branch and its value is not Zero.

2. RISC-V bitmanip extension [40] supported conditional move instructions before the ratification (0.9.4 version), but the version ratified in November 2021 (1.0.0 version) does not include the conditional move instructions.

3 Proposed Solution: gZDC

To improve nZDC error protection and its portability, in this work we introduce generic-nZDC (gZDC) which combines the most effective general control-flow error detection techniques with nZDC data-flow error detection strategy. gZDC also uses a coarse-grained main-redundant instruction scheduling strategy and asymmetric control flow signatures for hard-to-detect control flow errors.

3.1 Fault Model and Failure Mode

gZDC assumes that the microprocessor TLBs and memory subsystem is protected by ECC and aims to protect the rest of the microprocessor core components including microprocessor pipeline registers, register file, program counter register, functional units, and load-store unit.

In line with previous works [1, 9, 12, 27], the fault model under the consideration is transient a single-bit flip or a single event upset. As Sangchoolie et al. [41] demonstrated in most cases single-bit flip error detection also covers many multiple-bit errors. gZDC goal is to prevent an application from generating seemingly correct but wrong output or SDCs (Silent Data Corruption). Therefore, reducing the number of hardware detected failures, i.e., segmentation faults and exceptions, is not the goal of this work.

3.2 Wrong Direction Control Flow Errors

To detect wrong-direction control-flow errors, gZDC performs a redundant check on the direction of each conditional branch instruction. This is based on the concept of an inverted branch scheme [18] where redundant direction-checking branch instructions are added to the code. These branch-checking instructions use redundant registers, and their destination is always the error detection routine. The only difference is that against [18] we use replicated registers for branch-checking instructions for redundancy. Contingent upon the direction of the original branch instruction, the opcode of branch checking instructions (particularly branch condition) can be either equal or opposite to the original branch condition. Basically, there are two possible paths after each conditional branch instruction: (1) Taken path – when the branch condition is true and control of execution (PC) should be updated, and (2) Fall through path – when the branch condition is false and there will be no change in program control flow. Based on the possible outcomes of a conditional branch (taken or not taken) the opcode and the position of branch checking instructions are determined as follows:

Direction checking for taken branches: Placing branch-checking instructions after the conditional branches are taken is a waste because branch-checking instructions would not even get a chance to be executed. A naive solution can be placing branch checking instructions right at the beginning of the branch target basic block. Unfortunately, such a solution will lead to a false alarm when the branch target basic block is a merge basic block – it has more than one predecessor. For instance, consider the program shown

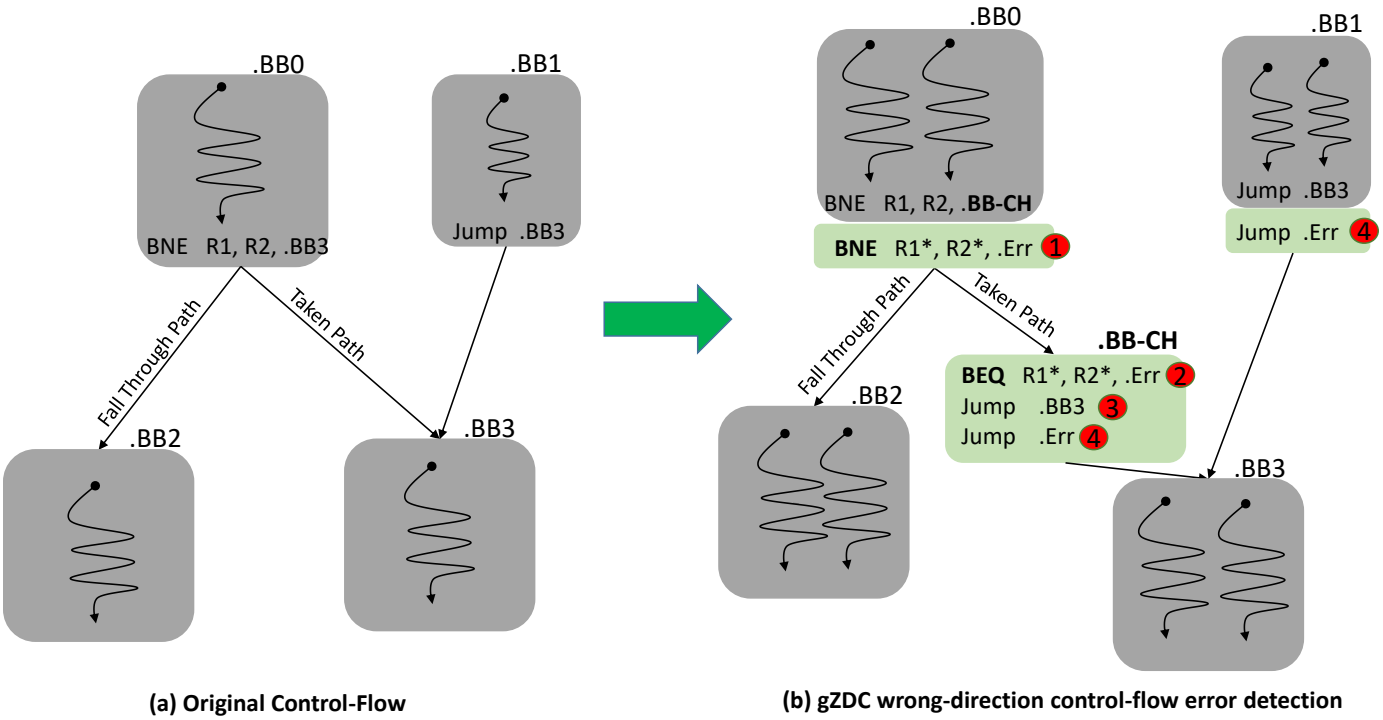

Fig. 3. gZDC inserts a branch direction check basic block between all control flow edges from a taken conditional branch to a merge basic block. The inserted BB is always composed of a branch direction-check instruction followed by two direct jump instructions.

in Figure 3(a). In the code, control can reach `.BB3` (target address of the conditional branch `BNE R1, R2, .BB3`) from either `.BB0` or `.BB1`. When control reaches the `.BB3` from `.BB0` the value of `R1` register is "not equal(NE)" to the value of `R2`. However, this condition may or may not be true if the control lands to `.BB3` from `.BB1` block. To address this issue, gZDC transformation first creates an intermediate block (`.BB-CH` in Figure 3(b)), then verifies the direction of the branch by executing branch checking instruction (marked as ② in the Figure), and finally transfers the control flow to the target basic block by inserting a direct jump instruction (marked as ③ in the Figure). The condition of branch checking instruction in the taken branch cases is against the original conditional branch instructions. For example, in this example, the opcode of branch checking instruction in the intermediate block `.BB-CH` is "BEQ (Branch Equal)" which is opposite to the "BNE (Branch Not Equal)" operation. The reason is that if the original branch is taken, the condition is true (`R1` is not equal to `R2` in our example) and the opposite condition should be false. Therefore, the branch checking instruction is always not taken and control will transfer to the destination block by the next direct jump. However, if an error influences the direction of the branch and alters its direction from not taken to taken, the control flow will reach the intermediate block wrongly. In those cases, the program control flow will be directed to the error handling block by branch-checking instruction because their condition is always opposite to the original error-free branch – the branch-checking instruction is taken because the original error-free branch was not taken.

Furthermore, to make sure that the actual jump takes place (control redirects to the destination BB), we insert a direct jump to the error detection block, after the original direct jump instruction (marked as ④ in Figure 3). This is required for the cases where an error alters the opcode of the jump instruction (marked as ③ in Figure 3) to another instruction and causes wrong-direction control flow error.

Direction checking for not taken branches: For the cases where the original branch is not taken and control flow falls through the basic block right after the branch, gZDC inserts a branch checking instruction with the exact same opcode (condition) immediately after the original branch[3]. Instruction marked as ① in Figure 3 is an example of branch checking instruction for not taken branches. The key point here is that if the original branch is not taken, the branch-checking instruction will be not taken as well and the program execution continues as expected. However, if an error alters the direction of the original branch from taken to not taken, the branch-checking instruction will direct the program's control flow to the error handling block. Note that all conditional branches in 3 can be changed to separate compare and branch instructions (for the cases where the underlying architecture only supports cmp/branch) without any effect on the proposed solution.

3.3 Unexpected Jumps

We divide unwanted jumps (defined in section 2.2) into intra-BB (unwanted jumps within a basic block) and inter-BB (unwanted jumps from one basic block to another) jumps. gZDC adopts different solutions to address each of these cases:

3. Technically, in this case, a new basic block will be inserted between the original basic block and the fall-through basic block.

