nZDC: A Compiler technique for near Zero Silent data Corruption

Moslem Didehban
Compiler-Microarchitecture Lab
Arizona State University
Moslem.Didehban@asu.edu

Aviral Shrivastava
Compiler-Microarchitecture Lab
Arizona State University
Aviral.Shrivastava@asu.edu

ABSTRACT

Exponentially growing rate of soft errors makes reliability a major concern in modern processor design. Since software-oriented approaches offer flexible protection even in off-the-shelf processes, they are attractive solutions in protecting against soft errors. Among such approaches, in-application instruction duplication based approaches have been widely used and are deemed to be the most effective. Such techniques duplicate the program assembly instructions and periodically check the results to identify possible errors. Even though early reports suggest that such techniques achieve close to 100% protection from soft errors, we find several gaps in the protection. Existing techniques are unable to protect several important microarchitectural components, as well as a significant fraction of instructions, resulting in Silent Data Corruptions (SDCs). This paper presents nZDC or near Zero silent Data Corruption – an effective instruction duplication based approach to protect programs from soft errors. Extensive fault injection experiments on almost all the unprotected microarchitectural components in simulated ARM Cortex A53, while executing benchmarks from MiBench suite, demonstrate that nZDC is extremely effective, without incurring any more performance penalty than the state-of-the-art.

1. INTRODUCTION

Rapid technology scaling, the main driver of the power and performance improvements of computing solutions, has also rendered our computing systems extremely susceptible to transient errors called soft errors [1, 2]. Among the many sources of transient faults in the system (e.g., electrical noise, external interference, cross-talk, etc.) sub-atomic particles (low and high energy neutrons) that strike on sensitive areas of a transistor cause majority of soft errors in electronic devices [3]. Soft errors have already been attributed to cause large fiscal damages [4]. At the current technology node, soft errors may occur in a high-end server once every 170 hours, but they increase exponentially, and are expected to reach alarming levels of once-per-day[5]!.

While most previous works propose protection from soft errors by altering the hardware of the processor [6], software approaches are attractive since they can be applied to any existing processor; plus, they can be applied more prudently. For example, protection can be turned "on", only for critical applications, or just the critical parts of the application. Among software approaches, in-application instruction duplication is one of the most popular and seemingly effective approaches. In these techniques computational and logical instructions are duplicated with different registers, and at some synchronization points, such as memory operations, compares, branches and function calls, the redundant registers get checked against their original ones. These methods have been shown to be quite effective. The state-of-the-art approach SWIFT[7] claims that they can completely eliminate Silent Data Corruptions (SDCs). In fact, following works [8, 9] argued that SWIFT provides protection, and try to improve performance by applying instruction duplication only for parts of the application.

However, our investigation reveals that the above conclusion is not accurate, and existing techniques are not as effective as advertized. The existing instruction duplication schemes cannot completely protect against: i) faults in the micraoarchitectural resources used by the non-duplicated instructions[1], like loads, stores, compares, branches and function calls. The microarchitectural resources include most components in the processor pipeline, like pipeline registers, fetch queue, functional units, commit queue, reorder buffer etc., ii) faults (or soft errors) in the load store queue, iii) faults in the condition code register, iv) faults in the register file, v) and control flow (CF) errors. The first effect is most significant, since the unprotected non-duplicatable instructions can constitute 20-40% of the program instructions [10]. Also, the unprotected pipeline components constitute a pretty significant fraction of the bits of the core – given that most caches-like structures are already protected in most processors.

In this paper, we propose nZDC or near Zero silent Data Corruption – a software approach to protect applications from soft errors. nZDC is extremely effective because it comprehensively protects: i) the store instructions by reloading the stored value and checking it, ii) the load instructions by duplicating the loads, iii) the compare and branch instructions by duplicating them, iv) detects branches in the wrong direction by using direction check in addition to signature checking, and v) the register file by checking the duplicate registers after a non-duplicated instructions instead of before it.

We implement SWIFT and nZDC in the LLVM compiler for ARM V8-a ISA. We compile the applications from MiBench benchmark suite, and run them on gem5 simulator modeling ARM cortex-53. We perform extensive fault injection experiments in all the major microarchitectural components with state-bits in the gem5 simulator to achieve 95%

[1]Although the name suggests otherwise, existing instruction duplication schemes do not duplicate all instructions.

confidence in our results. We found that unlike SWIFT, nZDC is extremely effective. In fact, compared to original version of the programs, for faults in load store queue, SWIFT increases the average percentage of failure from 3.4% to 3.9%, and reduces the average percentage of failure for functional units, pipeline registers and register file from 9.8%, 16.3% and 10.3% to 1.7%, 1.4% and 0.1%, respectively. However, nZDC eliminates failure due to faults in load-store queue, functional units and register file. And brings down the percentage of failure to 0.3% in pipeline registers. To evaluate the ability of SWIFT and nZDC in detecting control flow errors, we specifically performed fault injections on pipeline registers while carrying control flow (CF) instruction - Branch and Compare instructions. In both SWIFT and the original versions of the programs, on average, 18% of faults on CF instructions result in SDC. On the other hand, only 0.4% of faults on CF instructions lead to SDC in nZDC version of the programs. nZDC achieves this high-level of protection while incurring 10% less performance overhead than SWIFT.

