A Compiler Technique for Processor-Wide Protection From Soft Errors in Multithreaded Environments

Moslem Didehban and Aviral Shrivastava, *Senior Member, IEEE*

***Abstract*—Aggressive transistor scaling down and near-threshold computing have rendered modern microprocessor susceptible to soft errors. Software approaches that protect computations against soft errors are desirable because they offer flexible protection and are suitable for mixed-critical systems. In particular, fine-grain instruction duplication based techniques are deemed to be most effective; however, many of the existing instruction duplication techniques either suffer from many vulnerable intervals or are not suitable for multithreaded environments. In this paper, we present multithreded near zero silent data corruption (MZDC), a software scheme which provides high-level processor-wide error coverage in multithreaded environments. MZDC duplicates all programs' instructions and uses diagnosis block after replicated memory operations to overcome the inconsistency issue in a multithread environment. Statistical fault injection experiments on a dual-core ARM cortex-A53 μ architecturally simulated microprocessor show that on average, MZDC can achieve more than 37× better fault coverage than the state-of-the-art.**

***Index Terms*—Compiler transformation, multithreading, reliability, soft errors, transient faults.**

I. Introduction

RAPID technology scaling, high integration density, and near-threshold computing are viewed as the main drivers of modern microprocessors power and performance improvements. The irony is that these reasons that make all our technological dreams possible are the same reasons why modern microprocessors are becoming increasingly prone to transient faults [1]–[3]. Among many sources of transient faults in systems (e.g., electrical noise, external interference, and crosstalk), subatomic particle strikes (low and high energy neutrons) on sensitive areas of a transistor are considered as the major source of transient faults or soft errors in electronic devices [4]. While high energy neutrons (100 KeV–1 GeV from cosmic background) have been considered as the main source of soft errors in the past, ITRS 2015 [5] predicts that soon even low-energy ground-level muon particles will become the main source of soft errors. This results in a multiplicative effect because of the presence of exponentially more low-energy particles than those at higher energies [6]. At the current technology node, a soft error may occur in a high-end server once every 170 h, but this is also expected to increase exponentially with technology scaling [7], [8].

Most soft error protection solutions have been developed at the hardware level. For instance, systems, such as IBM Z-series servers [9], HP NonStop-systems [10], and HERMES [11], execute instructions on redundant pipelines, and compare the redundantly-computed for errors. However, the expensive cost of such hardware techniques restricts their applicability to cost-agnostic systems, such as aircrafts and space shuttles.

Software approaches have a distinct advantage in that they can be applied to any existing processor, and their application can be more prudent, i.e., they can be applied only to more critical or vulnerable applications. More than a dozen software-based in-application instruction replication approaches have been proposed, including EDDI [12], SWIFT [13], nZDC [14], SWIFT-R [15], [16], Shoestring [8], IPAS [17], DRIFT [18], ESoftCheck [19], and [20]–[22]. The key idea in these methods is that soft errors can be detected by duplicating computational/logical instructions of programs with different sets of registers and frequently comparing the values of these redundant registers. Memory operations, compare, branch, and function call instructions are the typical soft error checking points in the existing instruction-replication based techniques. Our in-depth investigation of the existing fine-grained instruction-duplication based schemes reveals that they either suffer from many single points of failures (unprotected instructions) or are not applicable to the multithreaded environment. For instance, SWIFT [13], IPAS [17], and ESoftCheck [19] schemes can detect the manifestation of errors only if the error permutes the execution of a duplicated instructions. Nevertheless, about 20%–40% of program instructions including memory (read and write) and control flow instructions (e.g., compare, branch, and call instructions) are not replicated and therefore are susceptible to soft errors. Other schemes (i.e., EDDI [12], nZDC [14], DRIFT [18], Casted [23], and Shoestring [8]) may face frequent false alarms in multithreaded applications. Examples of that are where correctly synchronized shared accesses, such as barriers, flag-synchronization are implemented by nonlock-based methods. In such cases, programmer allowed race conditions or benign races [24] are deciphered as the manifestation of soft error, and the program's execution will be interrupted by the soft error protected scheme.

In this paper, we explore the problem of memory instruction replication in multithreaded applications. Then we propose MZDC or multithreaded nZDC, which checks the results of

Manuscript received October 6, 2016; revised March 19, 2017, June 30, 2017, and October 24, 2017; accepted January 7, 2017. This work was supported in part by the National Science Foundation under Grant CCF 1055094 and Grant CNS 1525855. Associate Editor: J. Zhang. *(Corresponding author: Moslem Didehban.)*

The authors are with Arizona State University, Tempe, AZ 85287, USA (e-mail: Moslem.didehban@asu.edu; aviral.shrivastava@asu.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TR.2018.2793098

redundant memory read instructions right after their execution and utilizes an off-critical-path diagnosis blocks to distinguish between soft errors and intervening memory stores operations from other threads. Furthermore, in the case of error, MZDC provides information about error propagation scope which can be used for recovery purposes, i.e., an error is local to the thread or error has been propagated to the shared memory. Furthermore, we propose several optimizations to reduce the performance overhead of MZDC technique without jeopardizing its coverage.

We evaluated the effectiveness of MZDC scheme using fault injection experiments. We have implemented SWIFT [13] and MZDC in the LLVM compiler [25] for ARM v7-a ISA. We compiled several applications from Splash-2 [26] and ParMiBench benchmarks suite [27], and run them on gem5 [28] full system mode while simulating an ARM cortex-53 dual-core like microprocessor. We inject thousands of faults in different microarchitectural components and find that when compared to the state-of-the-art, MZDC reduces the effective failure rate (EFR) by approximately 30×.

II. Background

The general idea behind most error detection approaches is that by executing two redundant versions of computations, the presence of an error can be concluded from any discrepancy in results. The redundant computations can take place at different design layers – hardware [11], [29]–[32] only, software only [8], [12]–[14], [18]–[20], [33], or hybrid methods [34], [35]. This section provides an overview of the existing hardware error detection schemes. Then we focus on the most well-known software-level fault detection scheme.

A. Hardware Approaches

Traditional, hardware error detection techniques explore the spatial redundancy and run two redundant executions on different hardware components – different core or different execution path inside a core, and compare the redundantly computed results. These techniques achieve a high degree of fault coverage, but also introduce a considerable amount (more than 2×) of performance, power, and area overheads. Hardware multithreading techniques, such as RMT [30], execute redundant threads in a simultaneous multithreaded processor to protect computations against soft errors. In such techniques, two exactly similar threads, named leading and trailing threads, are executed on one or different core(s) of a multithreaded processor, and error-detection is accomplished by checking some redundantly-computed result (mainly store instruction's value and address). Although redundant multithreading based techniques suffer from significantly less performance overhead than traditional fully redundant lock-stepped processors [36], they still require modifications in underlying hardware by adding some hardware components, such as load value queue and store value queue [30]. Researchers have also explored chip-multiprocessors (CMP) to address the problem of soft errors. Now, instead of running two redundant threads (such as in SRT), redundant processes can execute on different cores of a CMP, thus providing better performance overhead [31]. Overall, all these techniques demand moderate to significant hardware modification, and are not suitable for mixed-critical systems [37] because their protection is hard-wired to the hardware.

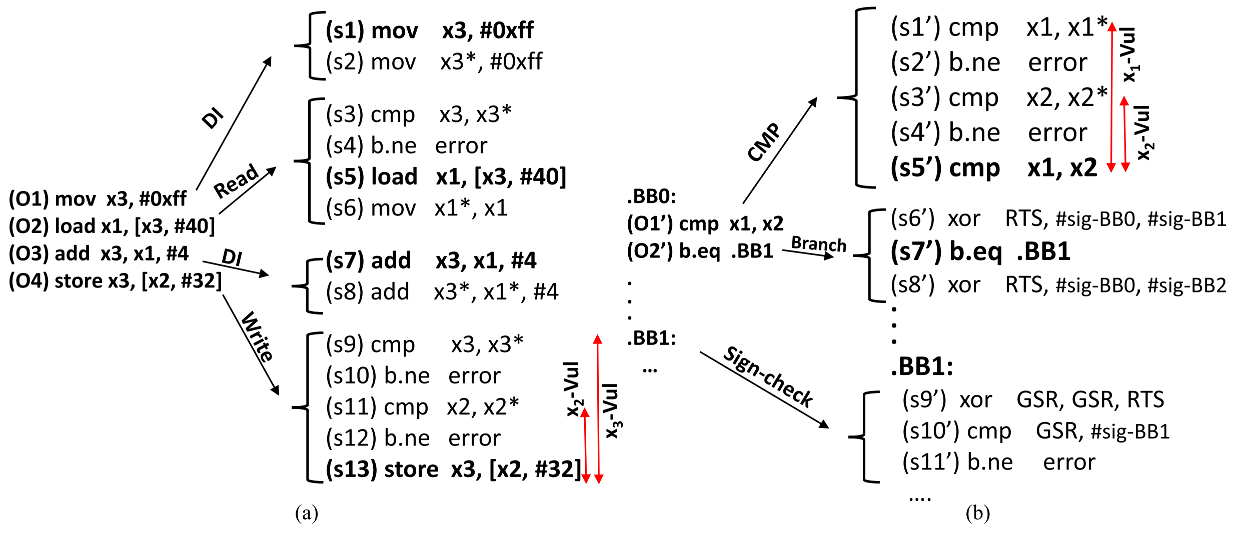
B. Software Approaches

In order to achieve a low-cost and flexible fault tolerance mechanism, several software-level techniques have been proposed. Among software-only fault tolerance techniques, low-level instruction-duplication based schemes [8], [12], [13], [15], [18]–[20] are the most popular ones. The main idea behind such methods is that by partitioning programmer available registers into two sets and replicating program instructions with different registers, the manifestation of soft errors can be detected by inserting checking instructions (CI) at some specific points of execution. For example, EDDI [12] duplicates all program instructions expect compare and branch instructions and performs checking operations before stores and conditional branches. Although EDDI seems to be very effective, an error during the execution of branch or compare instructions can corrupt the program's output. In addition, since EDDI duplicates the memory subsystem, its performance overhead can be very high, especially for memory-intensive applications.

In an attempt to improve the performance and fault coverage of EDDI, SWIFT [34] eliminates the need of memory duplication by assuming that the memory subsystem is protected (by other means, such as ECC). In SWIFT transformation, all computational/logical instructions are duplicated with different registers and the CI are inserted before three type of instructions

1) memory read operations,
2) memory write operations, and
3) control flow instructions, i.e., compare, branch, and function calls.

