NEMESIS: A Software Approach for Computing in Presence of Soft Errors

Moslem Didehban Sai Ram Dheeraj Lokam Aviral Shrivastava
Compiler-Microarchitecture Lab, Arizona State University
Moslem.Didehban@asu.edu, Dlokam@asu.edu, Aviral.Shrivastava@asu.edu

ABSTRACT

Soft errors are considered as the main reliability challenge for sub-nanoscale microprocessors. Software-level soft error resilience schemes are desirable because they are applicable on the existing processor, and their protection can be tuned based on the application requirements. However, existing software-level error tolerant schemes do not provide high-level of protection. In this work, we present NEMESIS – a compiler-level fine-grain soft error detection, diagnosis and recovery technique that can provide high degree of error-resiliency. NEMESIS runs three versions of computations and detects soft errors by checking the results of all memory write and branch operations. In the case of mismatch, NEMESIS diagnosis routine decides if the error is recoverable. If yes, NEMESIS recovery routine reverts the effect of error from the architectural state of the program and program resumes its normal execution. Our extensive μ-architectural-level fault injection experiments results show that NEMESIS transformation is able to detect all soft errors and recover from 97% of detected errors.

1. INTRODUCTION

The ever-increasing use of digital systems in everyday life has made reliability as a key factor in modern microprocessors. Soft errors – caused by high-energy particles, power supply noises, transistor variability and etc.– can modify the logic value stored in microprocessor memory element(s) and cause a timing or functional failure. Historically, soft errors were considered as a challenge for high-attitude applications because most of the high-energy particles get cascaded by the earth's atmosphere before they reach at ground level.However, as ITRS 2015 [14] predicts, soon even terrestrial-level muon-induced particles can causes soft errors in the microprocessors.

Hardware-level techniques are quite popular enterprise servers and high-critical applications to mitigate the effect of soft errors [3, 9]. While hardware solutions can be effective, they require the building of special-purpose hardware, that will be expensive until they become mainstream. Software solutions to protect from soft errors on the other hand, can be applied on commercial off-the-shelf processors. They will be applicable to all past and future processors. Plus software protection approaches can be applied selectively – either to only the safety/mission critical applications, or only to the critical parts of an application. Software solutions are especially attractive for mixed-criticality systems like automobiles where the high-critical tasks (e.g. anti-lock braking and air-back control tasks) and less-critical ones (e.g. entertainment tasks) share the same underlying processor [27].

Redundancy-based techniques [11, 22, 10, 13, 16, 6, 15] have been considered as the most effective soft error protection schemes. Depending on the recovery strategy, existing software fault tolerant schemes can be categorized into backward and forward recovery schemes. Many schemes [7, 10, 30, 31, 21, 20] have developed only for error detection. We reckon that all these techniques assume some kind of backward recovery. There are 2 main kinds of backward-recovery techniques, one is restart, and second is checkpointing and rollback. While restart-based recovery techniques can be useful for some small applications, but they are not useful in many cases, including for hard real-time, long-running and interactive applications. Checkpointing can solve the problems of global restarting by periodically saving a snapshot of the programs architectural state – memory and register state(checkpoint). In case of error, program rolls back to the last saved checkpoint and re-executes the instructions from the checkpoint. However, due to their significant overhead the usage of full checkpoint-based recovery techniques is limited to HPC (high performance computing) applications [8, 1, 11]. Some researchers [11, 19] have proposed light weight checkpointing techniques. Unfortunately, such techniques offer a limited error recovery. For instance, EnCore[11] scheme can not provide recovery for the errors which affect the address of memory write instructions or modify the control flow of the program.

In contrast to the backward recovery, forward recovery schemes do not implicitly detect errors; they mask errors by applying majority-voting between redundantly-computed results. Coarse-grain techniques like PLR [26], which perform infrequent voting on the arguments of system-call arguments cannot provide protection in cases that a pointer is in the list of system call arguments. This is because PLR verifies the correctness of redundant computed pointers, but not of the data that is actually stored into the memory. On the other hand, fine-grain forward recovery schemes perform voting operations on specific point of execution and can get the best from ECC-protected components like cache and memory subsystem. For instance, Swift-R [22, 5]

triplicates the arithmetic/logical instructions in a program and performs 2-of-3 majority-voting for register operands of critical instructions, i.e., memory and control flow instructions. Swift-R is the-state-of-the-art fine-grain error tolerant scheme in terms of error coverage. Works after Swift-R, only try to improve the performance overhead of Swift-R, while assuming that Swift-R error coverage is enough. For instance, ELZAR [15] utilized SIMD instructions for fast execution of triplicated instructions. Selective Swift-R [23] trades off the coverage of Swift-R for performance by applying Swift-R transformation to some parts of program.