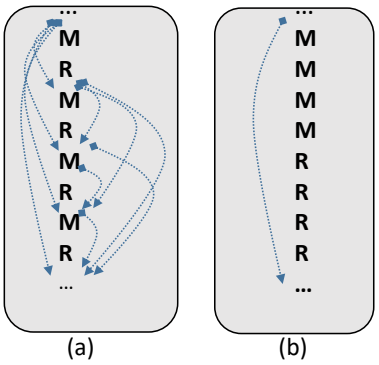

Fig. 4. Impact of fine-grained vs coarse-gained instruction scheduling on intra-BB undetected unwanted jumps. Main and Redundant instructions are shown by M and R letters respectively and arrows represent undetected intra-BB forward unwanted jumps. Part (a) shows a fine-grained instruction scheduling that leaves many unwanted jumps undetected because such jumps cause no mismatch between the state of redundant registers. Part (b) shows a coarse-grained scheduling policy that has a lesser chance of undetected unwanted jump errors.

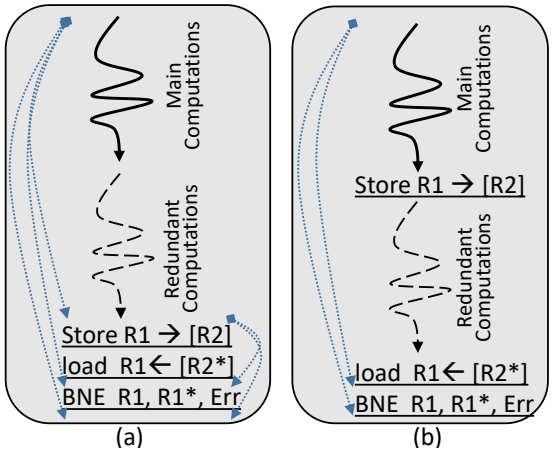

Fig. 5. Coarse-grained scheduling in the presence of store and checking operations. Part (a) shows that adding store operation and the corresponding checking operations at the end of the basic block introduces new possibilities for undetected unwanted jumps. Dashed arrows represent these undetected jumps. Part (b) shows gZDC coarse-grained main-redundant instruction scheduling policy.

Intra-BB unwanted jump detection: To detect the manifestation of intra-BB unwanted jumps, we introduce a novel static instruction scheduling scheme. Figure 4(a) illustrates a widely-used scheduling policy (used by [9] and [1] techniques) for scheduling main and redundant instructions. Such instruction scheduling (i.e., interleaving main and redundant operations one by one) is extremely vulnerable to unexpected short jumps.

We define **equal-point-of-execution** as program execution points[4] which at that point the value of all corresponding main and redundant registers are equal. Any unexpected jump from an *equal-point-of-execution* that skips (in a forward or backward direction) over the exact same number of main and redundant instructions will remain undetected. For instance, all program execution points before the execution of main instructions (represented by M in Figure 4(a)) are *equal-point-of-execution*[5]. Dashed arrows in Figure 4 represent forward unwanted jumps that cannot be detected by instruction duplication-based schemes because they do not cause any mismatch in the state of registers. Note that Figure 4 only shows forward undetected jumps, undetected backward jumps can be easily pictured by reversing the direction of arrows. Figure 4(b) shows an alternative scheduling policy (called coarse-grained scheduling of main and redundant instructions) which significantly reduces the chance of undetected jump errors. The main idea behind coarse-grained scheduling is that if an unexpected jump leads to a discrepancy between the state of main and redundant registers, most probably will be detected later by further data-flow error checking operations.

Furthermore, the presence of gZDC data-flow and control-flow error-checking operations restricts the window of the coarse-grained scheduling policy. For instance, Figure 5(a) shows a naive implementation of the proposed scheduling strategy for a basic block with a store instruction. Compared to Figure 4(b), the number of possible undetected unwanted jumps actually increases from 1 to 5. Now, jumps from the beginning of the basic block to the right before store or right before error checking instruction (`BNE R1, R1*, Err`) are also undetectable. In addition, jumps from right before the store to the end of the basic block or right before error-checking instructions also remain undetected. To mitigate this problem, we consider store instruction as a main and checking load operation and error checking instruction (`BNE R1, R1*, Err`) as redundant instructions. Figure 5(b) shows the gZDC instruction scheduling policy which reduces the number of undetected jumps from 5 to 2. Note that, we do not count a short jump skipping over the last two instructions in Figure 5(b), because such unwanted jumps are benign and do not change the functionality of the program. Note that if there is a dependency between a store and prior load operations inside a basic block (they access the same memory location), the redundant load should be inserted before the conflicting store. Similar to the store case, placing wrong direction control-flow checking operations at the end of basic blocks (Figure 6(a)) also introduces new vulnerable cases for unwanted jump errors. Figure 6(b) demonstrates gZDC wrong direction control-flow errors with coarse-grained scheduling policy. As shown, gZDC transformation first schedules the main stream of instructions followed by the conditional branch instruction. Then it inserts the corresponding redundant stream of instructions and direction-checking operations in both taken and fall-through paths.

Inter-BB unwanted jump detection: Coarse-grained main/redundant instruction scheduling not only reduces the chance of intra-BB unwanted jumps but also is effective against inter-BB unwanted jump errors. The reason is the drastic reduction in the number of *equal-point-of-execution* (i.e., the program points that *live* main and redundant registers have the same values). Generally, all unwanted (intra-

4. Program execution points are points between two consecutive instructions in a program.

5. Note that if redundant instructions are inserted into the code before the instruction scheduler phase in the compilation pipeline, the scheduling of main and redundant instructions may be different. Nevertheless, our definition of *equal-point-of-execution* is independent of scheduling policy.

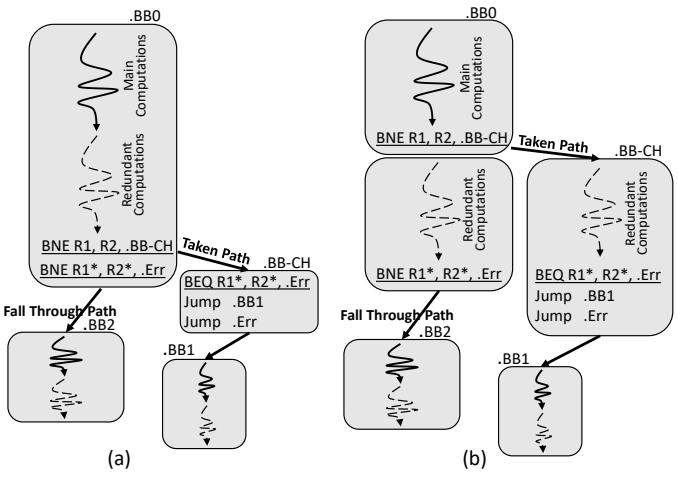

Fig. 6. Coarse-grained scheduling in the presence of conditional branch operations. (a) Naive scheduling by inserting the conditional branch at the end of the basic block. (b) gZDC wrong direction control flow and coarse-grained instruction scheduling.

/inter- basic block) jumps from program point *S* to *E* remain undetected if both *S* and *E* are *equal-point-of-executions* in instruction-replication schemes. That is because such unwanted jumps cause no discrepancy in the state of main and redundant registers and therefore there is no chance to be detected by error-checking instructions. The coarse-grained scheduling policy reduces the chance of undetected unwanted errors by simply reducing the number of *equal-point-of-execution* in instruction-duplication schemes.

To further reduce the possibility of undetected jumps, similar to existing signature-based control-flow checking schemes [42, 43] gZDC encodes static control-flow footprints (or signatures) into the program. And it checks their results before and after the system calls for error detection. gZDC transformation designates two specific general-purpose registers, called MICR (Main Instruction Check Register) and RICR (Redundant Instruction Check Register) for control-flow signature updating and checking. These two registers get updated during the execution of the program and their values are checked against each other for error detection purposes before and after system calls. Defining these two redundant registers imposes another condition to program *equal-point-of-executions*. Now, for a point of execution to satisfy the condition of *equal-point-of-executions* apart from equal value for all live main and redundant registers, MICR and RICR registers should also have the exact same values. The key point in gZDC unwanted jump detection strategy is **asymmetrical updates** for MICR and RICR registers which try to keep the values of MICR and RICR registers different as far as it is possible and therefore removes the *equal-point-of-executions*.

Listing 1 shows gZDC asymmetric control flow registers updating algorithm[6]. Algorithm 1 consists of three phases: 1) updating MICR, 2) updating RICR, and 3) eliminating symmetric updates. First, the compiler assigns a unique number called basic block signature to each program's basic

6. The redundant signature registers only carry the same value before system calls or at the end of the function and error detection operations also will be inserted both before/after system calls and at the end of functions.