Fig. 1(a) shows the SWIFT data flow transformation and the corresponding original code. In the figure, shadow registers are different from the master ones by a star (`x1*` is the shadow for `x1`). In the snippet code presented in the figure, original "mov" (inst. O1) and "add" (inst. O2) instructions are duplicated by SWIFT transformation. This transformation is marked by `DI` in the figure because we classify these type of instructions as duplicatable instruction (DI). SWIFT transformation does not duplicate the memory read instructions; however, it checks for error in address register before the execution of such instructions and copies the loaded value into the corresponding shadow register. The SWIFT memory read instruction transformation is marked by `read` in the figure. For the "load" (inst. S5) instruction, the value of address register `x3` is checked against the redundant-computed value `x3*`, before the execution of "load" instruction and the loaded value `x1` is copied into the corresponding shadow register `x1*` right after the "load" instruction. We named this extra "move" instructions (inst. S6) as "copying-move" instruction. SWIFT transformation only executes one version of memory writes instructions; however, the register operands of such instructions are checked before their execution. The memory write instruction transformation is marked as `Write` in the figure, and, as it shows, the "store" in-



```
(O1) mov  x3, #0xff
(O2) load x1, [x3, #40]
(O3) add  x3, x1, #4
(O4) store x3, [x2, #32]

(s1) mov    x3, #0xff
(s2) mov    x3*, #0xff

(s3) cmp    x3, x3*
(s4) b.ne   error
(s5) load   x1, [x3, #40]
(s6) mov    x1*, x1

(s7) add    x3, x1, #4
(s8) add    x3*, x1*, #4

(s9) cmp     x3, x3*
(s10) b.ne   error
(s11) cmp    x2, x2*
(s12) b.ne   error
(s13) store  x3, [x2, #32]
```

```
.BB0:
(O1') cmp  x1, x2
(O2') b.eq  .BB1
:
.BB1:
...

(s1') cmp    x1, x1*
(s2') b.ne   error
(s3') cmp    x2, x2*
(s4') b.ne   error
(s5') cmp    x1, x2

(s6')  xor    RTS, #sig-BB0, #sig-BB1
(s7') b.eq  .BB1
(s8')  xor    RTS, #sig-BB0, #sig-BB2
:
.BB1:
 (s9')  xor    GSR, GSR, RTS
 (s10') cmp    GSR, #sig-BB1
 (s11') b.ne    error
 ....
```

Fig. 1. SWIFT transformation: part (a) data flow protection and (b) control flow protection.

struction operands, `x3`, `x2` are checked against their shadows `x3*`, `x2*` before the execution of `store` instruction.

To protect the control flow of a program, SWIFT checks the "compare" instructions register operands before the execution of those instructions and uses statically-assigned signatures to detect errors affecting the execution of "branch" instructions. Fig. 1(b) shows the SWIFT control flow transformation. Before the execution of `cmp` (inst. s5'), the register operands `x1` and `x2` are checked against their shadow registers `x1*` and `x2*`. Before each "branch" instruction (including function calls) or in the end of each basic-block, the run-time-signature (RTS) register is computed from the current basic-block signature and the branch destination basic-block signature (inst. S6' and S8'). The control flow error detection takes place in the beginning of each basic-block, by extracting the basic-block signature from the RTS and general signature registers (GSR) registers, and checking the dynamically calculated signature against the statically-assigned ones. This checking process is labeled as `Sign-check` in the figure.

For more than a decade, SWIFT has been considered as the most effective in-thread instruction-duplication technique in terms of fault coverage. Acknowledging the near perfect fault detection ability of SWIFT, several works [8], [18], [18]–[20] have tried to improve the performance overhead of SWIFT transformation. For instance, in Shoestring [8], the authors express that "SWIFT has the advantage of being purely software-based, thus requiring no specialized hardware, and can achieve nearly 100% coverage". Targeting noncritical applications, Shoestring compromises between fault coverage and performance overhead of instruction duplication by taking advantage of low-level hardware symptom detectors and applying instruction duplication to those instructions which error on them most likely cause no symptom. Similarly, research IPAS [17], also apply instruction-duplication judiciously of most critical instructions of programs and tradingoff coverage for performance overhead. On the other hand, DRIFT [18] and ESoftCheck [19] try to improve SWIFT

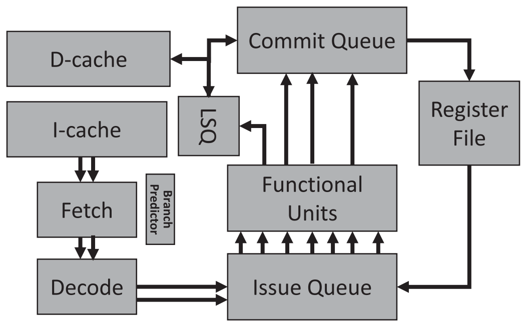


Fig. 2. Baseline processor. Note that Caches and branch predictor are excluded from our error coverage analysis.

performance overhead without jeopardizing its fault coverage. A recent study [38] explores the use of vector instruction for instruction-replication based fault tolerant techniques and illustrates considerable performance improvements.

III. Limitation of the Existing Instruction-Duplication Techniques

Ideally, a perfect software fault detection scheme should be able to protect the execution of all program instructions against soft errors on various hardware components. However, the main error coverage limitation of existing instruction-duplication techniques is that they can only detect the impact of errors on the execution of some instructions (i.e., the ones that they replicate) and leave the rest unprotected. To provide a detailed analysis of the protection offered by instruction-duplication techniques, in particular, SWIFT [13], we investigate the impact of single bit flip transient faults on different hardware components while executing a SWIFT-protected program. We consider an in-order baseline processor (shown in Fig. 2) and examine the impact of errors on the different type of instructions in a SWIFT-protected code. We limit our analysis to the microprocessor core

TABLE I
SOFTWARE–HARDWARE VIEW OF SWIFT PROTECTION

Inst. Type	Fetch	Decode	IQ	FUs	Commit	LSQ	RF
DI	P[1]	P	P	P	P	na	P
MemRead	NP[2]	NP	NP	NP	NP	NP	NP
CopMov	P	P	P	P	P	na[3]	NP
MemWrite	NP	NP	NP	NP	NP	NP	NP
OrgCMP	NP	NP	NP	NP	NP	na	NP
OrgBr	NP	NP	P	P	P	na	NP
CI	NP	NP	P	P	P	na	P

[1]Protected.
[2]NotProtected.
[3]Not applicable.

components excluding branch predictor and memory subsystem. We do not consider soft errors in branch predictor structure because we assume that the soft errors on branch predictor will affect the performance of the processor, not its functionality [39]. We assume that memory subsystem, including TLBs and caches error detection and correct code (ECC) are protected by ECC and the data is protected while it is in memory. However, faults on cache controllers while decoding an address can still result in a wrong access (read/write) to the memory, and an ECC-protected cache is not immune to such errors [40]. In fact, a recent in-field research [41], reveals that non-DRAM memory failures from the memory controller and memory channel contribute the majority of memory errors.

First, we classify the instructions in a SWIFT-protected code into seven categories

1) Duplicable instructions (DIs): these are program computational/logical instructions that are duplicated by SWIFT transformation,
2) Memory read instructions (MemRead): these are load instructions that SWIFT checks their address register before their execution,
3) Copying-move (CopMov) instructions: these are inserted after MemRead instructions to provide consistent input replication for shadow and master instructions,
4) Memory write instructions (MemWrite): wherein value and address register operands are checked before their execution,
5) Original compare (OrgCMP) instructions: wherein register operands are checked to prevent control flow errors,
6) Original branch (OrgBR) instructions, and
7) Checking instructions (CIs), which includes instruction responsible for updating the RTS and GSR registers, checking for discrepancy in signatures or redundantly computed registers, and terminating the program execution in case of error.

Table I summarizes the protection offered by SWIFT in the presence of single bit-flip in different hardware components. The rows in table represent various type of instructions in a SWIFT-protected program and the columns show various hardware components. If SWIFT transformation is able to detect the effect of error in component `Comp.`, while it is utilized with instruction `Inst.`, the letter P (Protected) is placed in the location `(Inst.,comp.)` of the table; otherwise, it is filled by NP (NotProtected), which means SWIFT transformation cannot protect the execution of instruction `Inst.` against faults in the hardware component `Comp`.

A. *Duplicable Instructions*

The first row in the table says that SWIFT transformation can detect errors affecting the execution of a duplicated instruction, regardless of the location of the fault. If the error happens on any hardware component while processing a DI, the impact of error will get masked or detected by SWIFT CI. It is worth mentioning that if an error hits some specific bits in the microarchitectural resources, e.g., the valid-bit of an entry holding two redundant instants of a duplicated instruction, the error alters both instructions in the same way and remains undetected. However, we do not consider these rare scenarios because by interleave scheduling [12] of master and shadow instructions, this possibility can be eliminated.

B. *Memory Read Instructions (MemRead)*

As the second row of Table I shows SWIFT transformation leaves memory read instructions unprotected during their execution. This occurs because there is no redundant version or execution check for load instructions. Therefore, all errors that modify the address or the size of loaded value can result to a failure. Examples of such errors are: errors that hit fetch, decode, or issue stage registers while they are occupied by load instruction. Errors on the functional unit that is responsible for load effective address calculation. Errors on memory read request address or size while the load request is processing in the load-store queue. Even if an error hits the source register of a memory read instruction [e.g., register `x3` of `load` in Fig. 1(a)] before getting *directly*[1] accessed by the memory read instruction, it will cause a wrong-memory-location access error. Note that in this case, since the state of load address register is different from its shadow, the error will remain undetected if the next access to the load source register is a write.

Note that the execution of memory read instructions are unprotected in many similar instruction-based replication techniques—including [15], [19], [20], [22], [23], [33], [38], and [42]–[46].

C. *Copying Mov Instructions (CopMov)*

Soft error on all microprocessor hardware components, expect register file, while processing CopMov instructions is covered by SWIFT transformation. If an error hits a CopMov instruction source registers [e.g., `x1` of CopMov instruction (inst. s6) in the Fig. 1(a)], the error will propagate from master register (`x1`), to its corresponding shadow register (`x1*`) and remain unnoticed.

All instruction-replication based techniques that do not duplicate the memory read instructions, including [15], [19], [20], [22], [23], [33], [38], and [42]–[46], also suffer from this vulnerable interval.

[1]An instruction accesses a register directly, if that is the last access to the register so far.