However, our detailed analysis of Swift-R based techniques reveals that such schemes have quite restricted error coverage. The main reason is that SWIFT has *always-on voting* – meaning that it performs voting of the operands before all critical instructions. This causes two main problems: i) the critical instructions themselves are executed only one, and therefore are not protected. If any error occurs during the execution of critical instructions (on average 45% of instructions in Swift-R protected programs) it remains undetected (and of course unrecovered-from). ii) frequent voting operations introduces vulnerable intervals for the operands of critical instructions and imposes significant performance overhead.

Motivated by these limitations, we propose NEMESIS: a fine-grain software approach that achieves a high degree of reliability by performing active error detection and handling. Instead of *always-on voting*, **before** the execution of critical instructions, NEMESIS performs error detection **after** the execution of the critical instructions. As a result, NEMESIS protects the execution of critical instructions. Also, since there is no voting, the vulnerable intervals created due to voting are also eliminated. Finally, the overhead associated will voting is reduced to the overhead of detection. Once NEMESIS detects an error, it analyzes the extend of error propagation. Contingent upon the scope of error propagation, NEMESIS decides whether the error is recoverable. If yes, NEMESIS recovery routine reverses the effect of error from memory (by writing back the memory backup into the memory) and architectural registers (by performing majority voting between redundant registers).

To evaluate the effectiveness of our proposed scheme, we performed about 15 million μarchitectural-level single bit-flip fault injection experiments on different hardware resources of a ARM-cortex A53-like simulated microprocessor. The results demonstrate that no injected fault lead to SDC (Silent Data Corruption or wrong output) in NEMESIS-protected programs, while about 237k and 120k of faults cause SDCs in original and Swift-R protected versions of the programs, respectively. Additionally, NEMESIS-protected programs, on an average execute 25% faster than Swift-R protected ones.

2. LIMITATIONS OF SWIFT-R

Swift-R [22, 5] is the state-of-the-art fine-grain forward error recovery technique in terms of coverage. It

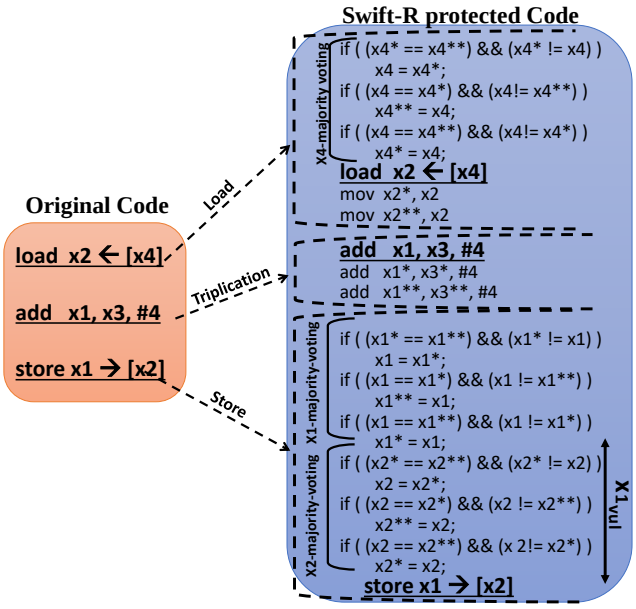

Figure 1: Swift-R transformation: Swift-R checks the register operands of the critical (load/store/compare/branch) instructions, and triplicates the arithmetic instructions.