Algorithm 1 gZDC asymmetric CF signature updating

Input: gZDC coarse-grained scheduled program
Output: gZDC coarse-grained scheduled program with unwanted jump error detection
Initialization : $MICR = RICR = 0$;
Assign unique number Sig_i to each main-instruction-included BB_i in Input
*/*First Phase: Updating MICR*/*
*/*Top-down control flow traversal*/*
for each BB_i in Input **do**
  if (BB_i is a main-instruction-included BB) **then**
    $lmInst$ = last main instruction in BB_i
    Create Instruction $Inst$: $MICR = MICR \oplus Sig_i$
    Insert $Inst$ after $lmInst$
  end if
end for
*/*Second Phase: Updating RICR*/*
*/*Top-down control flow traversal*/*
for each BB_i in Input **do**
  if (BB_i is a predecessor of a join BB) **then**
    $aggrSig$ = aggregated xor of BB_i signature with its all consecutive predecessors to the first join BB
    $frInst$ = first redundant instruction in BB_i
    Create Instruction $Inst$: $RICR = RICR \oplus aggrSig$
    insert operation $Inst$ before $frInst$
  end if
end for
*/*Third Phase: eliminating symmetric updates*/*
/ bottom-up control flow traversal*/*
for each BB_i in Input **do**
  if (BB_i includes $MICR$ and $RICR$ instructions) **then**
    $immMICR$ = immediate value of $MICR$ updating instruction in BB_i
    $immRICR$ = immediate value of $RICR$ updating instruction in BB_i
    if $immMICR == immRICR$ **then**
      $randomNum$ = a random number
      $immRICR = immRICR \oplus randomNum$
      for each $preBB_i$ of BB_i **do**
        if $preBB_i$ includes a $RICR$ updating instruction **then**
          $immRICR$ = immediate value of $RICR$ updating instruction in $preBB_i$
          $immRICR = immRICR \oplus randomNum$
        else
          $frInst$ = first redundant instruction in $preBB_i$
          Create Instruction $Inst$: $RICR = RICR \oplus randomNum$
          insert $Inst$ before $frInst$
        end if
      end for
    end if
  end if
end for

block that has at least one main instruction. Note that main instructions are arithmetic, logical or memory operations that use only main (not shadow) registers as their operands. Then it inserts instructions to update the value of the MICR register by xoring the value of the MICR register with the basic block signature right after the last main instruction in the basic blocks (lines 4-6). By the end of this step, each basic block that includes at least one main instruction includes one MICR-updating operation. In the second phase (lines 9-16), RICR-updating operations will be inserted only into BBs that are predecessors of join BBs[7]. These lazy updates on the RICR register lead to asymmetric (updates with different immediate values) on MICR and RICR registers which minimizes the number of *equal-point-of-executions*. To insert a RICR-updating instruction into a basic block, we first calculate the correct immediate value of xor operation which is an aggregated xor between all basic block signatures in a backward path from the current BB to the first join BB (line 11).

Figure 7 shows gZDC asymmetric control-flow signature updating algorithm for a simple case. As Figure 7(a) illustrates both `BB0` and `BB1` are join BBs. Applying gZDC wrong direction control flow error detection transformation (Figure 7(b)) increases the number of program BBs by three (`BB0_1`, `BB0_2`, and `BB0_CH` are added by gZDC transformation). This transformed control flow graph includes two basic blocks with main instruction (`BB0` and `BB0_1`) and two BBs with join successor ((`BB0_2` and `BB0_CH`)). According to phase one of the Algorithm 1, MICR-updating instructions are inserted into `BB0` and `BB0_1` right after the last main instruction. These instructions are shown as `MICR` $\oplus$ `sig_BB0` and `MICR` $\oplus$ `sig_BB0_1` in the Figure. Based on the second phase of the algorithm, RICR-updating operations are inserted into `BB0_2` and `BB0_CH`. To compute the offset of xor operation for `BB0_2`, we should traverse the control-flow graph backward to the last join BB (The dashed arrow in the Figure shows this path). Such a path consists of `BB0_2`, `BB0_1`, and `BB0`. However, since `BB0_2` does not include any main instruction, it does not have a signature and the aggregated signature is calculated by xoring the signature of `BB0_1` (`Sig_BB0_1`) and signature of `BB0` (`Sig_BB0`). The last instruction in `BB0_2` shows the inserted RICR-updating instruction. Similar to the `BB0_2`, a RICR-updating instruction is also inserted in the `BB0_CH`.

The last phase of Algorithm 1 (lines 17 to 36) deals with the problem of symmetric updates on both MICR and RICR registers. Symmetric updates lead to new *equal-point-of-executions* which alleviates the chance of undetected jumps. For instance, unwanted jumps from the beginning to the end of a BB which consists of main instructions followed by symmetric updates on MICR and RICR registers and redundant instructions remain undetected because such jumps do not cause any mismatch in the state of the pair main/redundant registers. To detect symmetric update cases, gZDC transformation goes over all program basic blocks starting from the end of the program and first checks if a basic block contains both MICR-updating and RICR-updating instructions (line 18). If yes, it extracts the

7. Basic blocks with more than one predecessor are join basic blocks.

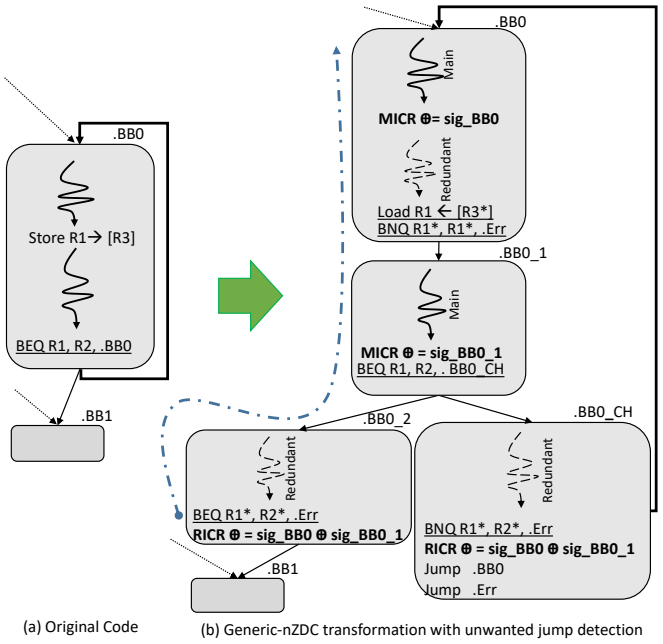

Fig. 7. Complete gZDC data-flow and control-flow transformations for a simple loop. (a) shows control-flow for a simple loop, and (b) shows corresponding gZDC code with static signature updating operations. MICR-updating instructions are inserted into the main-instruction-included-BB and RICR-updating instructions are inserted into successor BBs of a join BBs. The dashed arrow shows the backward trace path to compute the aggregated signature required for RICR-updating instruction in $BB0\_2$.

immediate field of signature updating instructions to see if those values are the same (lines 18-21). If yes, then it generates a random number and updates the immediate field of RICR-updating instruction (lines 2-23). This makes signature updating instructions asymmetric. To maintain consistency, signature modification for predecessor BBs of the current BB is required (lines 24 to 33). For that purpose, the generated random number should be xor-ed with the RICR-updating instruction in the predecessor blocks. If the predecessor block does not include a RICR-updating instruction, the algorithm creates one and inserts it into the basic block before its first redundant instruction (lines 29 and 31).

4 Experimental Methodology

In this section, we describe our experimental environment and evaluation strategy to measure the error coverage and performance overhead of the SWIFT, nZDC, and gZDC transformations. Note that for nZDC control-flow transformation we use an equivalent implementation (similar to Figure 2(c)).

4.1 Microprocessor and Fault injection Environment

We used the single-issue 5-stage pipeline Mor1kx cappuccino microprocessor (the last version of OpenRISC1000 processor family) which is capable of running the Linux operating system [44]. The microprocessor configuration is shown in table 1. We simulated the synthesizable HDL Verilog codes of Mor1kx microprocessor by Icarus Verilog simulator [45].

TABLE 1
Mor1kx microprocessor configuration

Parameter	Value
ISA	OpenRISC
CPU model	In-order Single-issue Mor1kx
Pipeline	5 stage (cappuccino)
# of General Purpose Registers	32
Branch prediction	BTFN
L1 D/I cache	8KB(2 way)/8KB(2 way)
Cache line size	32 bytes
D/I Cache Latency	Miss(20 cycle)/Hit(1 cycle)
RAM size	32 MB

TABLE 2
Fault Injection Features

Component	Fault Model	# of fault sites
Register File	Single Event Upset[a]	1024
Fetch/Decode Unit	Single Event Upset	200
Decode/Execute Unit	Single Event Upset	216
Execute/Control Unit	Single Event Upset	183
Write/Back Unit	Single Event Upset	32
ALU	Single-event transient[b]	36
LSU	Single-event transient	101

[a] Errors are injected on flip-flops.
[b] Errors are injected on combinational circuits.