2. BACKGROUND

Software approaches to protect from soft errors have use redundancy at several levels. Replication has been implemented at process-level [11, 12], where the system calls become the checking points. [13, 14, 15] duplicate high-level program statements and insert frequent checking of the results. Compiler-level redundant multi-threading approaches [16, 17] generate two copies of each thread at compilation time, called leader and trailer threads. The leader thread sends the critical values to the trailer thread for checking purpose. The trailer thread compares the values received by leader thread against its own values to detect errors. Instruction duplication can also be applied at the assembly level. ED4I [18] was the first in-application instruction duplication scheme that duplicates all instructions except branches and compares. In ED4I, checking instructions are inserted before stores and branches. Although effective, yet an error in the branch or compare instruction can still corrupt the program's output. In addition, since ED4I requires memory duplication, its performance overhead can be very high, especially for memory-intensive applications.

SWIFT [7] was proposed to improve the performance and fault coverage of ED4I. In SWIFT, all computational instructions get duplicated with different registers and checking instructions being added before memory operations, compare instructions and function calls. In order to reduce the performance overhead and memory pressure, SWIFT replaces the duplicate load by a move from the first load, and assuming that memory is protected by other means, e.g., ECC. SWIFT also proposes a control flow checking (CFC) mechanism to detect control flow errors.
Although there are many following works [8, 9, 19, 20] to SWIFT, it is still considered as the state-of-the-art in terms of protection capabilities of instruction duplication techniques. This is because all the following works concur with the fault detection capabilities of SWIFT, and concentrate on reducing the performance overhead of SWIFT. For example, authors of [8] claim, "SWIFT has the advantage of being purely software-based, requiring no specialized hardware, and can achieve nearly 100% coverage," and then attempt to find out regions of code where symptoms may be used to detect faults, and therefore instruction duplication can be avoided.

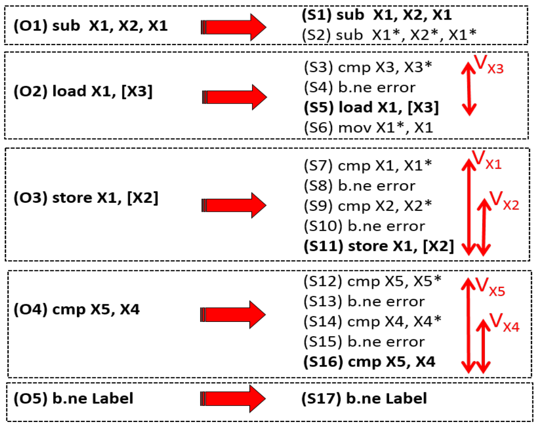


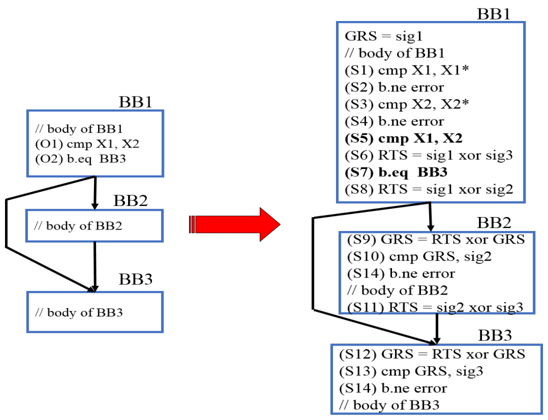
Figure 1: SWIFT transformations. SWIFT duplicates the arithmetic and logical instructions, but does not duplicate loads, stores, compares and branches. The execution of non-duplicated instructions is vulnerable.

Techniques in DRIFT [19] and ESoftCheck [20] also try to improve SWIFT performance without jeopardizing its fault coverage.

3. LIMITATIONS OF SWIFT

We look at protection from a mixed hardware-software perspective. An ideal fault tolerant scheme should be able to completely protect all the microarchitectural components during the execution of all instructions of the program. We analyze the protection offered by SWIFT on various microarchitectural components during the execution of various instructions.