divides programmer-visible registers into three sets and triplicates computational instructions. In an attempt to prevent the propagation of soft errors to the memory subsystem, Swift-R performs 2-of-3 majority-voting between redundant-computed values of source register operands of memory and compare instructions, just before their execution. Figure 1 shows Swift-R transformation for a simple piece of code. It shows that majority-voting is performed between the redundant-computed values of `load` address registers before its execution (marked as `x4-majority-voting`). The loaded value (`x2`) is then copied into the corresponding redundant registers (`x2*` and `x2**`). The `add` instruction is triplicated with redundant registers. Before the execution of store, two majority-voting operations, one for the store value register(`x1`) (marked as `x1-majority-voting`) and one for store address register (`x2`) (marked as `x2-majority-voting`), are performed. Swift-R transformation suffers from the following problems:

i) The execution of the critical instructions ($\sim$ 45% of program operations) is not protected: Swift-R majority-voting operations eliminate the effects of the errors which may incur during the execution of computational instructions. However, if error occurs during the execution of any memory and control-flow instruction, it remains unprotected. For instance, if the soft error happens on the pipeline registers during the execution of `load` instruction, the effective address of load may be modified to an arbitrary value and ultimately a wrong value will be loaded into the `x2` register. Once this happens, Swift-R copies the erroneous value into the corresponding redundant registers(`x2*`, and `x2**`), making the state of all three streams consistently wrong. These type of errors can lead to a failure in Swift-R protected programs. The same problem can happen during the execution of all critical instructions

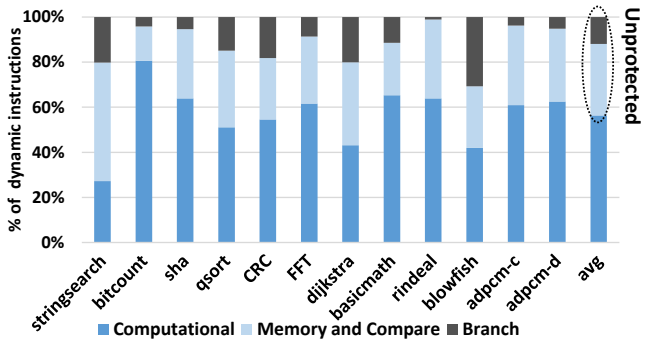

Figure 2: Swift-R transformation cannot protect the execution of more than 45% of dynamic instructions.

that are executed only once, including store, compare and branch instructions, and Swift-R scheme does not verify their correct execution. To quantify the severity of this problem, we collect the number of dynamic critical and triplicated instructions from Mibench benchmark programs (figure 2), after reserving registers for Swift-R redundant instructions. The figure shows that on an average, about 55% of dynamic instructions are arithmetic instructions, and can be triplicated and protected by Swift-R. But, about 45% of total instructions are critical which Swift-R transformation fails to protect. Note that Swift-R scheme significantly increases the register pressure and causes more spill-code (load and store to the stack).

ii) Voting operations introduces vulnerability: Software implementation of majority Voting requires several compare and branch instructions (see `x1-majority-voting` and `x2-majority-voting` in figure 1) and base on implementation may demand 4 to 10 machine instructions. Swift-R frequent voting operations introduce unprotected intervals even for the operands of critical instructions. Particularly, if soft error happens on registers that are carrying the operands of the critical instructions, after the operands are checked, then it can cause critical instructions to execute incorrectly. For example, if an error happens on the register `x1` after the last access by `x1-majority-voting` operation and before it gets accessed by the `store` instruction, the wrong value will be written into the correct memory location. This vulnerable period is marked by `x1_vul` vertical line in figure 1. This is significant, owing to the length of the vulnerability window, and the frequency of voting (one or two times before all load, store, compare and function call instructions).

Overall, the Swift-R sphere of protection is restricted to the execution of programs triplicated instructions. All works after Swift-R mainly assume that Swift-R can achieve high-level of protection and they try to reduce Swift-R performance overhead. In this work, however, our goal is to improve the error coverage of Swift-R and improve its performance overhead at the same time.

3. NEMESIS: AN OVERVIEW

NEMESIS is a set of compiler transformations which provide a soft error hardened code by adding redundancy and reforming control flow of the original code. NEMESIS partitions programmer-available machine registers into three sets, called M-regs (Master), D-regs (Detection) and R-regs (Recovery), and runs three independent sequences of instructions, named M-stream, D-stream, and R-stream. The M-stream has all instructions needed for functionally correct execution of a program. D-stream is a redundant copy of M-stream which does not include any memory write and functional call instructions, however, it does include all arithmetic, memory read, compare and branch instructions. R-stream just contains arithmetic and memory read instructions. This is because that R-stream results only will be used in majority-voting to mask the effect of the errors from the general purpose register file. In addition to these three redundant streams, a NEMESIS-protected program includes error detection, diagnosis and recovery instructions. NEMESIS assumes ECC-protected caches and memory, and its sphere-of-protection includes the entire microprocessor core components (excluding memory subsystem). The objective of NEMESIS is to detect and correct the effect of all transient faults and prevent a program from experiencing any form of functional and timing failures. NEMESIS salient features are:

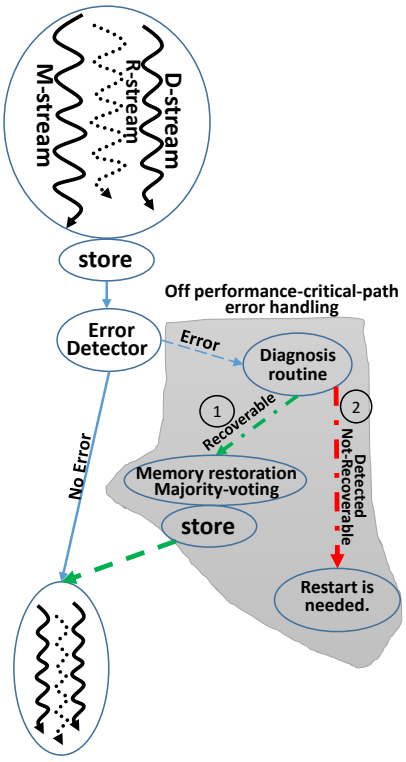

Figure 3: NEMESIS data-flow error handling. After each store instruction, Error Detector unit checks for errors, and if any observed, the diagnosis routine will get involved and classifies the error as either Recoverable or Detected/not-recoverable. If an error is recoverable, memory and register restoration will take place and program continues with executing the store instruction. Otherwise, the program stops the execution by raising an error flag.

i) NEMESIS sphere of protection includes all program instructions: NEMESIS checks for the er-

rors in the outcome of branch and memory write instructions rather than their register operands. This enhances the coverage of SWIFT-R based schemes from the triplicated instructions to the all program instructions including triplicated and critical instructions. If the checking fails, NEMESIS calls the corresponding diagnosis routine and attempts to recover from the error. Figure 3 shows the overview of NEMESIS error detection/handling on memory write instructions. For each store instruction, corresponding error detector verifies if the store operation has written the correct value into the the right memory location by loading back the stored data from the memory and check that against the D-stream computed store value. For branch instructions, direction check takes place by executing the corresponding D-stream compare and branch instruction, and verifying the destination basic block. Nevertheless, if the presence of any error is detected, a diagnosis routine will get invoked and determines the scope of error propagation. If the error is diagnosed as recoverable (marked as 1 in Figure 3), the effect of error from the register file and memory will be eliminated, then the recovery routine re-executes the corresponding critical instruction and the program execution resumes. Otherwise, if the diagnosis routine declares the presence of an unrecoverable error (marked as 2 in Figure 3), the program execution will be terminated.

ii) NEMESIS transformations closes all known software vulnerability windows: Software vulnerability window, defined as the duration between checking a value in software and the time to use the value, exist in almost all existing software-level techniques. These intervals can be a major source of failure especially in voting-based techniques. Instead of voting, NEMESIS checks for errors in the results of critical instructions. Since this checking will take place after the execution of store instructions, NEMESIS transformation preserves the value inside each memory location before update, for recovery purposes. This strategy completely eliminates the software vulnerability window.

4. NEMESIS: DETAILS

NEMESIS error coverage encompasses the execution of all program instructions because it verifies the results of critical instructions rather than blind voting between redundant versions of the register operands of such instructions. However, this makes error recovery challenging because once the error is detected the program has already committed the wrong memory write operation or the program execution control is in the wrong path. In this section we explain how NEMESIS detects and recovers from soft errors.

4.1 Error detection on Store Instructions

The core concept of NEMESIS error detection on memory write instructions is to load back the written value from the memory and check that against corresponding D-stream computed value. This idea has been employed in nZDC [7] and EDDI with store-readback [18], however, in the load-back error detection strategy there is a not-reported-before vulnerability window – called silent store vulnerable window. We identify/define it, show why it is tricky to eradicate and then present our solution to fix that.

Silent store vulnerable window: A store is said to be silent if it writes a value into a memory element which is already holding the same value [2]. If an error alters the effective address of a silent store, it can make a random modification to the state of memory and the error cannot be detected by load back strategy because the loaded value from the memory is as same as the stored value.