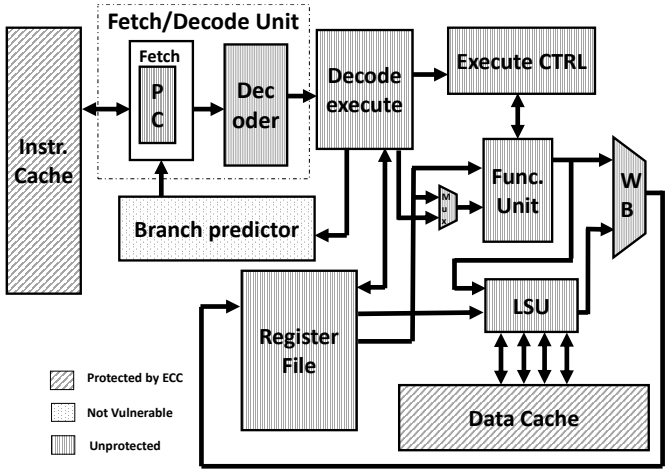

Fig. 8. Mor1kx architecture. Caches and Branch predictor are excluded from fault injection analysis.

Figure 8 illustrates the high-level block diagram of the mor1kx microprocessor. The microprocessor core components include instruction and memory caches, a simple branch predictor, a register file, a processor data path, and a load-store unit. Note that for simplicity, forwarding paths from ALU to Decode/Execute and from WriteBack to Decode/Execute stages are not shown in the Figure. In line with previous works [1, 9, 11, 12, 13, 46], we assume that instruction/Data caches are ECC-protected and are excluded from our fault injection sites (these components are highlighted as protected by ECC in the Figure). We also exclude branch predictor because as pinpointed by Mukherjee et al. [47], transient errors on such performance-enhancing structures do not cause lead to a failure. We injected single bit-flip faults on the rest of the microprocessor hardware components which are Fetch/Decode, Decode/Execute, Execute Control, Register File, ALU, LSU, and WriteBack. Table 2 summarizes the number of fault sites and the injected fault model in fault injection target components.

4.2 Compilation and Binary Generation

We utilize clang and LLVM-or1k [48] compiler infrastructures and compile 8 programs from the Mibench [49] test suite and a 10x10 matrix multiplication for mor1kx microprocessor with -O3 optimization level. Note that we compile the whole application as appeared in the Mibench test suite and did not reduce programs to micro-benchmarks. We choose to implement error detection transformations as late back-end passes (right before assembly code emission) in the LLVM-or1k compiler. The reason is to take advantage of all standard compiler optimizations like dead code elimination and common-subexpression elimination and to prevent optimizations from removing redundant and error-checking codes. We only apply protection schemes to user-level functions and exclude standard library functions from all evaluations including performance overhead and error coverage estimation. However, to expand the domain of evaluation for the benchmarks which spend most of their execution time in the library calls (e.g., qsort program heavily uses library provided qsort function), we manually copy the source code of the dominant library function call to the program source code from open-source GNU C (glibc) library and apply protection to them. We generate four executable versions for each benchmark:

Original (ORG) version is the unprotected version of the program without applying any software-level protection scheme.

SWIFT version is protected by SWIFT data and control flow protection transformations. Note that we include SWIFT signature-based control-flow error detection in this version.

nZDC version is protected by nZDC data-flow transformation (shown in Figure 1(d)) and functionally-equivalent control-flow error detection (shown in Figure 2(c)).[8]

gZDC-WithoutJumpDet version is protected by nZDC data-flow transformation (shown in Figure 1(d)) and generic wrong direction control-flow error detection (explained in Subsection 3.2).

gZDC version is **gZDC-WithoutJumpDet** plus coarse-grained main-redundant instruction scheduling policy and asymmetric execution control-flow footprints (Algorithm 1). We develop and evaluate gZDC and gZDC-WithoutJumpDet separately to show the effectiveness of the coarse-gain scheduling and asymmetric control-flow signature techniques on error detection and their implications on performance overhead.

4.3 Fault Injection Process and Output Classification

For each fault injection experiment, we randomly select a fault injection site, `b`, and a fault injection time, `t` among all 1792 fault injection sites and all cycles that a program executes user-level functions. Then we start the simulation

8. Although OpenRISC ISA supports cmov (conditional move) instruction, we did not use cmov operations in our implementation since our goal here is to evaluate the effectiveness of SWIFT and nZDC on embedded architectures that do not support conditional codes like basic RISC-V or Xtensa ISA.

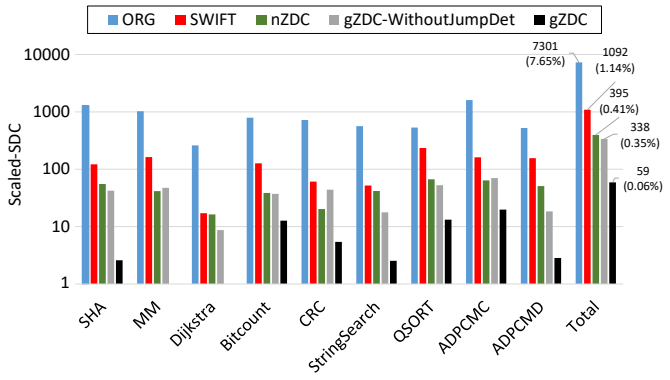

Fig. 9. gZDC transformation reduces the number of scaled-SDCs from 7,301 to 59 compared to the original code, more than a hundredfold improvement.

normally and once the execution reaches cycle `t`, we invert the logical value of `b` (by XORing it with '1') for one cycle. We let the execution continues till execution terminates permanently or the execution time exceeds from allowable simulation time which is 2 times more than the fault-free run of the program. We classify the result of each fault injection trial into one of the following categories:

Masked: Fault injection simulation terminates normally and generates the exact same output as a fault-free run.

SW Detected: Cases where the protection scheme detects the manifestation of error and raises the error detection flag.

OS/HW Detected: Fault injection simulation runs that terminate permanently by generating an exception (i.e. segmentation faults or unknown instruction exception) or cause a time-out error.

Silent Data Corruption (SDCs): Cases where execution terminates normally, but the results are different from the fault-free execution.

For each version of a program, we conducted 10,600 random fault injection experiments on hardware components listed in table 2, which statistically provides around a 2.5% error margin for a 99% confidence level [50]. The number of injected errors in each hardware component depends on the number of flip-flops on that component. For instance, since the register file includes around 57% of targeted fault sites (1024 of 1792), around 57% (around 6,000) of all injected errors also were injected in the register file. Since we have 9 benchmarks, we injected 95,400 (10,600 * 9) faults for each version of the programs. Overall, for all versions of programs, we injected 477,000 (5 * 95400) fault injection experiments.

Since all error detection schemes including SWIFT and gZDC transformations presume some type of backward recovery (i.e. restarting or checkpoint/rollback [51]) as a post error detection handling strategy, errors in the `Exceptions and Hangs` category can be considered harmless. The only difference between `Exceptions and Hangs` and `Detected` errors is that the former is identified by operating system or hardware protection schemes and later is recognized by software-level error protection schemes. Nevertheless, in both cases, the recovery routine should be invoked to remove the manifestation of error from the system. Similar to many prior research [9, 11, 12, 14], we consider SDC-induced faults as failed cases because such errors remain unnoticed.

4.4 Number of scaled SDCs as Comparison Metric

To fairly quantify and compare the error detection capability of SWIFT, nZDC, gZDC-WithoutJumpDet, and gZDC techniques, we use the number of scaled SDCs (or scaled-SDC for short). This metric was introduced in [52] and is calculated based on an overhead-dependent correction factor. This metric has been used in several recent works [1, 26, 53, 54] and captures the negative impact of runtime overhead (α) of protection schemes on execution reliability. Scaled-SDC is estimated as below:

$$scaled - SDC = \#ofSDCs \times \alpha \quad (1)$$

Since there is no runtime overhead for the original version of programs the number of scaled-SDCs is in fact equal to the number of SDCs. As pinpointed by [52] considering the number of SDCs (or percentage of SDCs) significantly overestimates the error coverage capability of the protection schemes and leads to the wrong conclusion.

4.5 Error Coverage Results and Analysis

Processor-wide error coverage: Figure 9 shows the number of scaled SDCs (calculated based on equation 1) for different benchmarks extracted from our microprocessor-wide fault injection experiments. In the Figure, X-axis represents different benchmarks and Y-axis represents the number of scaled-SDCs on a logarithmic scale. The rightmost set of bars (denoted as total) represents the aggregated number of scaled-SDCs for each version of programs across all benchmarks. This aggregated sum can be seen as a large application that consists of all the benchmarks. As shown in Figure, from 95,400 fault injection experiments conducted on the original version of the programs around 7.65% of them result in SDCs. SWIFT transformation reduces the number of scaled-SDC to 1.14%. nZDC transformation reduces the number of SDCs by around 95% compared to SWIFT transformation. The number of scaled-SDCs in gZDC-WithoutJumpDet scheme is very close to nZDC. Finally, gZDC improves the scaled-SDC rate to only around 0.06%, which is less than 1/6 of nZDC. These processor-wide results show the effectiveness of coarse-grained main-redundant instruction scheduling policy and asymmetric execution control-flow footprints (Algorithm 1) in the reduction of the number of SDCs.