i) The execution of non-duplicated instructions is not protected: Figure 1 shows the basic transformations of SWIFT. SWIFT duplicates the arithmetic and logical instructions. The original sub instruction (O1) is duplicated into two (S1 and S2). However, all other instructions, e.g., loads, stores, compares, branches and even function calls are not duplicated. Figure 1 shows that the load instruction (O2) is not duplicated. The address register of load is checked (against it's duplicate) before the execution of the load (S3), and error if any is reported (S4) and the loaded value is moved to the duplicate register (S6). The store instruction (O3) is also not duplicated (only one store, S11). It's operands are checked (against their duplicates) before the store (S7, S9) and error if any is reported (S8, S10), and then the store (S11) is performed. The compare instruction (O4) is also is not duplicated. The operands of the compare are checked (S12, S14), error if any is reported (S13, S15), and then the comparison is performed (S16). While the execution of duplicated instructions is protected, but the execution of non-duplicated instructions is not completely protected. This means, that if there is a soft error in the microarchitectural resources that these instructions use, then the error may go undetected. For example, if there is a soft error in the pipeline register during the execution of the compare instruction (S16), then that error may not be detected, and only cause an SDC. Also note that the branch instruction is also not duplicated. To gauge the magnitude of this unprotected instructions, we calculated the fraction of the non-duplicated instructions in all the committed instructions in MiBench benchmarks for a ARM-V8 architecture, and found that to be about 18%, even when we compile them



```
// body of BB1
(O1) cmp X1, X2
(O2) b.eq  BB3
```

```
// body of BB2
```

```
// body of BB3
```

```
GRS = sig1
// body of BB1
(S1) cmp X1, X1*
(S2) b.ne error
(S3) cmp X2, X2*
(S4) b.ne error
(S5) cmp X1, X2
(S6) RTS = sig1 xor sig3
(S7) b.eq  BB3
(S8) RTS = sig1 xor sig2
```

```
(S9) GRS = RTS xor GRS
(S10) cmp GRS, sig2
(S14) b.ne error
// body of BB2
(S11) RTS = sig2 xor sig3
```

```
(S12) GRS = RTS xor GRS
(S13) cmp GRS, sig3
(S14) b.ne error
// body of BB3
```

Figure 2: SWIFT CFC: SWIFT uses statically generated signatures for CF checking. By definition, the checking has to allow for error-free execution in either direction of a branch. As a result, SWIFT cannot detect wrong directed branches.

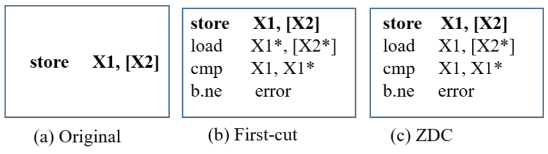
using the -O3 optimization flag. Moreover, to apply SWIFT the program has to be compiled with only half of the registers, since the other half are needed for duplicate registers. This causes register spilling, and increases the number of loads and stores in the program. Furthermore, the microarchitectural components used by these instructions include the pipeline registers, fetch queue, reorder buffer, commit queue, Functional Units(FUs) etc. These comprise a major portion of the processor state, given that the bigger cache-like components, including branch predictors and TLBs are routinely protected in modern processors [21].

ii) The load store queue in the processor is not protected: Since only load store instructions use the load store queue, and their execution is not protected, it follows that errors in load store queue may go undetected.

iii) Wrong direction branches cannot be detected: A wrong direction branch occurs, if the control flow (CF) of a program alters in a way that a taken branch changes to a non-taken or vise versa. A wrong direction branch may be caused by soft error on four components: (a) registers which hold the operands of a compare instruction,(b) pipeline registers while executing a compare instruction, (c) conditional code register and (d) pipeline registers while holding the opcode of a branch instruction. However, SWIFT CFC mechanism can just partially protect the first component and the reset remains undetected. Figure 2(b) shows the CFC scheme in SWIFT. SWIFT uses basic block signatures to detect CF errors. These signatures are assigned to the basic blocks statically. In the figure, the basic blocks BB1, BB2 and BB3 are assigned signatures $sig1$, $sig2$, and $sign3$ respectively. By definition, the signature checking has to allow CF changes from BB1 to either BB2 or BB3. As a result, if soft error causes a wrong direction branch, then SWIFT CFC cannot detect that; and it may end up as a SDC.

iv) Wrong target branches may not be detected: If an error affects the target address of a branch, a wrong target branch happens. For example, if error happens on functional unit registers while computing the effective address of branch or occurs on branch target address buffer, CF of a program transfers to a wrong address. The ability of SWIFT CFC on detecting these type of errors is extremely restricted. In fact, just wrong target branches whose destination is the beginning of a basic block can be detected by SWIFT CFC. It cannot detect branching to the middle of any basic block, because the signature checking, which takes place at the beginning of the basic block, is already passed.

v) Register file is not completely protected: Although SWIFT checks the duplicates before every non-duplicatable instruction, these registers may be vulnerable from the check to the actual non-duplicatable instructions. Lets take the example of the transformation for the store instruction (O3). First the duplicates are compared (S7, S9), and error if any is reported (S8, S10), before executing the store instruction (S11). However, the register X1 is vulnerable from after the comparison (S7) to the execution of the store instruction (S11), i.e., from S7 – S11(pointed to by the vertical line segment V_{X1} in figure 1), and register X3 is vulnerable from S10 – S11. Authors of SWIFT themselves have acknowledged this vulnerability of register file [22, 23].



```
store    X1, [X2]
```

(a) Original

```
store    X1, [X2]
load     X1*, [X2*]
cmp      X1, X1*
b.ne     error
```

(b) First-cut

```
store    X1, [X2]
load     X1, [X2*]
cmp      X1, X1*
b.ne     error
```