D. Memory Write Instructions (MemWrite)

Similar to MemRead instructions, memory write instructions are also vulnerable all through their execution. Errors affecting opcode, data, and address register pointers, immediate, shift, size, and rotate field while a memory write instruction is processing by pipeline registers are the examples of such undetected errors. Likewise, errors altering memory effective address while the instruction is utilizing functional unit or load-store queues also remain undetected. Furthermore, the operands of store instructions are also susceptible to soft errors that occur before the register gets accessed directly by such instructions. However, there is a chance that such errors in the register file are detected by the upcoming CI if the next access to faulty registers is a read. Memory write instructions are also single-point-of-failure in many fine-grain instruction-replication techniques—including [8], [15], [18], [19], [21]–[23], [33], [38], and [42]–[48].

E. Original Compare Instructions (OrgCMP)

As the Fig. 1(b) shows SWIFT transformation adds checks to the operands of `compare` (inst. s5') to avoid wrong-direction control flow errors. However, since the `compare` instruction itself is not duplicated, errors during the execution of `compare` instruction itself, can change the control-flow of the program and remain undetected. Note that in this case, the signature-based part of SWIFT control flow checking mechanism is also unable to detect the error because the RTS register is set for both (taken or not-taken) directions (inst. s6' and s8'). There are many similar techniques, including [8], [12], [15], [18], [19], [21]–[23], [38], [42]–[44], and [46]–[48] that suffer from the same control-flow vulnerability.

F. Original Branch Instructions (OrgBr)

The original branch instructions are vulnerable to soft errors when they are in the fetch and decode units. If an error changes the branch opcode or condition field of a branch instruction, it can change the direction of the branch from taken to not-taken or vice versa, and the error remains unnoticed. Most of the errors that change the target address of a branch will get detected by the signature-part of the SWIFT control flow checking mechanism. However, if an error causes a jump back to the body of the source basic-block (after the signature-CI) it still remains undetected because the signature-checking part is already passed. Furthermore, in Table I, it is noted that the branch instructions are not protected in the register file. This is because the source register for (direct or indirect) branch instructions is the program status flag register, which holds zero, overflow, negative, and carry bits, and if an error happens on the flag register, the direction of the branch will be changed and the error is undetectable. For function calls, depending on the implementation, if just one copy of registers is sent to the callee function, the registers are unprotected between checking (in caller function) and duplicating time (in callee function). Many of similar works, including [8], [15], [18], [19], [21]–[23], [38], [42]–[44], and [46]–[48], also suffer suffer the same problem.

G. Checking Instructions

SWIFT transformation inserts a plenty of CI into the code to ensure the correct execution of the program. These CI are either redundant-register mismatch-CI (i.g. instructions s3, s4, s9, s10, s11, s12, s1', s2', s3', and s4' in Fig. 1) or signature setting/CI (i.e., instructions s'6, s8', s9', s10', and s11'). Generally, errors affecting these instructions just cause false alarms and not lead to a failure. However, in some special cases (e.g., the opcode of a CI instruction changes to a store instruction), it is possible that the program experiences failure because of soft error on a CI instruction.

Overall, the memory and control-flow instructions are the main single-point-of-failures in many in-thread instruction-replication based techniques. In [49] and [50], the amount of MemRead, MemWrite, OrgCMP and OrgBr instructions are reported as 50%, 55%, and 40% on average for X86, ARM, and MIPS processors, respectively.

IV. Multithreaded Programs Challenges

In order to improve the performance overhead and error coverage of SWIFT, several schemes, including shoestring [8], nZDC [14] and DRIFT [18] simply duplicate memory read instructions and do not insert checking operations before/after such instructions. The intuition behind such optimization is that if there is any soft error in the load instruction address register, it will cause a discrepancy between loaded value and eventually will be detected by further checks. This solution works well on single threaded applications, however, in a multithreaded environment, such schemes encounter frequent false alarms [51], [52]—interrupting the program execution because of soft error while in fact there is no error. Note that these false alarms are mainly an issue for soft error detection only schemes that replicate memory operations.

The main reason behind false alarm in instruction-duplication (with redundant memory read operation) schemes is that they decipher race conditions as soft errors. Although generally, a multithreaded program is better to be data race free, however, in many cases the developers may allow some harmless (benign) data race conditions. Benign data race conditions are defined as races that do not affect the correctness of the program [24]. Most of these benign race conditions are allowed intentionally by the developers to achieve better performance and they are accepted in real applications, such as Windows Vista and Internet Explorer [53]–[56]. The research [57] investigates 14 applications in the Splash-2 benchmark suite and conclude that seven of them contain benign data races.

For example, consider the snippet code shown in Fig. 3 which represents a user-level synchronization barrier extracted from Cholesky program from SPLASH2 benchmark suite. Function "Send" [Fig. 3(a)] is responsible to update shared variables `taskQ` and `probeQ`. A user-level synchronization scheme is implemented in function "GetBlock" [Fig. 3(b)] which uses busy-waiting to check for update in the `taskQ` and `probeQ` shared variables. Note that the updates (write accesses) on these shared variables are protected by explicit LOCK and UNLOCK

```
LOCK (tasks[i].taskLock ) ;

      // Update shared variables
     // tasks[i].probeQ and tasks[i].taskQ

UNLOCK (tasks[i].taskLock ) ;
```

(a) **Protected Update on shared variables (Cholesky send function)**

```
while (! tasks[j].taskQ && !tasks[j].probeQ);
```

(b) **User-defined Synchronization (Cholesky GetBlock function)**

```
Loop:
      load   temp1 ,[x1] // x1 is memory address of tasks [ j ]. taskQ
      load   temp2 ,[x2] // x2 is memory address of tasks [ j ]. probeQ
      cmp    temp1, #0
      beq    .Loop
      cmp    temp2, #0
      beq    .Loop
```

(c) **Assembly-level implementation of part (b)**

```
Loop:
Redundant  ⎧load   temp1 ,[x1] // x1 is memory address of tasks [ j ]. taskQ
Loads      ⎩load   temp1* ,[x1*] // x1* is shadow register for x1

Redundant  ⎧load   temp2 ,[x2] // x2 is memory address of tasks [ j ]. probeQ
Loads      ⎩load   temp2* ,[x2*] // x2* is shadow register for x2

Checking   ⎧cmp    temp1, temp1*
instruction⎩bne    .Error
            cmp    temp1, #0
            beq    .Loop

Checking   ⎧cmp    temp2, temp2*
instruction⎩bne    .Error
            cmp    temp2, #0
            beq    .Loop
```

(d) **Protected version of (c)**

Fig. 3. Memory write intervening problem. The snippet code in part (a) and (b) are extracted from "send" and "GetBlock" functions implemented in mf.C file from Cholesky application (SPLASH2 kernel program). Part (c) shows a simple assembly-level implementation of part (b). Part (d) shows a naive instruction duplication scheme which suffers from false alarm.

mechanism, however, the read accesses (in dequeue function) are not protected. Fig. 3(c) shows a corresponding low-level implementation for the busy-waiting loop shown in Fig. 3(a) which is materialized by two memory read and two compare operations.

Now let us assume that we want to protect the "GetBlock" function by applying an instruction-duplication scheme which replicates memory read instructions. Examples of such soft error protection schemes are EDDI [12], shoestring [8], nZDC [14], or DRIFT [18]. Adopting such soft error protection scheme will result to assembly code shown in Fig. 3(d). Such transformation can result in false alarms in a concurrent environment where a thread executing "Send" function updates the shared variable(s) between redundant load operations. To quantify the impact of intervening store problem we have implemented a user-level synchronization mechanism similar to the one illustrated in Fig. 3(b) and initiate 10 workers which update the shared variables in a protected region of the code [similar to the Fig. 3(a)]. We applied load duplication and run the program for 1000 times on an Intel Core I7 desktop PC. We observed that in 30% of runs the error flag was raised and the program execution was terminated. A naive solution to this problem is to protect all accesses to the shared variables by explicit LOCK and UNLOCK mechanisms. However, the performance overhead of such implicit protection strategy is significant.

V. MZDC Method

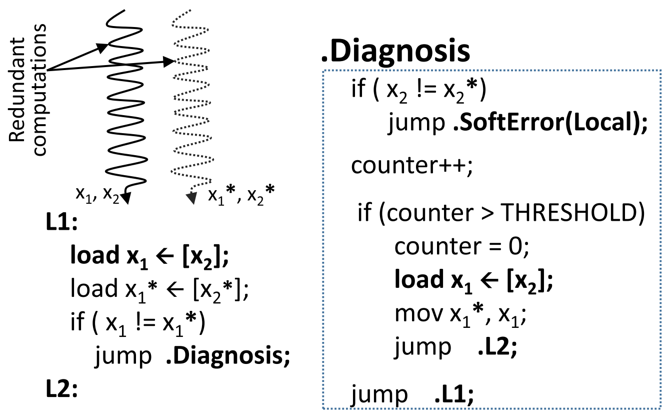
In this section, we introduce a multithreaded full instruction-replication based technique, named MZDC technique that not only protects the execution of all programs instructions, but also does not suffer from intervening store problem in multithreaded applications.

A. *MZDC: Main Idea and Assumptions*

MZDC is a fine-grain instruction-duplication based technique whose goal is to never permit a program to generate a wrong output. In other words, the MZDC transformation tries to detect all errors that may lead to silent data corruption or a user-visible failure. For this purpose, MZDC divides the program available registers into two sets, and duplicates of all programs instructions (including computational, memory read, compare, direct branches), excluding memory write and conditional branches. For these nonduplicated instructions, MZDC adopts specific strategies to verify the correctness of their execution. In line with related works [8], [13], [14], [18], [20], and [23], MZDC assumes an ECC-protected memory subsystem and tries to protect the execution of the program's instruction on microprocessor core components. Against multithreaded unsafe transformations, such as [8], [12], [14], [18], [20], and [23], MZDC takes into account race conditions and inserts extra CI to determine the presence of soft errors from (benign/harmful) race conditions. Note that MZDC is not developed to determine concurrency or race bugs in a program. The main assumption is that the MZDC transformation should just provide soft error protection of a given program and do not change its behavior.