Component-wise error coverage: Figure 10 shows the aggregated number of scaled-SDCs per component for different versions of programs across all benchmarks. We start by looking at the register file since around 60% (4411 of 7301) of all SDCs in original programs (Figure 10(a)) were caused by faults injected on the register file. We can see that SWIFT transformation is effective in protecting the register file and it reduces the scaled-SDCs to only 474 cases – almost 90% reduction of scaled-SDCs for the register file. Further examination (recreating the SDC cases and tracing the trajectory of inserted faults to the final output) shows that almost in all of the SDC cases of SWIFT-protected programs injected fault alters the value of a dead register

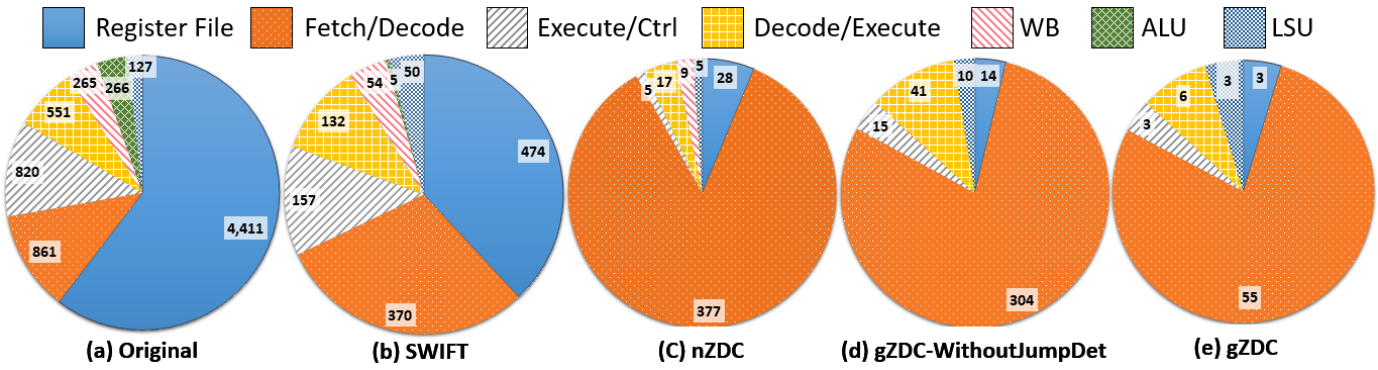

Fig. 10. Component-wise scaled SDC analysis. While instruction duplication schemes can effectively improve the register file vulnerability, errors affecting the fetch/decode stage of the pipeline remain the main source of SDCs.

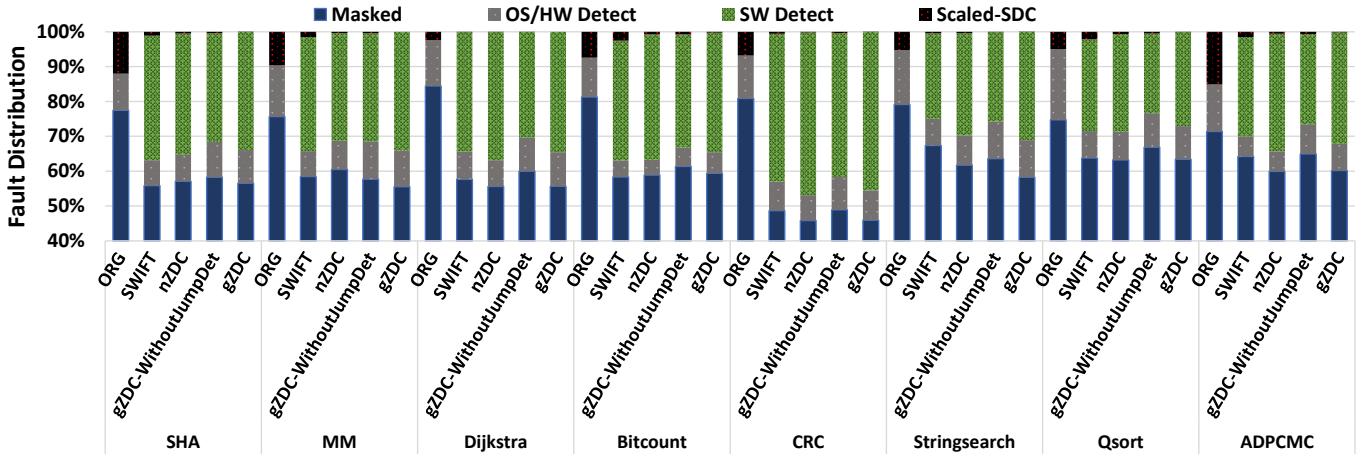

Fig. 11. Error Distribution.

operand[9] of memory instructions immediately before actual use (read) of register by the instruction. We have realized that in some cases error happens even before checking instruction, but since the check instruction gets its value from pipeline forwarding passes (not from the register file) the error remains undetected. Furthermore, since SWIFT transformation cannot detect unwanted taken to not taken branch direction changes, any corruptions of the register file that alter the branch direction from taken to not taken can result in SDCs without being detected when the corrupted registers are overwritten after the branch instruction.

Consequently, although SWIFT reduces almost 90% scaled-SDCs compared to the unprotected version on the register file, the register file is still the biggest vulnerable hardware component for SWIFT-protected schemes as shown in Figure 10(b). On the other hand, All nZDC-based transformations (Figure 10(c), (d), and (e)) considerably reduce the register file scaled-SDCs rate. That is because they detect errors by checking registers after the execution of critical instructions (i.e., memory and control-flow operations) rather than before their execution. However, there are still a few SDC cases for nZDC-based transformations and our tracking reveals that in all SDCs scenarios errors hit the address register of silent stores.

9. A register operand is dead if the next access to that register is a write operation.

As Figure 10 reveals bits in the fetch/decode stage of the pipeline are the most challenging parts of hardware to protect against errors by software-level redundancy schemes as faults injected on them cause the majority of failed cases almost in all protected versions of the programs. While the number of scaled-SDC due to error injection on fetch/decode stage flip-flops for the original version of programs is around 861, SWIFT, nZDC, and gZDC-WithoutJumpDet transformations can only improve such failures by slightly more than 65%. The main reason is that soft errors affecting several components in the fetch/decode stage of the pipeline frequently induce unwanted jumps. Examples of these components are the fetch/decode stage including PC (32 fault injection site), next PC (32 fault injection site) and instruction address bus register (32 fault sites), instruction address bus register (32 fault sites), and decoded instruction (32 fault injection sites). However, gZDC transformation can significantly mitigate the SDC rate of the fetch/decode stage by more than 94%. This shows that unwanted jump detection transformations proposed in section 3.3 are really effective.

Bits in the Decode/Execute stage of the pipeline show similar behavior to the fetch/decode stage in terms of the effectiveness of the protection scheme. We can see, while around 551 SDCs were caused due to fault injection on Decode/Execute bits in the original version of programs,

SWIFT transformation can only reduce the number of scaled-SDCs to 106 (around 75% improvement). On the other hand, nZDC-based schemes are much more effective against errors in the Decode/Execute stage of the pipeline. The reason for this is because as demonstrated in Figure 1 the SWIFT transformation leaves the execution of memory write operations unprotected while they are in microprocessor pipeline stages. Note that while nZDC and gZDC-WithoutJumpDet transformations are more effective than SWIFT, they cannot completely remove the SDC rate for Decode/Execute stage component. We investigated some of the failed cases and realized that errors in some registers in these components (i.e. 32 bit immediate address register) can cause unwanted jumps while being used by branch/jump operations. As Figure 10 shows the gZDC transformation reduces the decode/execute stage SDCs rate by 65% and 85% compared to nZDC and gZDC-WithoutJumpDet transformations, respectively. For the Decode/Execute stage of the pipeline, nZDC is more effective than gZDC-WithoutJumpDet transformation because it is equipped with an unwanted jumps detection solution while gZDC-WithoutJumpDet does not have any support for unwanted jump error detection. Execute/Ctrl errors also follow the same trend as Decode/Execute stage.

The Writeback stage of the pipeline is responsible to select the source of register file updates (functional unit in cases of arithmetic instructions or memory in case of load operations). Many of the faults affecting this unit (around 14%) cause SDC in the original version of programs. Although sensitive to errors, since the area of the writeback stage is small the chance of faults hitting its circuitry is low, and therefore its contribution to processor-wide vulnerability is small. While nZDC-based transformations can significantly reduce the writeback stage vulnerability (by ∼98%), SWIFT transformation improves its reliability by ∼80% (from 265 SDCs to 33).