(c) ZDC

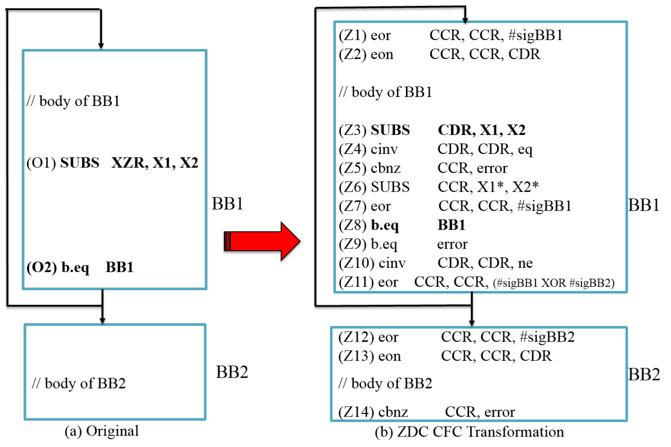
Figure 3: nZDC store checking mechanism

4. OUR APPROACH - NZDC

In this section, we present nZDC, a compiler approach to almost eliminate SDCs. In-line with previous works, we assume that soft errors can modify the data within the CPU but memory and caches are protected by other orthogonal techniques such as ECC. The salient features of nZDC are:

i) Protect stores by reloading and checking: nZDC introduces the concept of “checking load instruction” to make sure that the store instruction has executed fault free. The main idea here is to load back the stored value from the memory and check that against the stored value, if they match there is no error otherwise error handler routine get involved. Figure 3(b) shows the first-cut of our approach to insert checking load instruction. Although it can detect errors which affect the address part of the store instruction, yet errors on value part can simply propagate from the store's value register to checking load value registers, and remain undetected. For instance, if soft error alters the value of X1, store instructions writes this corrupted value into memory. The checking load instruction, loads the corrupted value in X1*. Now, both X1 and X1* have same wrong values, and comparing them can not catch the error. This happens because the value register of checking load instruction is the shadow of the value register of store instruction. nZDC protects stores by using the same value register for both store and checking load instructions and later check that register against its shadow for soft error (Part (c) of figure 3). By this method, in addition to the producer chains of store value and address registers, the execution of store instruction itself is also protected.

The performance overhead of nZDC solution for store instruction may seem high at first, but thanks to store-to-load forwarding mechanism in LSQ of modern microprocessor, the checking load instructions normally take their values from the store buffer and execute very fast. The only problem here is, if error happens on store buffer after that the store forwarded its data to the checking load instruction,


```
(a) Original

BB1:
// body of BB1

(O1) SUBS   XZR, X1, X2

(O2) b.eq   BB1

BB2:
// body of BB2
```

```
(b) ZDC CFC Transformation

BB1:
(Z1) eor      CCR, CCR, #sigBB1
(Z2) eon      CCR, CCR, CDR

// body of BB1

(Z3) SUBS     CDR, X1, X2
(Z4) cinv     CDR, CDR, eq
(Z5) cbnz     CCR, error
(Z6) SUBS     CCR, X1*, X2*
(Z7) eor      CCR, CCR, #sigBB1
(Z8) b.eq     BB1
(Z9) b.eq     error
(Z10) cinv    CDR, CDR, ne
(Z11) eor   CCR, CCR, (#sigBB1 XOR #sigBB2)

BB2:
(Z12) eor     CCR, CCR, #sigBB2
(Z13) eon     CCR, CCR, CDR

// body of BB2

(Z14) cbnz    CCR, error
```

Figure 4: nZDC CFC transformation

the error may remain undetected. This unprotected interval can be completely removed in two ways; i) flush store buffer after each store, or ii) use ECC in store buffer. The former comes with significant performance degradation; it is similar to the case that there is no store buffer at all. The second approach may not have performance overhead and it does not require any extra hardware, but the ECC code should be generated before the stores arrive at the store buffer.

ii) Protect loads by duplicating them: Memory read instructions are the most frequent unprotected instruction in SWIFT and many other software redundancy based techniques [9, 20, 17]. These instructions behave such as the input of duplicated instructions chain, and if a memory read instruction gets faulty, the execution can go wrong and checking instructions are unable to detect such an error.

Our solution for memory read instructions is simple, we use ”load” duplication, which in memory read instructions get duplicated as well as logical and computational instructions. The load instruction duplication has two advantages; 1) it protects load instructions completely during their execution in pipeline, LSQ and removes load-related register file vulnerable intervals (Marked as V_{X3} in figure 1), and 2) it saves performance overhead by reducing the number of checking instructions. The load instruction duplication does not introduce any problem in single-threaded applications running on single or multiprocessor machines, which are the target of this work. However, it can slightly increase the risk of false alarm in a multithreaded environment, if an intervening store changes the state of the memory. A solution for this can be putting redundant loads in the critical section.

iii) nZDC control flow checking mechanism detects wrong direction branches: An effective CFC mechanism should be able to protect all of the CF determining parts of the execution; which are a) Operands of compare instructions, b) pipeline registers while executing compare instruction, c) conditional registers (NVCZ flags) and d) branch instructions. Against the existing CFC mechanism which can partially protect some of these parts, nZDC CFC mechanism can effectively protect all CF determining components. nZDC CFC mechanism needs two general purpose registers, called CDR (Compare Destination Register) and CCR (Compare Check Register) and it works based on three main insights: a) duplicate compare instructions and save the results in CDR and CCR, b) conditionally inverse the value of CDR register based on the direction of the following conditional branch and branch duplication, and c) using static signatures to make sure that the destination block is legal.