B. *Computational and Memory Read Instruction Transformation*

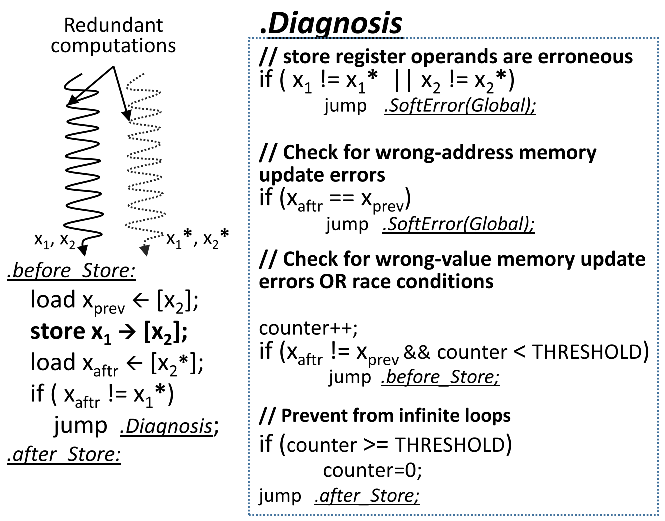
MZDC transformation duplicates all program computational and logical instructions, as well as memory read instructions. Load instruction duplication enables MZDC to protect against soft errors that happen during the execution of load instructions in microprocessor data path. Furthermore, load duplication also helps to protect a program from transient faults that may happen on the cache/memory controller during the execution of memory read operation. For instance, if an error alters the memory controller address decoder circuitry while processing a memory read request, it can cause loading from the wrong address. Even though the memory is assumed to be ECC protected, but ECC cannot help to detect wrong address memory operations. Note



```
L1:
    load x1 ← [x2];
    load x1* ← [x2*];
    if ( x1 != x1*)
        jump .Diagnosis;
L2:
```

```
.Diagnosis
  if ( x2 != x2*)
      jump .SoftError(Local);
  counter++;
  if (counter > THRESHOLD)
      counter = 0;
      load x1 ← [x2];
      mov x1*, x1;
      jump .L2;
  jump .L1;
```

Fig. 4. MZDC load instruction transformation.



```
.before_Store:
    load xprev ← [x2];
    store x1 → [x2];
    load xaftr ← [x2*];
    if ( xaftr != x1*)
        jump .Diagnosis;
.after_Store:
```

```
.Diagnosis
// store register operands are erroneous
if ( x1 != x1*  || x2 != x2*)
        jump .SoftError(Global);

// Check for wrong-address memory
update errors
if (xaftr == xprev)
        jump .SoftError(Global);

// Check for wrong-value memory update
errors OR race conditions

counter++;
if (xaftr != xprev && counter < THRESHOLD)
        jump .before_Store;

// Prevent from infinite loops
if (counter >= THRESHOLD)
        counter=0;
jump .after_Store;
```

Fig. 5. MZDC memory write instruction transformation.

that since both replicated load instructions read from an exactly same memory address, load duplication does not increase program memory footprint. However, since in multithreaded programs there is no guarantee that duplicated loads receive the same data from the memory, the MZDC transformation inserts special CI right after replicated load instructions.

Fig. 4 illustrates MZDC memory read instruction transformation. As it shows MZDC transformation redirects the program control to a diagnosis block if the results of redundant loads are different. In the diagnosis block, it first checks for errors in loads address registers (x2 and x2* in the figure). In case of a mismatch, the soft error detection flag will be raised with the hint of that error is thread local; the soft error has modified the execution of this thread and has not propagated to the memory. This information regarding the scope of error propagation can be useful for error recovery purposes, i.e., the application can decide whether the recovery is necessary, and if so, whether all threads should be rolled back or not.

If there is no error in replicated load address registers, there can be two possibilities for the inconsistency: 1) soft error happens during the execution of one of the load instructions and for instance, alters the effective address, or 2) intervening store from the other thread has modified the state of the memory. Either way, by jumping back to right before redundant-load instructions, the problem should be solved. If the soft error was the reason for the discrepancy (mismatch between x1 and x1* in the figure), simply re-execution provides recovery. If the discrepancy comes from an intervening store instruction (such as a race condition), the problem will be solved by repeating the execution of redundant loads. However, to prevent a program from going to an infinite loop (due to frequent memory update by other threads between the redundant loads instructions) MZDC transformation uses a threshold-based mechanism. Basically, it counts the number of iterations that a particular diagnostic routine has been involved (counter++ in the figure). If the number of iterations exceeds a predefined value, the MZDC first resets the counter. Then performs just one load and copies the loaded value to the corresponding shadow register, and transfers the program execution to right after the CI in the original control flow of the program.

Note that since the execution of diagnosis block is rare, the performance overhead of MZDC transformation is acceptable. Moreover, as we describe in Section VI-A, there is no need to insert checking instructions after all memory read instructions.

C. Memory Write Instruction Transformation

To verify the correct execution of memory write instructions, MZDC transformation adopts checking-load strategy that was introduced by nZDC scheme [14]. The main idea is to load back the stored value from the memory and check it against its redundant computed version. The checking-load strategy works in single-threaded applications, however, it fails in multithreaded applications where race conditions on memory update accesses are allowed. That is because if an intervening memory write instruction from other threads modifies the memory state, the loaded value by checking-load instruction will be different from the stored value and causes false alarms. An important note is that against the benign race conditions on read access to a shared variable, race conditions on write access are rare. However, as mentioned before MZDC goal is to protect the program's execution against soft errors even in the cases that a program (intentionally or because of a bug) may miss updates (writes) on some shared variables because of the race conditions on such variables.

Fig. 5 shows the MZDC transformation for a store instruction. In the figure, the original store instruction is shown as bold and the rest of instructions are inserted for soft error protection by MZDC transformation. For each memory write instruction, MZDC transformation inserts two load instructions, one before the memory write instruction, named diagnosis-load, and one after, called checking-load operation. The target address of these loads is as same as the memory write instruction. The diagnosis load (destination register xprev in the figure) is inserted to assist the diagnosis process and we explain its purpose in detail later. The address register operand of the checking-load instruction is the shadow register of the original store instructions and its destination (register xaftr in the figure) can be any arbitrary free register. MZDC transformation redirects the execution of the program to a diagnosis block if the loaded value by the checking-load instructions is not equal to the shadow register of store value register operand.

Store diagnosis routine is responsible to determine the cause of mismatch, which can be soft error or race condition. Store diagnosis routine first checks for the discrepancy between store register operands and their shadows. If there is any mismatch, the error flag will be raised with a hint of global error; soft error is propagated to the shared memory. Second, the diagnosis routine checks for errors which may happen during the execution of store instruction and cause a wrong memory update. For instance, soft errors on functional units while computing memory effective address of store operation can cause wrong memory update error. The diagnosis routine detects such wrong memory update errors by comparing the result of the diagnosis load instruction (the load instruction inserted before the store) and the checking load instruction (the load instruction inserted after the store). If the loaded values by diagnosis and checking load instructions are different, it means that the store instructions have not updated the memory target address that it supposed to update. Therefore, since we have already rolled out the chance of soft error on the store address register in the first step, we presume that soft error happens during the execution of store instruction and alters its effective address. In this case similar to the first case, MZDC transformation raises the soft error flag.

If we pass the first two checks in the diagnosis routine, it means that store instructions registers are fault free and no wrong memory location has been updated by the store instruction. Therefore, we conclude that the reason of discrepancy which was captured by the CI inserted after the store is one of the following: 1) soft errors affecting the store value during the execution of store instructions, or 2) an intervening store has modified the state of memory. In either case, diagnosis routine redirects the control of the program to before original store instructions and the program re-executes the store. Note that re-execution of the store instruction may cause a missing update on the shared variable from the other threads. However, as explained before, this is a risk which has been accepted by the program by allowing race conditions on the shared variables update operations.

Finally, to prevent a program from going to the infinite loop in the case that always intervening store(s) updates the memory state between store and the checking load instruction, we use a saturation counter policy. If the counter reaches its limit, MZDC just performs store and redirects the execution of the program to after store instruction.

D. Control Flow Instruction Transformation

To protect the execution of programs compare and branch instructions, MZDC adopts nZDC control flow transformation because such technique detects both unexpected jumps as well as wrong direction branches. Single-threaded control flow transformations are generally safe to be applied on multithreaded applications because control flow instructions do not directly deal with the (shared) memory accesses. However, since MZDC transformation increases the number of basic blocks of a program which most of them are load/store diagnosis blocks, it demands some modifications in nZDC single-threaded control-flow mechanism. In this section, we first explain the nZDC control flow checking mechanism and then address the issues that are imposed by MZDC transformation.

The nZDC control flow mechanism (shown in Fig. 6) demands two general purpose registers, called compare destination register (CDR) and compare check register (CCR). The nZDC control flow checking mechanism works based on three main insides

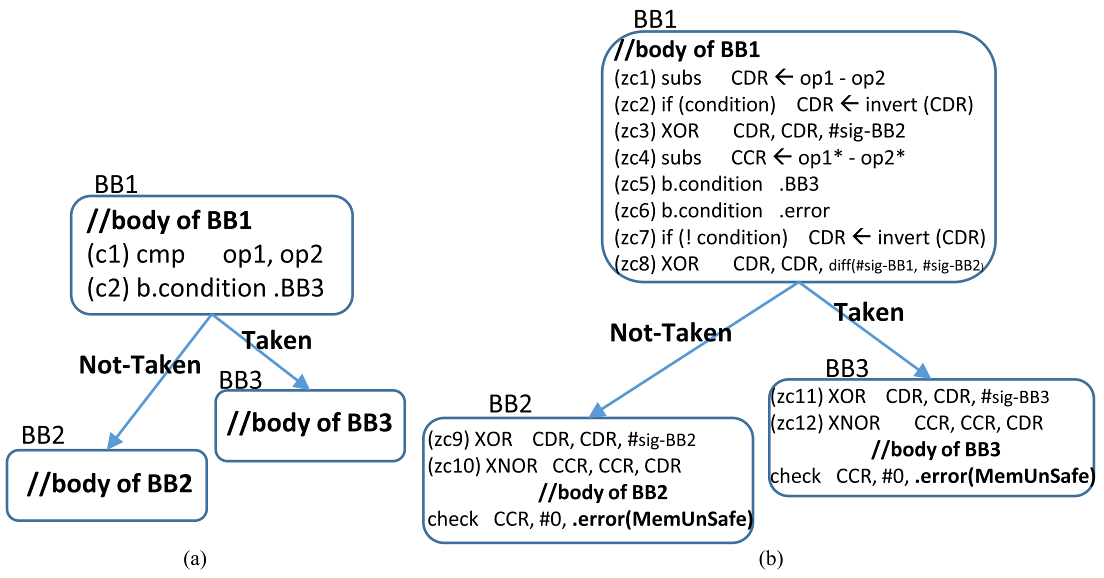
1) compare and branch instruction replication,
2) protect NVZC flag register by conditionally inverting the value of CDR register based on the direction of the following conditional branch, and
3) use static signatures for source-encoding/destination-decoding to make sure that the control flow of the program is traversed correctly.

The nZDC control flow mechanism consists of five main steps

1) Duplicating CMP Instruction: Generally, in ARM and X86 ISAs, a compare (CMP) instruction is implemented by a subtraction (SUB) instruction which disregards the results of the subtraction operation and updates the program status flags (NVZC). Leveraging this fact, nZDC control flow transformation converts all program compare instructions to their equivalent subtraction operations and duplicates them. However, rather than disregarding the results, nZDC control flow transformation saves the results of the subtraction operation into CDR and CCR registers. In Fig. 6, instructions (zc1) and (zc4) are for duplicated versions of the original CMP instruction (c1).