All four versions of instruction duplication schemes can effectively protect execution against single-event transient faults impacting ALU calculation results. Only a few errors cause SDC in SWIFT-protected programs and in those cases error hits the computations of the memory address of load operations. For the load store unit (LSU), SWIFT transformation reduces the number of scaled SDCs from 127 to 40. For nZDC-based transformations, the scaled number of SDCs for LSU is in order of ∼10 (around 1/5 compared to SWIFT) which shows the effectiveness of the load-back checking error detection strategy. Nevertheless, they are still SDC cases and our investigation attributes them to silent stores (store value is already in the store memory address). As we highlighted in [14], errors affecting the address part of a silent store cannot be detected by nZDC data flow checking solution.

Impact of protection schemes on fault distribution: Figure 11 shows the impact of instruction duplication schemes on error propagation and distribution. In the Figure, the x-axis shows different versions of benchmarks and the y-axis represents the percentage of masked, OS/HW Detected, SW Detected, and scaled-SDCs. Across all benchmarks, both SWIFT and nZDC-based transformations (nZDC, gZDC-WithoutJumpDet, and gZDC) considerably reduce the number of masked errors (the lowest segment of each bar in Figure 11) by on average around 20%. Similarly, the number of OS/HW detected faults also decrease on an average by half (from 14% to 7% for SWIFT and to around 9% for gZDC-WithoutJumpDet and gZDC). Since the main goal of instruction duplication error protection schemes is to prevent a program from producing wrong results, an ideal scheme should only detect faults that are going to alter the final program output. However, frequent error checks of instruction duplication-based schemes introduce some false alarms by detecting benign faults (faults that are going to be masked) as well as overprotection by detecting errors that are covered by OS/HW error protection schemes. To estimate false alarms and overprotection of applied error protection schemes, we can analyze the difference between SW-detected errors of protected programs and the percentage of SDCs for the original version of programs. While only on an average 8% of injected faults lead to SDCs in original versions of programs, error checks in SWIFT, nZDC, gZDC-WithoutJumpDet and gZDC raise the error detection flag on average around 31.1%, 34%, 30.2%, and 34.17% of times, respectively.

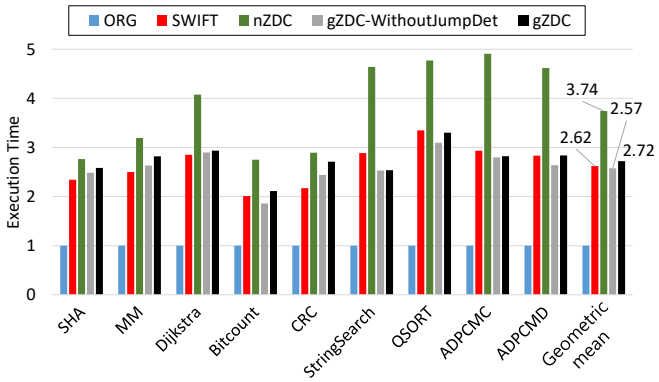

Fig. 12. Due to nZDC complex control-flow detection strategy, the execution time overhead of nZDC is considerably higher than other solutions.

4.6 Performance Overhead

Figure 12 depicts the execution time overhead of SWIFT, nZDC, gZDC-WithoutJumpDet, and gZDC transformations which is on a geometric mean around 2.62x, 3.74x, 2.60x, and 2.75x, respectively. From this imposed overhead, around 2.4x is because of the data-flow error detection part which includes instruction duplication and execution of frequent error detection/checking instructions. The rest is because of control-flow detection related computations. Note that since we tested the benchmarks on a small microprocessor (Mor1kx), this performance result might overestimate the performance overhead of the protection schemes. This is because the performance overhead of each protection scheme can be potentially improved by hardware optimization in high-performance processors such as macro-op fusion [55].

To measure the performance overhead imposed by the lack of conditional instruction for SWIFT and nZDC transformations (Figure 2 (c)), we also implemented SWIFT and nZDC with the conditional move instruction that is supported by OpenRISC. On a geometric mean, the runtime

TABLE 3
Overhead of control-flow protection based on the number of extra instructions

		SWIFT	nZDC	gZDC-Without JumpDet	gZDC
Basic block without Branch		2 or 4*	7	0	1 ∼2 + #ofStore
Basic block with Branch	Taken	4 or 6**	11	3	4 ∼5 + #ofStore
	Not taken	4 or 6**	14	2	3 ∼4 + #ofStore

*Requires 4 instructions for the basic blocks with store
**Requires 6 instructions for the basic blocks with store

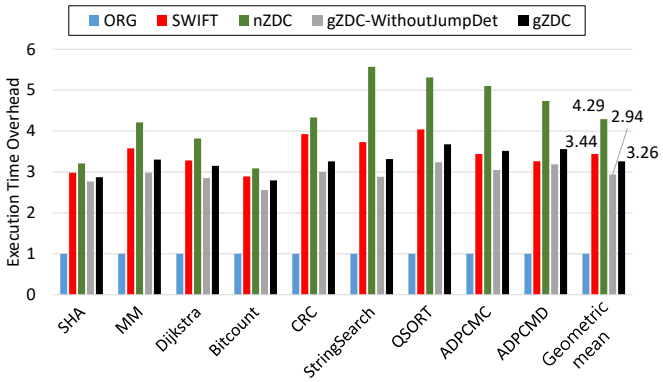

Fig. 13. gZDC code size overhead is less than SWIFT and nZDC.

overhead decreases from 2.62x to 2.53x and from 3.74x to 3.61x for SWIFT and nZDC transformations, respectively.

nZDC transformation suffers from significantly higher performance degradation due to its complex and lengthy control flow error detection strategy. Table 3 shows the number of dynamic instructions required by the control-flow protection part of the transformations described in Section 4.1. The first row shows the number of extra instructions for the basic blocks with only one successor. The second and third rows show the number of extra instructions in cases where the last operation in the basic block is a branch. nZDC control flow transformation requires considerably more instructions for all three cases. For instance, while SWIFT transformation requires only 2 extra instructions for storeless basic blocks (4 extra instructions for basic blocks with store instructions), nZDC transformation always adds 7 extra instructions to update and verify the value of CCR and CDR registers. The burden of the extra instructions becomes worse for the branch protection in nZDC transformation. Due to the code bloat issue discussed in Section 2.2.1, nZDC requires up to 14 dynamic instructions to protect compare and branch instructions.[10] Even with the support of ISA-dependent instructions such as cinv and subs in Figure 2 (b), nZDC still requires up to 10 extra instructions.

The performance overhead of nZDC is similar to other techniques for SHA and CRC benchmarks. This is because the average size of basic blocks is relatively big and therefore the protection overhead is dominated by instruction duplication and data-flow error checking operations. On the other hand, nZDC shows significant performance overhead for some benchmarks such as STRINGSEARCH and ADPCMC(D) because these programs have many small basic blocks where the nZDC control-flow protection overhead is larger than its data-flow protection. We observed that for several basic blocks, the number of extra instructions required for control-flow protection is larger than the number of original and shadow instructions. On the other hand, gZDC-WithoutJumpDet transformation requires a minimum number of extra instructions (up to 3) for control flow protection. These extra instructions include shadow compare instructions and additional branch instructions that are illustrated in Figure 6 (b). Note that the "Jump .Err" instruction in Figure 6(b) will not be executed in the absence of errors.

In addition to the wrong direction control flow error detection of gZDC-WithoutJumpDet, gZDC transformation also applies coarse-grained master-shadow instruction scheduling that does not increase the dynamic number of instructions, and asymmetric control-flow footprints to update MICR and RICR registers. While updating the RICR register can be skipped with lazy updates, MICR should be updated (1 + number of store operations in the basic block) times for one original basic block since store operation splits the basic block as shown in Figure 7. Overall, while the runtime overhead of gZDC transformation is only 6% more than gZDC-WithoutJumpDet, it achieves significantly better error coverage which proves the effectiveness of the coarse-grained master-shadow instruction scheduling and asymmetric control-flow signatures.

Since SWIFT, nZDC, and gZDC do not divide the data memory, they do not incur considerable data memory overhead. On the other hand, all of the instruction replication schemes inevitably incur code size overhead due to the additional instructions . Figure 13 shows the code size overhead for SWIFT, nZDC, gZDC-WithoutJumpDet, and gZDC. On a geometric mean, SWIFT, nZDC, gZDC-WithoutJumpDet, and gZDC show 3.4x, 4.3x, 2.9x, and 3.3x code size increases, respectively. Remarkably, gZDC shows the smallest average code size among the tested protection schemes.