Figure 4 shows nZDC CFC transformation for a loop. The original compare instruction (O1) is in fact a subtraction operation which updates the NVZC flags and disregards the output. First, nZDC CFC transformation saves the result of subtraction (compare) operation in CDR register (Z3), and replicates the compare instruction (Z6). In a fault free run of a program, the results of duplicated compare instructions are always equal. If error happens on registers which hold the operands of a compare instruction or pipeline registers while executing a compare instruction, the CDR and CCR would be differ and the error will get detected by nZDC CFC.

In the second step, nZDC CFC transformation takes advantage of ISA provided conditional instruction, and conditionally inverts the value of CDR register before the branch (Z4), if it is taken, and after the branch(Z9), if it is not taken. Note that the condition of these two conditional instructions are always opposite. Therefore, the values of CDR and CCR registers should be always inverse of each other at the beginning of the next basic block (BB1 or BB2), regardless of the direction of branch. Since the conditional invert instruction (Z4) and conditional branch (Z8) read condition resister value in two different instants of time, which is set by two different instructions (main and redundant compare instructions), error on condition register results zero or two inversions on CDR register. Therefore, at the beginning of next basic block the values are CDR and CCR registers are not inverse, and nZDC CF checking instructions can detect the error. Errors which happens on the opcode of a conditional branch and alters the branch from not-taken to taken, will end up with no inversion on CDR register, which will get detected by nZDC CF checking instructions. nZDC CFC introduces the redundant branch instruction (Z9) to detects errors which change the direction of branch from taken to non-taken. The idea behind the nZDC redundant branch is that it should not changes the CF of the program in a fault-free run. If main branch is taken, the control never reaches the redundant branch, and if main branch is not-taken, the redundant branch should also be non-taken. But, if error happens on the opcode of main branch and changes the branch from taken to not-taken, the redundant branch involves error handler routine.

iv) nZDC control flow checking mechanism detects wrong target branches: nZDC CFC instructions is inserted at the beginning and near to the end of each basic block in the third step of nZDC CFC transformation. Having this that the result of an XNOR (exclusive NOR) operation of two invert values is always Zero, at the beginning of each basic block, nZDC CFC XNORs the CCR and CDR registers together and put the results on CCR (Z2 and Z13). In a fault free run the value of CCR should be zero at this point. Right before the next write to the CCR register, nZDC CFC checks if its value is Zero (Z5 and Z14), if not, error handler routine gets involved. The distance between setting CCR value and checking that, enables nZDC CFC to detect errors that change the branch target to the body of a basic block. Also, nZDC CFC uses static signature to check if the next basic block is legitimate (Z1, Z7, Z11, Z12).

v) nZDC eliminates all register file vulnerable inter-

Table 1: Simulator parameters

Parameter	Value
CPU Model	ARM64 bit in-order processor
Pipeline	Two way/4-stage
# of FUs	2Int, 1Mul, 1Div, 1Float, 1Mem
L1 D/I-Cache	64KB (2-way) / 32KB (2-way)
TLB size	512 entries
Integer register file	32 registers (64-bit width)
Store buffer size	5 entries

vals: Since nZDC checks master registers against the redundant ones after the store instruction, rather than before; it eliminates the register file vulnerability intervals caused by store operation. In the case of function calls, nZDC checks all master registers against the redundant ones in the beginning of the callee function rather than before the function call.