Figure 4 exemplifies an undetected error case in store-loadback strategy. The store is silent because the value in memory location addr, *val*, before executing `store` instruction (upper part of fig.4) is equal to the values which are computed by main and detection streams, val^M and val^D, respectively. Therefore, the state of memory should not get changed by the execution of `store` instruction. Now, assume that the soft error hits the base address register of the store, and alters the store's effective address from `addr` to `f-addr`. Consequently, the store writes its data into the faulty memory address `f-addr` rather than `addr`, and changes the state of memory while it is not supposed to do so(lower part of fig.4). This error remains undetectable, since the following checking-load instruction will load the value, *val*, from the correct address (computed by the detection stream), $addr^D$, which is equal to val^M and val^D. Note that simply inserting a check for the base address register `store` wouldn't solve the problem since the error can alter the store address without affecting the address register, i.e, errors affecting functional unit or pipeline register while processing the store instruction. Since silent stores can consist around 18% to 64% of total program's store instructions [2], fixing the silent store vulnerability is important in critical applications.

First-cut solution for silent store vulnerable window: Since silent stores do not alter the state of microprocessor, not executing such useless instructions will eliminate their vulnerability without harming the correctness of program. Thus, an obvious solution could be to jump over silent stores in the program. Figure 5-(a) illustrates the first cut attempt for eliminating the silent store problem from the store-loadback error detection strategy. In the figure redundantly computed values of M-/D- and R-streams are differentiated by a superscript M, D or R letter. For instance, val^M, val^D and val^R denote redundantly-computed store's value by M, D and R streams, respectively. Initially in this scheme, a silent-check load (inst. 1) reads back from exactly the same address that `store` is going to write into, and saves the loaded value in an specific register, called Silent Check Register (SCR). Then, SCR is compared against the M-stream computed store's value, val^M (inst. 2), to determine if the `store` is silent. If the condition is true, the program jumps over the `store` instruction, otherwise, `store` (inst. 3) will get executed and the following checking-load (inst. 4) instruction reloads the store's written value from the memory into

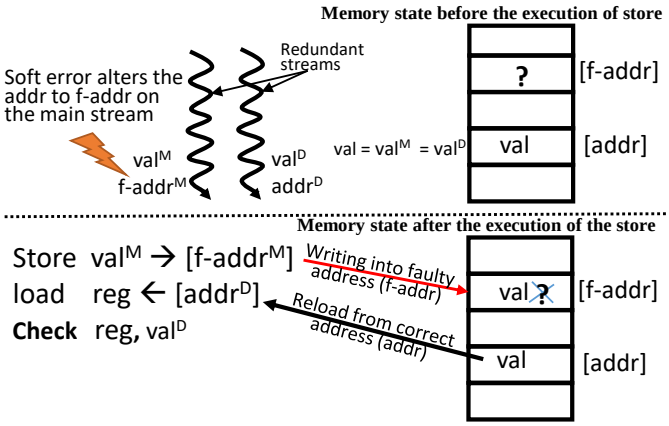

Figure 4: Silent store undetected error scenario in checking load mechanism. Since the store instruction is silent, errors which alter store address remains undetected.

SCR register. In the end, regardless of the store being silent or not, the SCR register should be checked against the value produced by D-stream, val^D (inst. 5), to make sure if store's value,val^M, was computed correctly.

NEMESIS solution for silent store vulnerable window: Although the first-cut method solves the problem of silent stores, it introduces yet another possibly undetected error scenario as the silent-check instructions (inst. 1, 2 in Figure 5-(a)) themselves are unprotected. Therefore, if any error alters the effective address of a silent-check load instruction in such a way that the wrongly loaded value is equal to the store value, a non-silent store will be treated like a silent one, and memory state would not get updated when it must. We named this type of errors as missing-memory update errors. These errors differ from silent store scenarios by a fact that the former does not change the state of memory while it should, and the later updates the state of memory while it should not.