5 Limitations of gZDC

Multi-threaded workloads. gZDC transformation issues redundant loads to the exact same memory location and expects that they receive the same value in fault-free runs. However, since that assumption does not hold for threaded applications with shared data, then gZDC transformation will cause a false alarm in such environments. Similarly, gZDC uses the load-back checking strategy for store operations assuming consecutive stores and loads from the exact same memory location see the same value in fault-free execution. However, that condition is not true in multi-threaded environments.

Undetected errors. As shown in Figure 9 and Figure 10, gZDC shows 59 scaled-SDCs (22 raw SDCs). We analyzed the SDC failures in gZDC to identify the reason for undetected failures. The undetectable soft errors in gZDC

10. Since some instructions such as A2 and A4 in Figure 2 (c) should be implemented with multiple instructions in OpenRISC, the number of extra instructions is higher than the one in Figure 2 (c).

are 1) errors affecting the address computations of silent stores[14]. That is because a silent store can wrongly update an arbitrary location of memory but the issued load after the store still seems the correct value. 2) If an error changes the opcode of silent operations (operations that do not cause any update in the state of architectural registers except PC) to memory write, the error remains undetected. Note that this is a limitation of all existing single-memory instruction duplication schemes. 3) while the proposed unexpected jumps are effective, still there can be cases where an unwanted jump error remains undetected i.e., jumps from one **equal-point-of-execution** to another.

Permanent faults. Since gZDC executes both main and redundant streams on one processor, the original instruction and the corresponding shadow instruction usually be executed in the same execution paths. Therefore, if there is a permanent fault, also known as a hard error, in the execution path of the processor and both the original and shadow instructions are executed with the faulty component, gZDC cannot detect such a fault since the original and shadow stream will be corrupted identically. To detect permanent faults, gZDC relies on additional solutions such as BIST (built-in self-test). Similarly, semi-permanent faults cannot be detected by gZDC if the fault occurs before the execution of original instructions and persists till the execution of the corresponding shadow instructions.

External memory accesses and peripheral registers. Since gZDC assumes that redundant loads will receive the same values in fault-free execution, its data-flow protection should not be adopted for external memory accesses or peripheral registers that their state might be changed by other sources. Solutions similar to SeRoHAL [35] can be used to enable error detection on such part of the execution.

6 Conclusion

We present a generic and effective software-level instruction duplication error detection scheme. The proposed scheme comprises three main error detection optimizations: i) data-flow error detection on the results of store instructions, ii) wrong-direction control-flow error detection by inserting redundant branch and compare operations, and iii) coarse-grained main-redundant instruction scheduling and asymmetric control-flow signatures for unexpected jump error detection. We evaluated the effectiveness of the proposed scheme by performing extensive microprocessor-wide Verilog-level fault injection experiments. Results show that the proposed scheme reduces the rate of silent data corruption errors by around 2 orders of magnitude.

Acknowledgments

This work was partially supported by funding from National Science Foundation Grants No. CPS 1646235, CCF 1723476 - the NSF/Intel joint research center for Computer Assisted Programming for Heterogeneous Architectures (CAPA); by National Research Foundation of Korea (NRF) grant funded by the Korea government(MSIT) (No. RS-2022-00165225).

References

[1] M. Didehban and A. Shrivastava, "nZDC: A compiler technique for near Zero Silent Data Corruption," in *Design Automation Conference (DAC), 2016 53nd ACM/EDAC/IEEE*. IEEE, 2016, pp. 1–6.

[2] G. Li, S. K. S. Hari, M. Sullivan, T. Tsai, K. Pattabiraman, J. Emer, and S. W. Keckler, "Understanding error propagation in deep learning neural network (DNN) accelerators and applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2017, p. 8.

[3] R. C. Baumann, "Soft errors in advanced semiconductor devices-part I: the three radiation sources," *IEEE Transactions on device and materials reliability*, vol. 1, no. 1, pp. 17–22, 2001.

[4] E. Ibe, H. Taniguchi, Y. Yahagi, K.-i. Shimbo, and T. Toba, "Impact of scaling on neutron-induced soft error in SRAMs from a 250 nm to a 22 nm design rule," *IEEE Transactions on Electron Devices*, vol. 57, no. 7, pp. 1527–1538, 2010.

[5] A. Haggag, N. Sumikawa, and A. Shaukat, "Reliability/yield trade-off in mitigating "no trouble found" field returns," in *On-Line Testing Symposium (IOLTS), 2015 IEEE 21st International*. IEEE, 2015, pp. 174–175.

[6] A. Haggag, N. Sumikawa, A. Shaukat, J. J. Lee, N. Aghel, and C. Slayman, "Mitigating "No trouble found" component returns," in *Reliability Physics Symposium (IRPS), 2015 IEEE International*. IEEE, 2015, pp. 3C–5.

[7] X. Iturbe, B. Venu, E. Ozer, and S. Das, "A Triple Core Lock-Step (TCLS) ARM® Cortex®-R5 Processor for Safety-Critical and Ultra-Reliable Applications," in *Dependable Systems and Networks Workshop, 2016 46th Annual IEEE/IFIP International Conference on*. IEEE, 2016, pp. 246–249.

[8] W. Lyons, "Enabling increased safety with fault robustness in microcontroller applications," *ARM Corporation*, 2010.

[9] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "SWIFT: Software implemented fault tolerance," in *Proceedings of the international symposium on Code generation and optimization*. IEEE Computer Society, 2005, pp. 243–254.

[10] A. Shrivastava and M. Didehban, "Software approaches for in-time resilience," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2019, pp. 1–4.

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

[12] I. Laguna, M. Schulz, D. F. Richards, J. Calhoun, and L. Olson, "IPAS: Intelligent protection against silent output corruption in scientific applications," in *Code Generation and Optimization (CGO), 2016 IEEE/ACM International Symposium on*. IEEE, 2016, pp. 227–238.

[13] M. Didehban, S. R. D. Lokam, and A. Shrivastava, "InCheck: An in-application recovery scheme for soft errors," in *Design Automation Conference (DAC), 2017 54th ACM/EDAC/IEEE*. IEEE, 2017, pp. 1–6.

[14] M. Didehban, A. Shrivastava, and S. R. D. Lokam, "NEMESIS: A software approach for computing in presence of soft errors," in *Computer-Aided Design (ICCAD), 2017 IEEE/ACM International Conference on*. IEEE, 2017, pp. 297–304.

[15] B. James, H. Quinn, M. Wirthlin, and J. Goeders, "Applying compiler-automated software fault tolerance to multiple processor platforms," *IEEE Transactions on Nuclear Science*, 2019.

[16] M. Bohman, B. James, M. J. Wirthlin, H. Quinn, and J. Goeders, "Microcontroller compiler-assisted software fault tolerance," *IEEE Transactions on Nuclear Science*, vol. 66, no. 1, pp. 223–232, 2018.

[17] "The RISC-V Instruction Set Manual." https://content.riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf, [accessed May-2020].

[18] J. R. Azambuja, S. Pagliarini, M. Altieri, F. L. Kastensmidt, M. Hubner, J. Becker, G. Foucard, and R. Velazco, "A fault tolerant approach to detect transient faults in microprocessors based on a non-intrusive reconfigurable hardware," *IEEE Transactions on Nuclear Science*, vol. 59, no. 4, pp. 1117–1124, 2012.

[19] A. Shye, J. Blomstedt, T. Moseley, V. J. Reddi, and D. A. Connors, "PLR: A software approach to transient fault tolerance for multicore architectures," *IEEE Transactions on Dependable and Secure Computing*, vol. 6, no. 2, pp. 135–148, 2009.

[20] B. Döbel, H. Härtig, and M. Engel, "Operating system support for redundant multithreading," in *Proceedings of the tenth ACM international conference on Embedded software*. ACM, 2012, pp. 83–92.

[21] Y. Zhang, S. Ghosh, J. Huang, J. W. Lee, S. A. Mahlke, and D. I. August, "Runtime asynchronous fault tolerance via speculation," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. ACM, 2012, pp. 145–154.

[22] C. Wang, H.-s. Kim, Y. Wu, and V. Ying, "Compiler-managed software-based redundant multi-threading for transient fault detection," in *Proceedings of the International Symposium on Code Generation and Optimization*. IEEE Computer Society, 2007, pp. 244–258.

[23] Y. Zhang, J. W. Lee, N. P. Johnson, and D. I. August, "DAFT: decoupled acyclic fault tolerance," *International Journal of Parallel Programming*, vol. 40, no. 1, pp. 118–140, 2012.

[24] K. Mitropoulou, V. Porpodas, and T. M. Jones, "Comet: Communication-optimised multi-threaded error-detection technique," in *2016 International Conference on Compliers, Architectures, and Sythesis of Embedded Systems (CASES)*. IEEE, 2016, pp. 1–10.