5. EXPERIMENTAL METHODOLOGY

We have performed extensive fault injection testing to evaluate the effectiveness of nZDC and SWIFT in reducing Silent Data Corruption. **Compilation framework**: We have implemented nZDC and SWIFT transformations as late backend passes in LLVM3.7 compiler infrastructure after register allocation and instruction scheduler. This implementation enables us to take advantage of all of the advanced compiler optimizations including Common Sub-expression Elimination (CSE) and Dead Code Elimination (DCE). We test the effectiveness of SWIFT and nZDC on applications from the Mibench benchmark suite. **Simulation environment**: We have used the GEM5 [24] - a popular cycle-accurate microarchitectural simulator. The simulator was run in ARM syscall emulation mode and modeled the ARMv8-a profile of the ARM64 bit architecture. We have used a two-way in-order ARM architecture for fault injection experiments and the details of the processor configuration can found in Table 1. The simulated CPU model is very close to "cache protection configuration" of ARM Cortex-A53, a popular modern high performance low power embedded microprocessor. In the this configuration of ARM cortex-A53 the memory subsystem including TLB, instruction and data L1 caches, and L2 data cache are protected with error detection and correction codes. **Fault sites:** Instead of injecting faults just on processor's register file, we inject faults on all the major sequential component of the processor. For the in-order ARM, this includes the pipeline registers, load-store queue, functional units and the scoreboard. All the other components are either not vulnerable (e.g., the branch predictor), or are already assumed protected (e.g., the caches and the TLBs). In addition, to show the effectiveness of the nZDC control-flow checking, we specifically perform fault injection on the branch and compare instructions while there are in the processor's pipeline. **Fault injection experiments:** For each fault site, a random bit in a random time is selected and is inverted. For example in the case of register file, at the start of each experiment a physical register, a bit and a cycle is selected randomly for fault injection. Simulation runs the program normally till the selected cycle. Then the value inside the selected bit gets inverted, and the program runs until completion or allowable simulation time (which is 10 times the nominal execution time) gets over. For each fault site, 400 faults are injected which gives us a 5% margin of error and 95% confidence interval [25]. This means, that for each version of the program (original, SWIFT and nZDC), 2400 fault injection experiments are performed. Among them 2000 faults are injected into register file, pipeline registers, load store queue, functional units and scoreboard, and the rest, 400 faults, are specifically injected into the main branch and compare instructions. Overall, we inject 72,000 faults in various components of the processors. **Output classification:** Since the main goal of this work is to prevent a program from producing the wrong output because of soft errors, the result of each fault injection trial is classified into only two categories: *1. SDC:* Those simulation runs which terminated without generating any detection alert, segmentation faults or crash, and *2. Others:* All other scenarios i.e, masked faults, detected fault, segmentation fault and crash fall into this category.

6. EXPERIMENTAL RESULTS

6.1 nZDC is extremely effective

The graphs in figure 5 present the Failure Percentage (FP) for each hardware component. In each graph, the FP is plotted on the Y-axis for each benchmark on the X-axis. We study the FP for original, SWIFT and nZDC versions of the benchmarks for the following components:

Load Store Queue(LSQ): Figure 5(a) presents the FP extracted from fault injection experiments on LSQ. For the LSQ, although the FP is 3.4% for the original program, but it actually increases to 3.9% for SWIFT. As mentioned in section 4, SWIFT does not protect the loads and stores - they are executed only once. Therefore the LSQ is vulnerable. However, we observe that it is actually more vulnerable than the original program. This is because to implement SWIFT, we need to reserve half of the registers in the processor, and that increases the register pressure and increases the spill code - causing a spike in the number of load and store instructions. This leads to an increase in the number of entires in the LSQ, and therefore the probability that a fault will cause an error/failure.

nZDC has 0% failure. nZDC is effective, since it protects loads by duplicating them, and protects the stores by reading the stored value again and checking it against the duplicate.

Functional Units (FUs): The result of fault injection on FUs is shown in graph (b) of the figure 5. For FUs, the average FP for original and SWIFT versions of the programs are 9.8% and 1.7%, respectively. We explored the failure experiments in SWIFT, and discovered almost all of them are the result of faults affecting the FU while computing effective address of memory instructions. As expected, Zero SDC for programs protected with nZDC mechanism is observed.

Pipeline Registers: Figure 5(c) presents the FP extracted from fault injection trails on pipeline registers. As it shows, nZDC reduces the FP of the original program by about 55X to fairly close to zero, and SWIFT reduces pipeline registers FP by about 11X. In contrary with what we expect, the FP for nZDC in not absolutely zero. This discrepancy is not a deficiency of our technique, but has its roots in our evaluation methodology. We analyzed the failures, and found that the reason is because of working with unmodified library calls. The soft error happens on the destination register pointer part of a checking instruction before a library call, and the fault changed the destination register from regis-