NEMESIS error detection scheme eliminates the problem of silent-store vulnerability and missing-memory update by redundant and intertwined execution of silent-check operations. This is best explained with example shown in figure 5-(b). Firstly, the value within the `store` destination memory location is loaded back into two specific registers, VCR (Value Check Register) and SCR (Silent Check Register) by two redundant `load` instructions (marked as inst 1 and 2 in part (b) of figure 5) which use M- and D- stream's redundantly-computed registers as their address operands. Then in order to find out whether the `store` is silent, the SCR register gets compared against the `store` value register computed by M-stream, val^M. If the `store` is identified as silent, the VCR register gets compared against the `store` value register computed by D-stream (inst. 8) for error detection. Since, the results of redundant silent-check loads are compared with the two M- and D- stream computed store's values, the missing-memory update error will get detected. Now assume that the store is not silent, and `store` and `checking-load` (inst. (5) and (6)) will get executed. Instruction (5) performs the actual memory write operation and instruction (6) reloads the written value back to the VCR register. The

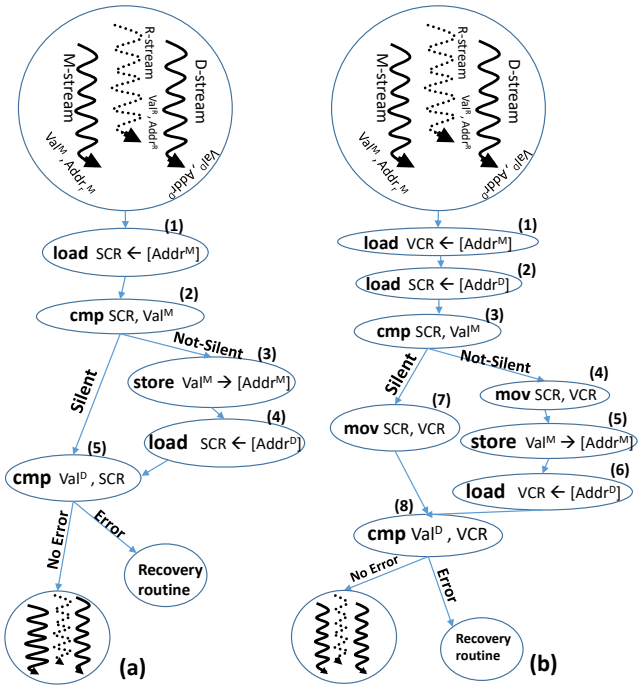

Figure 5: Memory write instruction checking mechanisms. (a) The first-cut solution which suffers from missing-memory update error and (b) NEMESIS memory write checking mechanism which solves the problem of silent-store and missing-memory update.

VCR register then gets compared against val^D, which is the redundant copy of `store` value computed by D-stream, for error detection purpose (inst. 8).

Since in NEMESIS transformation the effect of the error is detected after memory write instructions, it is necessary that a backup of about-to-be-written memory location preserved before each store instruction. Fortunately, the previous state of memory is already loaded to VCR register (inst. (1)). However, since this value will be overwritten by the checking load instruction (inst. (6)) in the case of not-silent stores, a copy of VCR is preserved in SCR register (inst. (4) and (7)).

4.2 Diagnosis/Recovery on Store Instructions

NEMESIS diagnosis routine is responsible to decide whether a detected error is recoverable. An on-store detected error is recoverable if the state of the register file and memory can be reverted to an error-free state before the execution of store instruction. Generally, the effect of error from registers can be masked by performing 2-of-3 majority-voting between corresponding M-, D- and R- registers. And, the memory state can be rolled-back to an error-free sate by writing back the backed-up data into the memory. However, there are two rare cases in which NEMESIS diagnosis routine declares a detected error as not recoverable:

Case 1: Inter-stream error propagation. If the effect of error has crossed the boundary of redundant streams, it is possible that all three redundant-computed registers contain different values, and, therefore, performing majority-voting cannot mask the effect of the error. For example, consider an error on the decode stage of the processor pipeline that alters the destination register pointer of an M-stream instruction to a D-