2) Conditionally Inverting the CDR: Since the NVZC register is not duplicable, nZDC uses time redundancy to protect that register against soft errors. For instance, as Fig. 6 shows, at time *t*, the first CMP instruction (zc1) sets the NVZC flag, which is going to be read by the following conditional invert instruction (zc2) at the next cycle (assuming one cycle per instruction). The second CMP instruction (zc4) will be set to the NVZC flag at time *t+3*, and the flag register will be read by conditional branch instruction (zc5) at time *t+4*. If the first CMP instruction (zc1) sets the NVZC flag in such way that the condition of the following conditional branch (zc5) is true, the CDR register gets inverted right after the first SUBS instruction (zc1). On the other hand, if the condition is not true and branch is supposed to be not taken, the CDR gets inverted after the branch (zc7). In a fault-free run of the program for each conditional branch, the CDR inverts just one time. Although these conditional instructions is ISA dependent, it can be replaced with a micro if the ISA does not support such instructions.

3) Duplicating Branches: nZDC duplicates all programs conditional branch instructions. However, the branch target addresses for the copy branch (zc6) is error handler basic block. The purpose of branch duplication is to protect the soft errors on the branch opcode. The main idea is if the condition is true, the main branch (instruction I5) will change the control flow of the program and the redundant one does not execute. If the condition is not true neither of the branches changes the control flow. But if errors happen on the original branch opcode, e.g., branch



```
//body of BB1
(c1) cmp      op1, op2
(c2) b.condition .BB3
```

```
//body of BB2
```

```
//body of BB3
```

```
//body of BB1
(zc1) subs     CDR ← op1 - op2
(zc2) if (condition)    CDR ← invert (CDR)
(zc3) XOR      CDR, CDR, #sig-BB2
(zc4) subs     CCR ← op1* - op2*
(zc5) b.condition   .BB3
(zc6) b.condition   .error
(zc7) if (! condition)    CDR ← invert (CDR)
(zc8) XOR      CDR, CDR, diff(#sig-BB1, #sig-BB2)
```

```
(zc9) XOR    CDR, CDR, #sig-BB2
(zc10) XNOR   CCR, CCR, CDR
              //body of BB2
check   CCR, #0, .error(MemUnSafe)
```

```
(zc11) XOR    CDR, CDR, #sig-BB3
(zc12) XNOR      CCR, CCR, CDR
              //body of BB3
check   CCR, #0, .error(MemUnSafe)
```

Fig. 6. MZDC control flow checking mechanism. All errors detected by control-flow checking mechanism are considered as global errors (error is propagated to the memory).

changes from branch less than (blt) to branch greater than (bgt) and if it changes from taken to not-taken, the redundant branch detects the error. On the other hand, if soft error alters the direction of a conditional branch from not-taken to taken, nZDC control-flow CI detect that in the wrong target basic block.

4) Adding Destination Signature: Based on the possible destination, the CDR register gets XORed with a unique and basic block signature assigned to all program basic block statically. Instruction (zc3) shows the destination related signature coding if branch is taken, otherwise, instruction (zc8) performs the signature coding for the next basic block.

5) Inserting Control-Flow CI: In order to check if the direction of a branch has been taken correctly, nZDC inserts two instructions at the beginning of the all program's basic blocks. The first instruction XORs the CDR register with the current basic block signature (zc10 in BB2 and zc9 in BB3) and saves the result back to the CDR register. The next instruction XNORs the CDR and CCR registers (zc12 in BB2 and zc10 in BB3) and stores the result in CCR. In a fault-free run of the program execution, before XNOR instruction, the value of CDR should be equal to the inverted value of CCR. Therefore, after the XNOR instruction, the CCR register value should be always zero because the inputs for the XNOR instruction are each other inverse (one's complement). Finally, the nZDC control flow error detecting instructions will be inserted into two points of execution: 1) before each write to the CCR (between inst zc3 and zc4), and 2) before all function calls and direct branches.

Since MZDC transformation increases the number of basic blocks of a program (proportional to the number of memory instructions), a naive combination of MZDC transformation and nZDC control-flow checking mechanism will result to huge performance degradation. To overcome this challenge, we thread original program basic blocks differently from the MZDC diagnosis blocks. Basically, we do not apply control-flow protection for the branches to/from the MZDC diagnosis blocks. However, in the beginning of each diagnosis block, we extract the value of CCR register and check it against zero. If the checks fail, MZDC transformation detects a control-flow error and raises the error flag. For error propagation hint, if the error is detected in a load-diagnosis block, it will be announced as thread local, otherwise as global. On the other hand, if the value of CCR is zero, the diagnosis routine restores the CCR and CDR values to their previous values before resuming the execution of the program. Note that since the MZDC diagnosis blocks execute just in some race conditions, the performance overhead of their execution is negligible.

VI. Performance Optimization

In this section, we introduce two types of performance optimization for MZDC transformation—the first one aims to reduce the number of diagnosis blocks and the second one reduces the overhead of MZDC store checking operation.

A. Reducing Number of Diagnosis Blocks

As explained before, the problem of false alarm in applying single-threaded nZDC raises just when there are potential data races in a multithreaded program. This implies that simply applying single-threaded load and store transformations to data race free memory access will case no false alarm. Therefore, the multithreaded-related soft error detection instructions (i.e., error checks after memory read operations, diagnosis-loads inserted before store instructions and corresponding diagnosis basic blocks) are unnecessarily for data race free memory accesses. Examples of such cases are

Memory accesses in a critical section of program: In a critical section of code that mutual exclusion is guaranteed by explicit memory synchronization objects like "lock" or "mutex," memory accesses can be protected against soft errors similar to single-threaded ZDC transformation.

Accesses to thread-local storage: Memory accesses (read and writes) to thread-local storages (i.e., variables identified by thread-local keyword in c++11) are guaranteed to be race-free. Therefore, they can be safely protected by single-thread nZDC load/store transformations.

Address-untaken local variables: Accesses to variables that reside in a thread stack storage can be considered as race-free if their addresses are never taken (i.e., the address-of operator is not used for them). For memory accesses to such variables multithreaded transformation is not required.

Read-only variables: A necessary condition for race condition is the presence of write operations. Access to read-only variables, i.e., immutable and constant objects, are always race free and can simply protect by single-threaded transformation.

In the above-mentioned cases, the overhead of MZDC transformation can be reduced by applying single-threaded ZDC transformation for memory operations. However, this approach is still conservative and many practically race-free memory operations will be protected by multithreaded ZDC transformation. To further reduce the overhead, memory profiling techniques can be used to determine more race-free memory operations.

B. Reducing the Overhead of Store Checking

For memory write operations that the compiler cannot guarantee their race-free execution, ZDC multithreaded store transformation should be applied (described in Section V-C). One way to reduce the overhead of store transformation is to fuse the diagnosis-load instruction with the main memory write instruction. Memory operation fusion can accomplished by utilizing ISA provided atomic operations, i.e., “SWAP” operation in ARM ISA or “XCHG” in X86 ISA. As mentioned by [61] memory instruction fusion can reduce instruction traffic through the pipeline for performance and energy efficiency.

C. Control Flow Checking Optimization

MZDC control-flow transformation uses static signatures [instructions (zc3), (zc8), (zc9), and (zc11) in the Fig. 6(b)] to detect control flow errors that lead to unwanted jumps in a program. An alternative for these signature-based unwanted jump detection is to schedule redundant and main instructions in an interleaved fashion that any unwanted jump causes a mismatch between redundant streams [12]. By applying such compile time instruction scheduling, the signature encoding/decoding part of MZDC transformation can be eliminated and the overhead of control flow checking can reduced significantly.

VII. Experimental Methodology

1) Compilation Environment and Benchmarks: Mixed applications from different category of ParMibench [27] and SPLASH-2 [26] test suits are used as representative workloads in our experiments. The detail of the selected workload is shown in Table II. The workloads are compiled with -O3 optimization level by LLVM-3.7 [25] compiler infrastructure. The MZDC and SWIFT transformations are implemented as late back-end passes, after register allocation and instruction scheduler. This implementation enables us to take advantage of all advanced compiler optimizations, including common subexpression elimination (CSE) and dead code elimination (DCE). In fact, we believe by disabling these optimizations and implementing instruction duplication techniques in LLVM IR-level, such as [18] and [20], the probability of faults masking increases due to redundant instructions, and ultimately leads to a wrong conclusion.

TABLE II
Workloads

Application	Description	Input
Dijsktra(mqueue) (ParMibench)	Single-source shortest path in a graph	input_small.dat
Dijsktra(All) (ParMibench)	All pairs shortest path in a graph	input_small.dat
Stringsearch (ParMibench)	Search specific words in given phrases	SearchString32.txt testpattern10.txt
Susan(corners) (ParMibench)	MRI image recognition application (corners)	input_small.pgm
Susan(edges) (ParMibench)	MRI image recognition application (edges)	input_small.pgm
FFT (Splash-2)	1-D Fast Fourier Transformation	1K points
Radix (Splash-2)	Iterative integer radix sort	1k integers radix 1024
Cholesky (Splash-2)	Matrix Factorization	tk15.O

TABLE III
Simulator Configuration

Parameter	Value
CPU Model	ARM 32-bit dual-core in-order processor
Pipeline	Two way/4-stage
# of FUs	2Int, 1Mul, 1Div, 1Float, 1Mem, 1Misc
L1 D/I-Cache	64 KB (2-way)/32 KB (2-way)
# of integer regs	16 registers (32-bit width)
Operating System	Linux Kernel version 2.6.22.9

2) Simulated Processor: The ideal evaluation strategy for fault injection experiments is to perform them on a gate-level RTL description of a modern microprocessor while it is running multithreaded programs. However, since we do not have access to such low-level description of a modern processor, we have used gem5 [28], a cycle-accurate simulator. We have configured gem5 simulator in full system mode. The programs were compiled for ARMv7-a profile and simulated on a dual-core two-way in-order ARM architecture with the configuration details shown in Table III. The simulated CPU model is close to the modern high-performance low-power embedded microprocessors, such as ARM Cortex-A53 processor. It is worth mentioning that the memory subsystem and the TLBs in protected version of ARM Cortex-A53 are protected with Parity/ECC, and our presumption about protected memory is consistent with the simulated CPU. We inject faults into register file and pipeline registers of an in-order CPU, however, since these structures also exist in the modern out-of-order processors, we expect that an out-of-order model would not affect our conclusions.