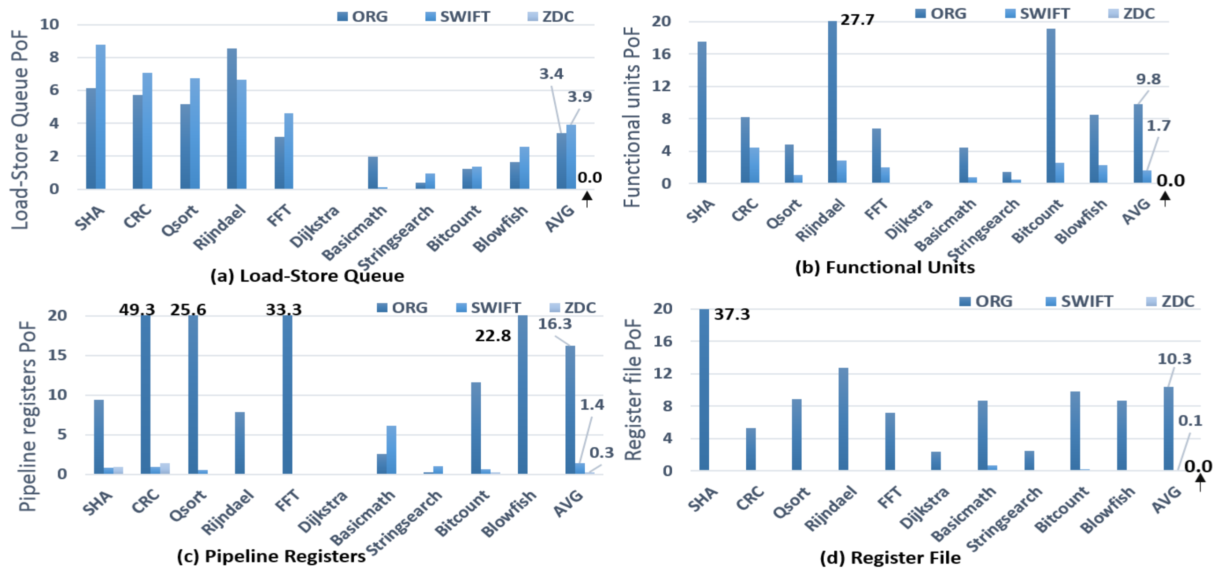


Figure 5: Component wise Failure Percentage (FP)

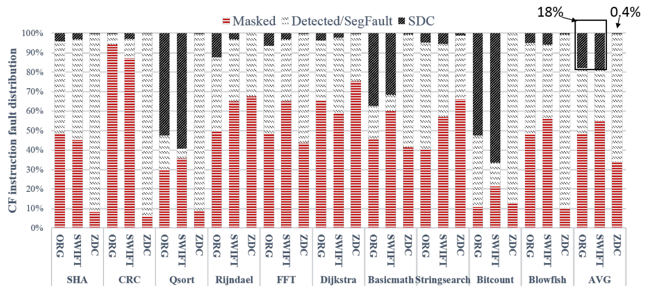


Figure 6: Fault injection results on CF instruction

ter Zero to the register X1, which was already checked. As a result the argument value of the library function call was wrong and it produces a SDCs. These errors would not happen when all code, including libraries is treated by nZDC.

Register File: Fig 5 part (d) displays the FP for register file, which on average for Original, SWIFT and nZDC is about 10.3%, 0.1% and 0.0%, respectively. Our fault injection results on register file is in accordance with previous works [7, 8, 19] which show almost near zero SDC for SWIFT. However, because of the register file vulnerable periods (marked by vertical lines in figure 1), there is always a chance of SDC caused by soft error in the register file. On the other hand, since nZDC can completely close the register file vulnerable intervals by performing checking instructions after memory write instructions instead of before them, the PoF is Zero.

We also inject faults on various field of the Scoreboard, but since all of our injected faults get masked, we conclude that the Scoreboard has a very low sensitivity to transient faults.

6.2 The efficacy of nZDC CF mechanism

Figure 6 shows the results of fault injection on the branch address and compare instructions of the programs to examine the efficacy of the nZDC control flow mechanism. In these experiments we randomly inject faults on the original compare and branch instructions (excluding checking instructions) while they are in pipeline register, and conditional registers. Since the target of fault injection has been selected among the program original instructions, in figure 6 we show the amount of Masked, Detected/SegFault and SDCs. The Detected/SegFault portion of the stacked bars demonstrate the percentage of injected faults which either detected by SWIFT/nZDC or detected by OS as segmentation fault.

As figure 6 demonstrates original and SWIFT are almost identical in failure rate, about 18% SDCs! This is because, SWIFT CFC can just only detect wrong direct branches to the beginning of a basic block not to the middle. Overall, the SWIFT CF mechanism is not effective, however, nZDC CF mechanism detects about 55% of faults and just 0.4% of faults lead to SCDs.

6.3 Performance evaluation

Figure 7 presents the results of performance overhead of nZDC and SWIFT for an in-order ARM processor with the configuration shown in table 1. On average, the execution time overhead for nZDC and SWIFT is about 224% and 213%. Performance overhead is higher than similar works, and it happened because of an inaccuracy that some of previous works have in their performance measurement. For instance, research [8] assumed that library functions are protected by other means, but they do not consider the performance overhead of the library function protection, and, since a program can spend a considerably large amount of its execution time in the library calls, this leads to performance overhead underestimation. However, in this work, for performance overhead evaluation, we just consider the cycles that a program spends in user functions.