[25] H. So, M. Didehban, Y. Ko, A. Shrivastava, and K. Lee, "EXPERT: Effective and flexible error protection by redundant multithreading," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2018*. IEEE, 2018, pp. 533–538.

[26] H. So, M. Didehban, A. Shrivastava, and K. Lee, "A software-level redundant multithreading for soft/hard error detection and recovery," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2019, pp. 1559–1562.

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

[28] M. Grosso, M. S. Reorda, M. Portela-Garcia, M. García-Valderas, C. López-Ongil, and L. Entrena, "An on-line fault detection technique based on embedded debug features," in *2010 IEEE 16th International On-Line Testing Symposium*. IEEE, 2010, pp. 167–172.

[29] J. R. Azambuja, Â. Lapolli, L. Rosa, and F. L. Kastensmidt, "Detecting sees in microprocessors through a non-intrusive hybrid technique," *IEEE Transactions on Nuclear Science*, vol. 58, no. 3, pp. 993–1000, 2011.

[30] J. R. Azambuja, G. Nazar, P. Rech, L. Carro, F. L. Kastensmidt, T. Fairbanks, and H. Quinn, "Evaluating neutron induced see in sram-based fpga protected by hardware-and software-based fault tolerant techniques," *IEEE Transactions on Nuclear Science*, vol. 60, no. 6, pp. 4243–4250, 2013.

[31] L. Parra, A. Lindoso, M. Portela, L. Entrena, F. Restrepo-Calle, S. Cuenca-Asensi, and A. Martínez-Álvarez, "Efficient mitigation of data and control flow errors in microprocessors," *IEEE Transactions on Nuclear Science*, vol. 61, no. 4, pp. 1590–1596, 2014.

[32] L. Parra, A. Lindoso, M. Portela-Garcia, L. Entrena, B. Du, M. S. Reorda, and L. Sterpone, "A new hybrid nonintrusive error-detection technique using dual control-flow monitoring," *IEEE Transactions on Nuclear Science*, vol. 61, no. 6, pp. 3236–3243, 2014.

[33] A. Lindoso, L. Entrena, M. García-Valderas, and L. Parra, "A hybrid fault-tolerant leon3 soft core processor implemented in low-end sram fpga," *IEEE Transactions on Nuclear Science*, vol. 64, no. 1, pp. 374–381, 2016.

[34] M. N. Lovellette, K. Wood, D. Wood, J. H. Beall, P. P. Shirvani, N. Oh, and E. J. McCluskey, "Strategies for fault-tolerant, space-based computing: Lessons learned from the argos testbed," in *Aerospace Conference Proceedings, 2002. IEEE*, vol. 5. IEEE, 2002, pp. 5–5.

[35] P. R. Kleeberger, J. Rivera, D. Mueller-Gritschneder, and U. Schlichtmann, "Serohal: generation of selectively robust hardware abstraction layers for efficient protection of mixed-criticality systems," in *Proceedings of the 24th Asia and South Pacific Design Automation Conference*, 2019, pp. 33–38.

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

[37] S. Schuster, P. Ulbrich, I. Stilkerich, C. Dietrich, and W. SchröDer-Preikschat, "Demystifying Soft-Error Mitigation by Control-Flow Checking–A New Perspective on its Effectiveness," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 16, no. 5s, p. 180, 2017.

[38] A. Rhisheekesan, R. Jeyapaul, and A. Shrivastava, "Control flow checking or not?(for soft errors)," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 18, no. 1, pp. 1–25, 2019.

[39] A. Shrivastava, A. Rhisheekesan, R. Jeyapaul, and C.-J. Wu, "Quantitative analysis of control flow checking mechanisms for soft errors," in *Proceedings of the 51st Annual Design Automation Conference*. ACM, 2014, pp. 1–6.

[40] "RISC-V Bit manipulation extension repository," https://github.com/riscv/riscv-bitmanip, [accessed 4-Nov-2022].
[41] B. Sangchoolie, K. Pattabiraman, and J. Karlsson, "One bit is (not) enough: An empirical study of the impact of single and multiple bit-flip errors," in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2017, pp. 97–108.
[42] O. Goloubeva, M. Rebaudengo, M. S. Reorda, and M. Violante, "Soft-error detection using control flow assertions," in *Defect and Fault Tolerance in VLSI Systems, 2003. Proceedings. 18th IEEE International Symposium on*. IEEE, 2003, pp. 581–588.
[43] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM conference on Computer and communications security*. ACM, 2005, pp. 340–353.
[44] "mor1kx - an OpenRISC Processor IP Core." https://github.com/openrisc/mor1kx, [accessed March-2018].
[45] S. Williams, "Icarus verilog," *On-line: http://iverilog.icarus.com*, 2006.
[46] Q. Liu, C. Jung, D. Lee, and D. Tiwari, "Compiler-directed soft error detection and recovery to avoid due and sdc via tail-dmr," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 16, no. 2, pp. 1–26, 2016.
[47] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor," in *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*. IEEE, 2003, pp. 29–40.
[48] "llvm-or1k - Low Level Virtual Machine for Or1k." https://github.com/openrisc/llvm-or1k, [accessed September-2018].
[49] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*. IEEE, 2001, pp. 3–14.
[50] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert, "Statistical fault injection: Quantified error and confidence," in *Proceedings of the Conference on Design, Automation and Test in Europe*. European Design and Automation Association, 2009, pp. 502–506.
[51] Q. Liu, C. Jung, D. Lee, and D. Tiwari, "Compiler-directed lightweight checkpointing for fine-grained guaranteed soft error recovery," in *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2016, pp. 228–239.
[52] H. Schirmeier, C. Borchert, and O. Spinczyk, "Avoiding pitfalls in fault-injection based comparison of program susceptibility to soft errors," in *Dependable Systems and Networks (DSN), 2015 45th Annual IEEE/IFIP International Conference on*. IEEE, 2015, pp. 319–330.
[53] E. Cheng, S. Mirkhani *et al.*, "CLEAR: Cross-layer exploration for architecting resilience: Combining hardware and software techniques to tolerate soft errors in processor cores," in *Design Automation Conference (DAC), 2016 53nd ACM/EDAC/IEEE*. IEEE, 2016, pp. 1–6.
[54] M. Didehban and A. Shrivastava, "A compiler technique for processor-wide protection from soft errors in multithreaded environments," *IEEE Transactions on Reliability*, vol. 67, no. 1, pp. 249–263, 2018.
[55] C. Celio, P. Dabbelt, D. A. Patterson, and K. Asanović, "The renewed case for the reduced instruction set computer: Avoiding isa bloat with macro-op fusion for risc-v," *arXiv preprint arXiv:1607.02318*, 2016.

Moslem Didehban received his MS degree in computer architecture engineering from Amirkabir University of Technology, Tehran, in 2010. He received his Ph.D.in Computer Systems from Arizona State University, Tempe, in 2018. His research interests include involving error resilience designs, computer architecture and microprocessor simulation, embedded software, and compiler optimizations.

Hwisoo So is a PhD student in Dependable Computing Lab at Yonsei University. He received Bachelor's degree in Computer science from Yonsei University, and currently, he is in integrated PhD Course at the same university. His research interests include reliability issues such as comprehensive vulnerability estimation of computer architecture and hardware/software based protection schemes against soft and hard errors based on redundancy.

Prudhvi Gali is a masters student at Arizona State University focusing more on Embedded systems. Working as a graduate research assistant at compiler and micro-architecture lab. Prior joining to ASU worked as an embedded software developer at NXP. Areas of interest: Reliability of processors, Real time systems, Compilers.

Aviral Shrivastava is a full Professor in the School of Computing Informatics and Decision Systems Engineering at the Arizona State University, where he has established and heads the Make Programming Simple Labs.

He received his Ph.D. and Masters in Information and Computer Science from the University of California, Irvine, and bachelors in Computer Science and Engineering from Indian Institute of Technology, Delhi. Prof. Shrivastava's research lies in the broad area of Software for Embedded and Cyber-Physical Systems. He is currently serving as associate editor for ACM Transactions Embedded Computing Systems (ACM TECS), IEEE Transactions on MultiScale Computing (IEEE TMSC), IEEE Transactions on Computer-Aided Design (IEEE TCAD), and Springer International Journal on Parallel Processing (Springer IJPP), and Springer Design Automation for Embedded Systems (Springer DAEM). He is currently the program chair of CODES+ISSS 2017, one of the top conferences in embedded systems.

Kyoungwoo Lee is an associate professor in the department of computer science and engineering at Yonsei University, Seoul, South Korea.He received B.S. and M.S. degrees in computer science from Yonsei University in 1995 and 1997, respectively, and Ph.D. degree in information and computer science at the University of California at Irvine in 2008. His research is in the area of embedded systems, with a specific focus on cross-layer design and optimization for error-aware and energy-efficient embedded systems.