A. *Fault Injection Strategy and Output Classification*

1) Fault Model and Fault Injection Sites: The fault model used in this work is a single bit-flip model. This model has been widely used in experimental evaluation of the previously proposed solutions [8], [12], [13], [15], [18], [45].

The experimental results shown in this paper are produced with fault injection trials per each fault site (register file and pipeline registers). For each fault injection simulation, a random bit and a random cycle is selected statically. Then, we start the simulation normally, and pause it once it reaches to the target fault injection cycle. Then, we invert the logical value stored in the target bit (chosen from register file or pipeline registers) and resumes the simulation. The simulation run continues its execution till the program execution terminates permanently, or the allowable execution time (2× more than fault-free run) is over. In addition, to show the effectiveness of the MZDC control-flow checking, we specifically performed fault injection on the branch and compare instructions while they utilize processor pipeline registers.

For each version of a program, we inject 900 faults (300 faults in register file, 300 faults in pipeline registers, and 300 in the control flow instructions). We produced three versions of each benchmark: ORG, SWIFT-protected, and MZDC-protected. Overall, we performed 21 600 (900 × 3 × 8) fault injection experiments in different hardware component running different versions of the benchmarks.

The result of each trial is classified into one of the following:

1) SDC: The simulation runs that permanently terminated with a user-visible data corruptions without any detection alert.
2) Others: All other scenarios, i.e., masked faults, detected fault and segmentation faults fall into this category.

B. *Comparison Metric*

The common practice to evaluate the efficacy of software fault tolerant techniques is by comparing percentage of failures/SDCs extracted from statistical fault injection with performing the same number of fault injection experiments for original and protected versions of the programs [8], [12], [15], [18], [19], [21]–[23], [33], [38], [42]–[48]. However, since usually software fault tolerant methods prolong the execution time of the program, they can decrease the percentage of SDCs just by increasing the amount of masked or detected errors caused by the faults that influenced the program-irrelevant parts of execution [59]. Program-irrelevant parts of execution is the segment of the execution time that is not part of the original program, but is needed to protect the original program, such as the duration of time that the processor spends to execute the redundant and control flow checking related instructions in SWIFT/MZDC protected programs. To clearly express the main idea of this section, we use a simplified example of running original and FT versions of a program on a simple in-order CPU. The execution time trace is shown in Fig. 7.

As the figure shows the original program, marked as ORG, starts its execution at time 0 and finishes at time 10. During the execution of this program, we assume some intervals as

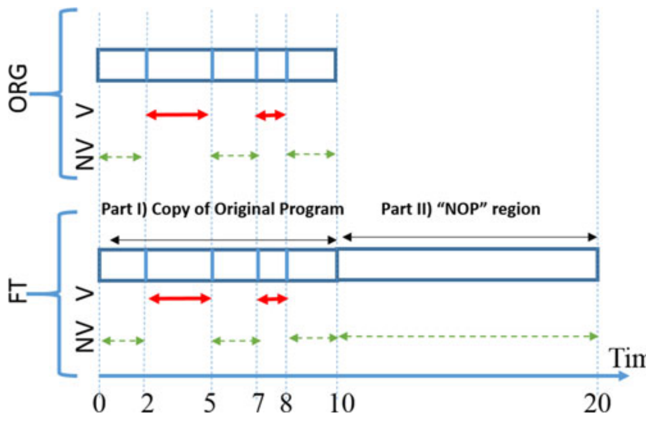


Fig. 7. Vulnerable and nonvulnerable intervals for original and FT version of hypothetical program.

Vulnerable (V) and some as nonvulnerable (NV), which means if soft error happens on V interval, it leads to program failure, and, if fault occurs on NV interval, it will get masked. This program spends 4 units of time in V intervals and 6 in NV intervals. Now, assume that ten random fault injections have been performed on this program. In an ideal random fault injection, one fault would happen on each unit of time, and since 6 units of execution time is NV and 4 is V, the amount of failures should be 4, or 40%. Now, consider a hypothetically FT version of the program. FT version of the program has two parts; the first part is exactly similar to the original ones, and the second part is just no-operations (NOP). The execution time of the FT version is as twice as the original one, which makes the execution time of the FT version 20 units of time. Now, assume we perform the same random fault injection experiments that we did on the original version of the program (ten fault injection). If we randomly select ten cycles to perform fault injection, statistically speaking, most likely five of them would occur on the second part of the program (the NOP execution part) which is NV interval, and therefore, will not result in failure. From the five remaining faults, happening on the first part of the FT program, two will happen on V intervals and three should happen on NV intervals. Therefore, the number of failures, in this case, is 2, or 20% of total injected faults. Although the ideal statistical fault injection results show significant (from 40% to 20%) reduction in failure rate, the question, however, is whether this reduction is real or just a false conclusion.

Intuitively, it is clear that the correct interpretation of SFI results should demonstrate exactly the same amount of failures for both original and FT version of the program. In this paper, we introduce effective failure rate (EFR) as a comparison metric that can be calculated by involving the execution time overhead of the FT version in two ways. 1) Adjusting the number of fault injection experiments according to the execution time overhead and just comparing the absolute number of failures, not percentage. For example, injecting 20 random faults instead of 10 random faults of the FT version of the program in the above-mentioned example should get us 4 failures, which is exactly equal to the number of failures of the original program. 2) Multiplying the number/percentage of failures into the execution time overhead. For example, for FT version of the program in the example above, we can use $20\% \times 2 = 40\%$, which is equal to the original program failure rate.

In addition, as mentioned in [59], showing the number/percentage of detected or masked fault is also misleading. For instance, in our simple example, injecting ten faults in FT version of the program would lead to 80% masked faults, which in comparison with the original program fault injection is 20% more. However, this fault masking improvement is the result of faults which are injected into the program-irrelevant parts of the execution of the program. Similar to NOPs, in the imaginary FT version of the above example, redundant instructions in the in-thread duplication FT schemes are also irrelevant to the main program. Faults that affect these original program irrelevant parts will result in either masked or detected/SegFault and should not be considered as the fault detection ability of the FT method.

In conclusion, since comparing the absolute or percentage of SDCs of original and protected versions of the program can result in an overestimation of the effectiveness of the software fault tolerance techniques, in this paper, we use the EFR metric which was calculated as follows:

$$\text{EFR} = \text{Percentage of SDCs} \times \text{execution time overhead.} \quad (1)$$

For the original version of the programs, the execution time overhead is considered as 1; therefore, the EFR is equal to the percentage of SDCs. For protected versions of the program, the execution time overhead is calculated as the protected program execution time divided by the original program execution time.

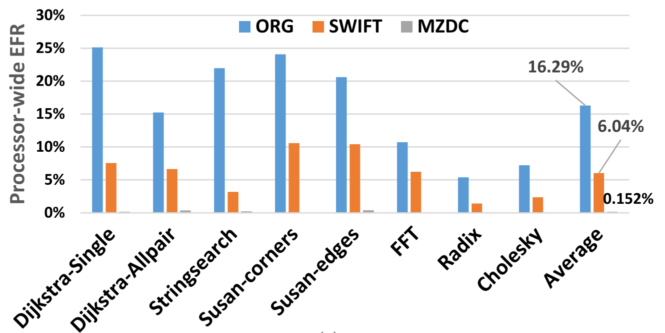
VIII. Experimental Results

In this section, we present an analysis of the effectiveness of data-flow and control-flow fault coverage of MZDC and SWIFT schemes. Then, we show the performance overhead results and analysis for these transformations.

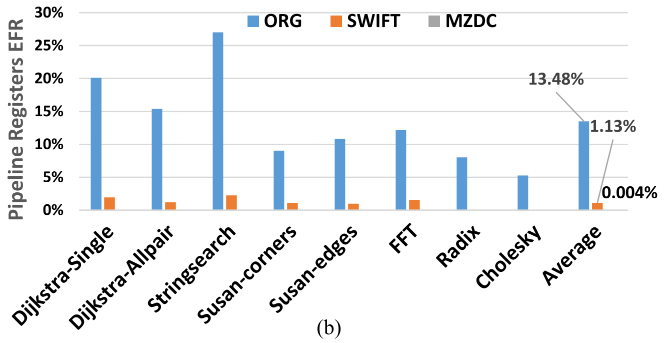
A. *Fault Injection Results*

Fig. 8 presents the results of our fault injection experiments. The y-axis plots the EFR and x-axis represents benchmarks. Fig. 8(a) illustrates the overall processor wide results. It shows that the EFR of the unprotected versions of applications is on an average around 16%. While MZDC can reduce the EFR by around $100\times$, SWIFT transformation just improves the EFR by $2.7\times$. The main reason is that unlike SWIFT transformation which only protects the execution of computational/logical instructions, MZDC protects the execution of memory and control-flow instructions as well as computational/logical instructions. In fact, SWIFT transformation increases the number of memory operations (spill code inserted by compiler due to assigning registers to redundant instructions) and leave them unprotected. Similarly, MZDC transformation also increases the number of memory operations, however, it verifies the correct execution of such operations by checking their result.

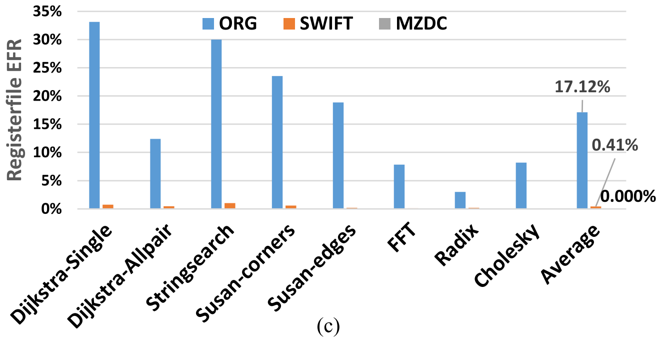
1) Pipeline Registers: Fig. 8(b) presents the percentage of EFR extracted from fault injection trails on pipeline registers, particularly registers between fetch and decode stage of the pipeline. Single bit-flip errors in these pipeline registers can alter the source/destination register pointers, immediate value, or the opcode of the fetched instruction. As the figure shows MZDC reduces the percentage of EFR from around 13.48% to fairly close to zero (0.004%), while SWIFT-protected programs suffer from 1.13% EFR. We expect that MZDC transformation detects all errors, however, we found out in a few cases the



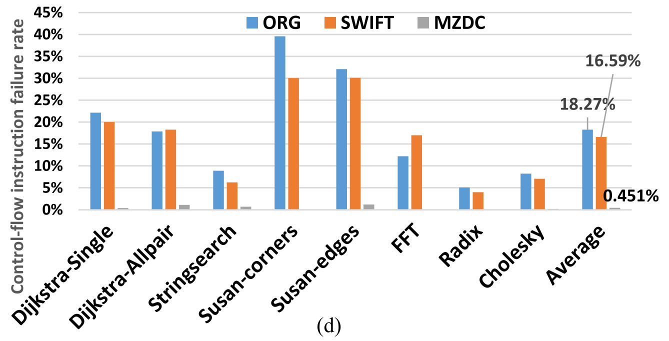
(a)



(b)



(c)



(d)

Fig. 8. Component wise EFR distribution for original, SWIFT and MZDC protected programs.