7. CONCLUSIONS

We presented nZDC, a compiler-only instruction duplication fault tolerant technique that completely protects the execution of programs against soft errors on various hardware components. nZDC is based on this idea that non-duplicated

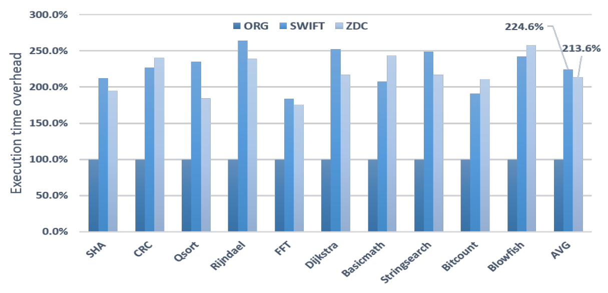


Figure 7: Execution time overhead for SWIFT and nZDC

instructions open doors for program failures and by duplicating all instructions near zero failure rate is achievable in software. However, since duplicating store and branch instructions is problematic, nZDC introduces checking load instruction and a new control flow checking mechanism. Checking load instructions are inserted after stores, and check the stored value against its shadow. By duplicating compare and branch instructions, nZDC control flow checking is able to detect almost all control flow errors. Our comprehensive evaluation based on random fault injection on various microprocessor components shows significant failure rate reduction compared to the state-of-the-art software instruction duplication technique.
In this work, we inject faults into major microarchitectural structures of an in-order CPU model, but since these structures also exist in modern Out-of-Order(OoO) processors, we expect that an OoO model wouldn't affect our conclusions significantly. In fact, unprotected instructions are also vulnerable in OoO microprocessor structures such as Reorder Buffer and Rename table in existing instruction duplication techniques, but nZDC can protect them. Fault injection on an OoO processor is our future work.

References

[1] P. Shivakumar *et al.*, "Modeling the effect of technology trends on the soft error rate of combinational logic," in *Dependable Systems and Networks, 2002. Proceedings. International Conference on*, IEEE.

[2] T. Karnik *et al.*, "Characterization of soft errors caused by single event upsets in cmos processes," *Dependable and Secure Computing, IEEE Transactions on*, vol. 1, no. 2, 2004.

[3] R. Baumann, "Soft errors in advanced computer systems," *Design & Test of Computers, IEEE*, vol. 22, no. 3, 2005.

[4] D. Lyons, "Sun screen: Soft error issue in sun enterprise servers," 2000.

[5] S. Kayali *et al.*, "Reliability considerations for advanced microelectronics," in *prdc*, IEEE, 2000.

[6] S. Mukherjee, *Architecture design for soft errors*. Morgan Kaufmann, 2011.

[7] G. A. Reis *et al.*, "Swift: Software implemented fault tolerance," in *Proceedings of the international symposium on Code generation and optimization*, IEEE, 2005.

[8] S. Feng *et al.*, "Shoestring: probabilistic soft error reliability on the cheap," in *ACM SIGARCH Computer Architecture News*, vol. 38, ACM, 2010.

[9] D. S. Khudia *et al.*, "Efficient soft error protection for commodity embedded microprocessors using profile information," in *ACM SIGPLAN Notices*, vol. 47, ACM, 2012.

[10] E. Blem *et al.*, "Power struggles: Revisiting the risc vs. cisc debate on contemporary arm and x86 architectures," in *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, IEEE.

[11] A. Shye *et al.*, "Plr: A software approach to transient fault tolerance for multicore architectures," *Dependable and Secure Computing, IEEE Transactions on*, vol. 6, no. 2, 2009.

[12] Y. Zhang *et al.*, "Runtime asynchronous fault tolerance via speculation," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, ACM, 2012.

[13] M. Rebaudengo *et al.*, "Soft-error detection through software fault-tolerance techniques," in *Defect and Fault Tolerance in VLSI Systems, 1999. International Symposium on*, IEEE.

[14] M. Rebaudengo *et al.*, "A source-to-source compiler for generating dependable software," in *Source Code Analysis and Manipulation, 2001. Proceedings. First IEEE International Workshop on*, IEEE.

[15] J. Lidman *et al.*, "Rose:: Fttransform-a source-to-source translation framework for exascale fault-tolerance research," in *Dependable Systems and Networks Workshops, 2012 IEEE/IFIP 42nd International Conference on*, IEEE.

[16] C. Wang *et al.*, "Compiler-managed software-based redundant multi-threading for transient fault detection," in *Proceedings of the International Symposium on Code Generation and Optimization*, IEEE, 2007.

[17] Y. Zhang *et al.*, "Daft: decoupled acyclic fault tolerance," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, ACM, 2010.

[18] N. Oh *et al.*, "Ed 4 i: error detection by diverse data and duplicated instructions," *Computers, IEEE Transactions on*, vol. 51, no. 2, 2002.

[19] K. Mitropoulou *et al.*, "Drift: Decoupled compiler-based instruction-level fault-tolerance," in *Languages and Compilers for Parallel Computing*, Springer, 2014.

[20] J. Yu *et al.*, "Esoftcheck: Removal of non-vital checks for fault tolerance," in *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, IEEE, 2009.

[21] `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0500d/DDI0500D_cortex_a53_r0p2_trm.pdf/`. [Online].

[22] G. A. Reis *et al.*, "Software-controlled fault tolerance," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 2, no. 4, 2005.

[23] G. A. Reis *et al.*, "Design and evaluation of hybrid fault-detection systems," in *ISCA, 2005*, IEEE.

[24] N. Binkert *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, 2011.

[25] R. Leveugle *et al.*, "Statistical fault injection: quantified error and confidence," in *Design, Automation & Test in Europe Conference & Exhibition, 2009.*, IEEE.