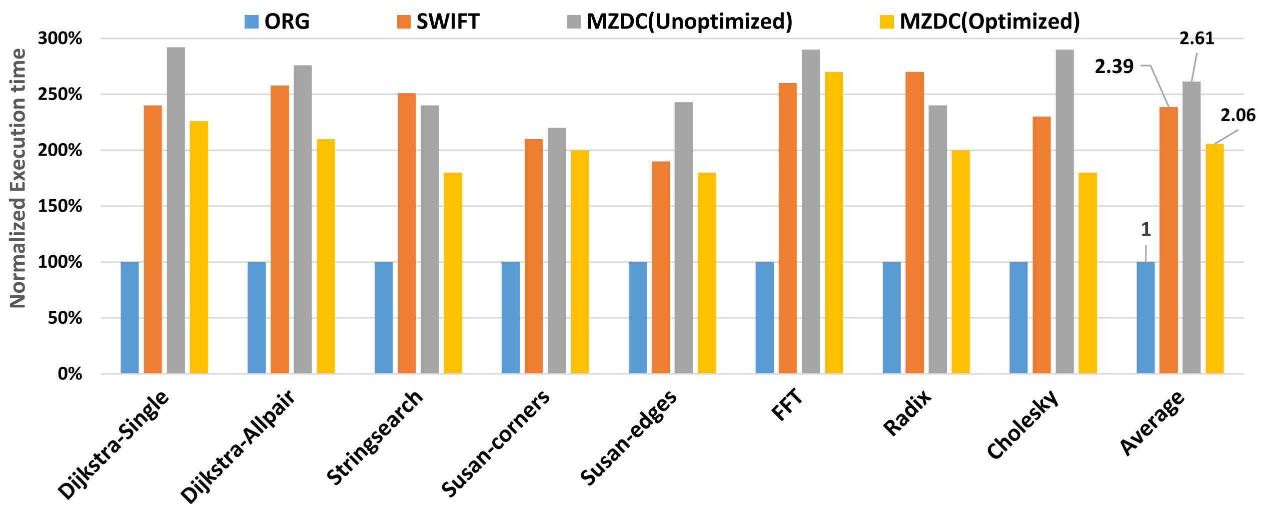


Fig. 9. Execution time overhead for SWIFT and nZDC.

injected errors remain undetected. We examined MZDC failed cases and realized in all of them the injected fault directly affects the address part (register pointers or immediate value) of a silent memory write operation. A silent memory operation is defined as a store instruction which writes a value into a memory which is already there. Now, if an error happens on the address part of a silent store, the memory write operation may modify the state of a useful memory location in an arbitrary manner. However, checking-load strategy adopted in MZDC does not detect such errors because since the store was silent, the loaded-value is equal to the redundantly computed version of the store value register.

2) Register Files: Fig. 8(c) demonstrates fault injection results on register file. On an average, register file EFR for original, SWIFT, and MZDC versions of the programs is about 17%, 0.4%, and 0.0%, respectively. MZDC transformation can completely protect register file against soft errors because it inserts CI after memory instructions rather than before. Therefore, there is no register file software vulnerable interval in MZDC protected programs. However, as described in Section III, there is always a chance that error occurs after SWIFT CI and before the execution of critical instruction and cause failure. In fact, the register file failure rate in SWIFT-protected programs is remarkably less than register file vulnerability window. The reason is that although faults on SWIFT register file vulnerability window will propagate to the memory, they do not necessarily lead to a failure. This is because in many cases even after that an error propagates to the memory, the discrepancy between redundant registers remains in the program state and will get detected by following CI.

3) Control-Flow Instructions: Fig. 8(d) demonstrates fault injection results on the programs control flow instructions. In these experiments, we randomly inject faults on the original compare and branch instructions (excluding CI) while they are in pipeline register. For a compare instruction, an injected fault can change the opcode, register pointers, or immediate value. For, conditional branch instructions the error may change the direction or the target address of the branch. Note that since the target of fault injection has been selected from program-related instructions, in this case, the EFR is equal to the percentage of SDCs. As the figure demonstrates, the percentage of SDC for original and SWIFT-protected programs is on average around 18% and 16%, respectively. This means that SWIFT control-flow protection scheme cannot detect most of the errors that directly affect the program control-flow instructions. For instance, if error alters the direction of a conditional branch, SWIFT control flow cannot detect the error, because the static signature should be set for all possible successor basic blocks. SWIFT CFC detects some of injected faults, however, since the amount of segmentation faults (detected by OS) in the original program is more than SWIFT, the failure rate remains almost same for SWIFT and original programs. In other word, errors which were detected by SWIFT control flow checking mechanism are also covered by OS. MZDC control flow mechanism, on the other hand, detects majority of control-flow injected faults (around 65%), a large portion of control-flow errors lead to segmentation faults (around 15%) and from the remaining faults almost all of the get masked. The amount of undetected control-flow errors in MZDC transformation is around 0.5%. We explored the reason and found out that in cases that the injected error modifies the target address of a taken branch in a way that the program control jumps to a standard library functions. And since we do not apply MZDC transformation to the standard library functions, the error remains undetected. We believe by applying MZDC transformation to all program code (user and library functions) all control flow error will be detected by MZDC transformation.

B. Performance Evaluation

Fig. 9 presents normalized execution time overhead of different versions of programs protected by SWIFT, MZDC (Unoptimized), and MZDC (Optimized) transformations. The performance overhead of unoptimized MZDC transformation is on an average around 22% more that SWIFT transformation—performance overhead of SWIFT is 2.4× while for MZDC it is around 2.6×. After applying all memory and control-flow optimization described in Section VI, the performance overhead of MZDC transformation reduces to 2× which is around 30% faster than SWIFT transformation. The performance overhead of SWIFT and MZDC transformations can be break down to three sources: first, overhead imposed by saving registers for shadow instructions and control flow checking algorithm, second, the performance imposed by duplicating instructions, and third, performance overhead of checking operations. The first two sources of performance overhead are almost similar for SWIFT and MZDC. That is because both schemes reserve more than half of the programmer available registers and duplicate all computational and logical instructions. The checking operations overhead is where that SWIFT and MZDC behave differently. Basically, for each memory read operation, SWIFT inserts three extra instructions (two for address checking and one for copying the loaded value to the redundant register). Unoptimized MZDC transformation also inserts three extra instructions (one replicated load and two for consistency check). Note that the replicated load instruction does not impose any new cache miss to the system. In optimized version of MZDC, in most of the times only one extra instruction is inserted for each memory read operation. For instance, in Cholesky program most of the memory read access are not from the shared memory and do not require checking operation. Therefore, MZDC optimization reduces the performance overhead of unoptimized MZDC by around 40%. In benchmarks, such as FFT and stringsearch where the memory access are mostly from the shared memory, the MZDC memory optimization is not effective. For memory write accesses, SWIFT and MZDC transformations insert four more instructions to detect the effect of soft errors. Unlike the SWIFT, MZDC inserts two extra memory read instructions for store protection, however, similar to the memory read case, these instructions do not cause a new memory access miss. The reason is these accesses are all from the same memory location that is required by the application. So, if the required memory word is not in the cache, the program face a cache miss to get access to the desirable memory location. The number of inserted instructions for control-flow checking mechanism in SWIFT and MZDC is also equal—both transformations execute six instructions to protect each program original conditional control flow instructions.

IX. Conclusion

We presented MZDC, a compiler-only error detection technique that closes all vulnerable windows in state-of-the-art in-thread instruction duplication techniques. MZDC is based on the idea that nonduplicated instructions open doors for program failures and by duplicating all instructions, complete error detection is achievable in software. However, since duplicating store and branch instructions is problematic, the proposed scheme uses checking-load instruction and a new control flow checking mechanism to verify the correct execution of such instructions. Our comprehensive evaluation based on random fault injection on various microprocessor components shows significant failure rate reduction compared to state-of-the-art software instruction duplication techniques.

References

[1] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi, “Modeling the effect of technology trends on the soft error rate of combinational logic,” in *Proc. IEEE Int. Conf. Dependable Syst. Netw.*, 2002, pp. 389–398.

[2] T. Karnik and P. Hazucha, “Characterization of soft errors caused by single event upsets in cmos processes,” *IEEE Trans. Dependable Secure Comput.*, vol. 1, no. 2, pp. 128–143, Apr.–Jun. 2004.

[3] P. Hazucha and C. Svensson, “Impact of CMOS technology scaling on the atmospheric neutron soft error rate,” *IEEE Trans. Nucl. Sci.*, vol. 47, no. 6, pp. 2586–2594, Dec. 2000.

[4] R. Baumann, “Soft errors in advanced computer systems,” *IEEE Des. Test Comput.*, vol. 22, no. 3, pp. 258–266, May/Jun. 2005.

[5] IRC, “International technology roadmap for semiconductors 2.0-executive summary,” 2015. [Online]. Available: http://www.itrs2.net/itrs-reports.html. Accessed on: Nov. 19, 2016.

[6] 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 Trans. Electron Devices*, vol. 57, no. 7, pp. 1527–1538, Jul. 2010.

[7] S. Kayali, “Reliability considerations for advanced microelectronics,” in *Proc. Pacific Rim Int. Symp. Dependable Comput*, 2000, pp. 99–102.

[8] S. Feng, S. Gupta, A. Ansari, and S. Mahlke, “Shoestring: Probabilistic soft error reliability on the cheap,” *ACM SIGARCH Comput. Archit. News*, vol. 38, no. 1, pp 385–396, 2010.

[9] L. Spainhower and T. A. Gregg, “Ibm s/390 parallel enterprise server g5 fault tolerance: A historical perspective,” *IBM J. Res. Dev.*, vol. 43, no. 5.6, pp. 863–873, Sep. 1999.

[10] D. Bernick *et al.*, “Nonstop advanced architecture,” in *Proc. IEEE Int. Conf. Dependable Syst. Netw.*, 2005, pp. 12–21.

[11] L. Clark, D. Patterson, C. Ramamurthy, and K. Holbert, “An embedded microprocessor radiation hardened by microarchitecture and circuits,” *IEEE Trans. Comput.*, vol. 65, no. 2, pp. 382–395, Feb. 2016.

[12] N. Oh, S. Mitra, and E. J. McCluskey, “Ed4i: Error detection by diverse data and duplicated instructions,” *IEEE Trans. Comput.*, vol. 51, no. 2, pp. 180–199, Feb. 2002.

[13] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, “Swift: Software implemented fault tolerance,” in *Proc. IEEE Int. Symp. Code Gener. Optim.*, 2005, pp. 243–254.

[14] M. Didehban and A. Shrivastava, “nZDC: A compiler technique for near zero silent data corruption,” in *Proc. 53rd ACM/EDAC/IEEE Annu. Des. Autom. Conf. (DAC)*, 2016, pp. 1–6.

[15] G. A. Reis, J. Chang, and D. I. August, “Automatic instruction-level software-only recovery,” *IEEE Micro*, vol. 27, no. 1, pp. 36–47, Jan./Feb. 2007.

[16] J. Chang, G. A. Reis, and D. I. August, “Automatic instruction-level software-only recovery,” in *Proc. IEEE Int. Conf. Dependable Syst. Netw.*, 2006, pp. 83–92.

[17] I. Laguna, M. Schulz, D. F. Richards, J. Calhoun, and L. Olson, “IPAS: Intelligent protection against silent output corruption in scientific applications,” in *Proc. Int. Symp. Code Gener. Optim.*, 2016, pp. 227–238.

[18] K. Mitropoulou, V. Porpodas, and M. Cintra, “DRIFT: Decoupled compiler-based instruction-level fault-tolerance,” in *Proc. Int. Workshop Lang. Compilers Parall. Comput.*, Springer, Cham, 2013, pp. 217–233.

[19] J. Yu, M. J. Garzaran, and M. Snir, “Esoftcheck: Removal of non-vital checks for fault tolerance,” in *Proc. 7th Annu. IEEE/ACM Int. Symp. Code Gener. Optim.*, 2009, pp. 35–46.

[20] D. S. Khudia, G. Wright, and S. Mahlke, “Efficient soft error protection for commodity embedded microprocessors using profile information,” *ACM SIGPLAN Notices*, vol. 47, no. 5, pp. 99–108, 2012.

[21] J. Xu, Q. Tan, L. Tan, and H. Zhou, “An instruction-level fine-grained recovery approach for soft errors,” in *Proc. 28th Annu. ACM Symp. Appl. Comput.*, 2013, pp. 1511–1516.

[22] J. Yu, M. J. Garzarán, and M. Snir, “Efficient software checking for fault tolerance,” in *Proc. IEEE Int. Symp. Parall. Distrib. Process.*, 2008, pp. 1–5.

[23] K. Mitropoulou, V. Porpodas, and M. Cintra, “Casted: Core-adaptive software transient error detection for tightly coupled cores,” in *Proc. IEEE 27th Int. Symp. Parall. Distrib. Process.*, 2013, pp. 513–524.

[24] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, “Eraser: A dynamic data race detector for multithreaded programs,” *ACM Trans. Comput. Syst.*, vol. 15, no. 4, pp. 391–411, 1997.

[25] C. Lattner and V. Adve, “LLVM: A compilation framework for lifelong program analysis & transformation,” in *Proc. IEEE Int. Symp. Code Gener. Optim.*, 2004, pp. 75–86.

[26] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The splash-2 programs: Characterization and methodological considerations," in *Proc. IEEE 22nd Annu. Int. Symp. Comput. Archit.*, 1995, pp. 24–36.

[27] S. M. Z. Iqbal, Y. Liang, and H. Grahn, "Parmibench-an open-source benchmark for embedded multiprocessor systems," *IEEE Comput. Archit. Lett.*, vol. 9, no. 2, pp. 45–48, Feb. 2010.

[28] N. Binkert *et al.*, "The gem5 simulator," *ACM SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, 2011.

[29] K. Sundaramoorthy, Z. Purser, and E. Rotenberg, "Slipstream processors: Improving both performance and fault tolerance," *ACM SIGPLAN Notices*, vol. 35, no. 11, pp. 257–268, 2000.

[30] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt, "Detailed design and evaluation of redundant multi-threading alternatives," in *Proc. IEEE 29th Annu. Int. Symp. Comput. Archit.*, 2002, pp. 99–110.

[31] J. C. Smolens, B. T. Gold, B. Falsafi, and J. C. Hoe, "Reunion: Complexity-effective multicore redundancy," in *Proc. 39th Annu. IEEE/ACM Int. Symp. Microarch.*, 2006, pp. 223–234.

[32] M. Gomaa, C. Scarbrough, T. Vijaykumar, and I. Pomeranz, "Transient-fault recovery for chip multiprocessors," in *Proc. IEEE 30th Annu. Int. Symp. Comput. Archit.*, 2003, pp. 98–109.

[33] C. Wang, H.-s. Kim, Y. Wu, and V. Ying, "Compiler-managed software-based redundant multi-threading for transient fault detection," in *Proc. Int. Symp. Code Gener. Optim.*, 2007, pp. 244–258.

[34] G. A. Reis, J. Chang, N. Vachharajani, S. S. Mukherjee, R. Rangan, and D. August, "Design and evaluation of hybrid fault-detection systems," in *Proc. IEEE 32nd Int. Symp. Comput. Archit.*, 2005, pp. 148–159.

[35] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, D. I. August, and S. S. Mukherjee, "Software-controlled fault tolerance," *ACM Trans. Archit. Code Optim.*, vol. 2, no. 4, pp. 366–396, 2005.

[36] T. J. Slegel *et al.*, "Ibm's s/390 g5 microprocessor design," *IEEE Micro*, vol. 19, no. 2, pp. 12–23, Mar./Apr. 1999.

[37] B. Stuart and R. Urzi, "A research agenda for mixed-criticality systems."

[38] Z. Chen, A. Nicolau, and A. V. Veidenbaum, "Simd-based soft error detection," in *Proc. ACM Int. Conf. Comput. Frontiers*, 2016, pp. 45–54.

[39] 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 *Proc. 36th Annu. IEEE/ACM Int. Symp. Microarch.*, 2003, pp. 29–40.

[40] L. Borucki, G. Schindlbeck, and C. Slayman, "Comparison of accelerated dram soft error rates measured at component and system level," in *Proc. IEEE Int. Rel. Phys. Symp.*, 2008, pp. 482–487.

[41] J. Meza, Q. Wu, S. Kumar, and O. Mutlu, "Revisiting memory errors in large-scale production data centers: Analysis and modeling of new trends from the field," in *Proc. 45th Annu. IEEE /IFIP Int. Conf. Dependable Syst. Netw.*, 2015, pp. 415–426.

[42] G. A. Reis and D. I. August, "Software fault detection using dynamic instrumentation," in *Proc. 4th Annu. Boston Area Archit. Workshop*, 2006.

[43] A. Martinez-Alvarez, S. Cuenca-Asensi, F. Restrepo-Calle, F. R. P. Pinto, H. Guzman-Miranda, and M. A. Aguirre, "Compiler-directed soft error mitigation for embedded systems," *IEEE Trans. Dependable Secure Comput.*, vol. 9, no. 2, pp. 159–172, Mar./Apr. 2012.

[44] G. A. Chang, R. Jonathan, D. I. August, and R. C. S. S. Mukherjee, "Configurable transient fault detection via dynamic binary translation," in *Proc. 2nd Workshop Archit. Rel.*, 2006.

[45] Y. Zhang, J. W. Lee, N. P. Johnson, and D. I. August, "Daft: Decoupled acyclic fault tolerance," *Int. J. Parall. Programm.*, vol. 40, no. 1, pp. 118–140, 2012.

[46] J. Yu, M. J. Garzarán, and M. Snir, "Techniques for efficient software checking," in *Proc. Int. Workshop Lang. Compilers Parall. Comput.*, 2007, pp. 16–31.

[47] Q. Liu, C. Jung, D. Lee, and D. Tiwari, "Clover: Compiler directed lightweight soft error resilience," *ACM SIGPLAN Notices*, vol. 50, no. 5, 2015, Art. no. 2.

[48] L. Xiong and Q. Tan, "A dynamic approach to tolerate soft errors," *Cluster Comput.*, vol. 16, no. 3, pp. 359–366, 2013.

[49] E. Blem, J. Menon, and K. Sankaralingam, "Power struggles: Revisiting the RISC vs. CISC debate on contemporary arm and x86 architectures," in *Proc. IEEE 19th Int. Symp. High Perform. Comput. Archit. (HPCA2013)*, 2013, pp. 1–12.

[50] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*. Newnes, Australia: Morgan Kaufmann, 2013.

[51] N. Rink and J. Castrillon, "Extending a compiler backend for complete memory error detection," *Automotive-Safety & Security 2017-Sicherheit und Zuverlässigkeit für automobile Informationstechnik*, 2017.

[52] N. A. Rink and J. Castrillon, "Trading fault tolerance for performance in an encoding," in *Proc. Comput. Front. Conf.*, 2017, pp. 183–190.

[53] S. Narayanasamy *et al.*, "Automatically classifying benign and harmful data races using replay analysis," *ACM SIGPLAN Notices*, vol. 42, no. 6, pp. 22–31, 2007.

[54] P. Ratanaworabhan, M. Burtscher, D. Kirovski, B. Zorn, R. Nagpal, and K. Pattabiraman, "Detecting and tolerating asymmetric races," *ACM SIGPLAN Notices*, vol. 44, no. 4, pp. 173–184, 2009.

[55] S. Lu, J. Tucek, F. Qin, and Y. Zhou, "Avio: Detecting atomicity violations via access interleaving invariants," in *Proc. ACM SIGARCH Comput. Archit. News*, vol. 34, no. 5, pp. 37–48, 2006.

[56] M. Bisson, M. Bernaschi, and E. Mastrostefano, "Parallel distributed breadth first search on the kepler architecture," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 7, pp. 2091–2102, Jul. 2016.

[57] C. Sakalis, C. Leonardsson, S. Kaxiras, and A. Ros, "Splash-3: A properly synchronized benchmark suite for contemporary research," in *Proc. IEEE Perform. Anal. Syst. Softw.*, 2016, pp. 101–111.

[58] S. Hu and J. E. Smith, "Using dynamic binary translation to fuse dependent instructions," in *Proc. IEEE Int. Symp. Code Gener. Optim.*, 2004, pp. 213–224.

[59] H. Schirmeier, C. Borchert, and O. Spinczyk, "Avoiding pitfalls in fault-injection based comparison of program susceptibility to soft errors," in *Proc. 45th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, 2015, pp. 319–330.

Authors' photographs and biographies not available at the time of publication.