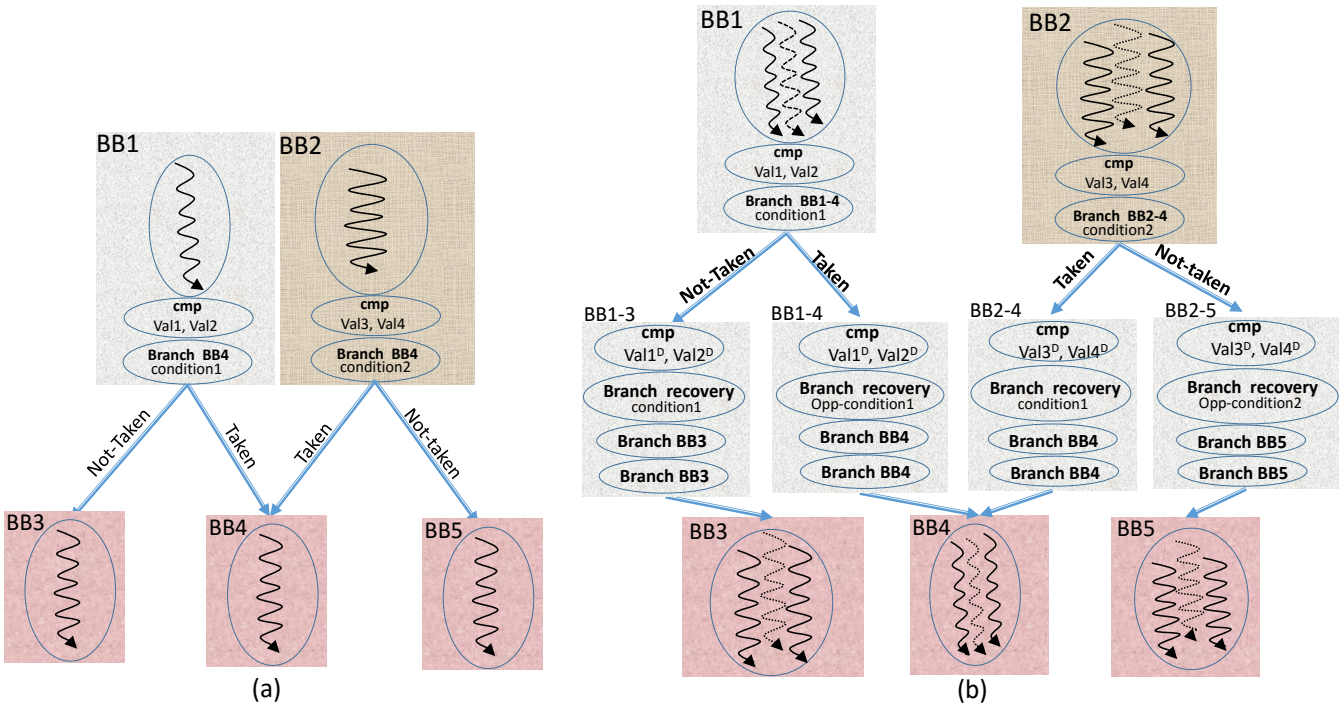

Figure 6: Control flow protection in NEMESIS. (a) unprotected program. (b) NEMESIS Control flow checking mechanism.

stream register which is going to be used as the operand for the corresponding redundant D-stream instruction. In this case, the three corresponding redundant registers will be holding different values, and, such an error can be detected but is unrecoverable.

Case 2: Unavailable memory backup. If the memory write instruction modifies a different memory location from the preserved one, the error is unrecoverable. Errors affecting store address register after the first silent-check load and before the store instruction (between inst. (1) and inst. (5) of part (b) of figure 5), or error altering the effective address of store instruction while it is processing in the processor are deemed to be detected but unrecoverable errors.

If none of the above cases are encountered, the diagnosis routine provokes the recovery block in which memory restoration takes place by writing the backup data into the faulty written memory address. Then majority-voting between registers eliminates the effect of error from register file and the normal program execution continues by retrying store instruction.

4.3 Control Flow Error Handling

Soft errors can alter the control flow (CF) of a program by producing unexpected jumps or wrong-direction branches in a program. An unexpected jumps arises when the program CF alters in a way which is not permitted in its control flow graph (CFG). Errors which directely modify PC register are examples of unexpected jumps. Such CF errors can be detected or even recovered with signature-based CF checking techniques [28, 29]. A wrong-direction CF occurs when a branch direction changes from taken to not-taken or vice-versa, which can be caused by errors affecting the computation of compare instruction register operands, execution and the opcode of compare and branch instructions, or even program status flag registers. The wrong-direction CF errors are more frequent than wrong-target errors and cannot be detected with most of the existing signature-based CF checking techniques [25]. Hence, the focus of NEMESIS transformation is to detect and recover from wrong direction control flow errors while a signature-based CF checking technique can be employed for handling unexpected jumps errors.

In order to detect wrong-direction CF errors, NEMESIS double checks the direction of a conditional branch by placing an intermediate block, called direction-check block, between each source and destination BBs (Basic Blocks) in the original program CFG. These direction-check blocks are necessary because they make direction double checking possible even for multiple-entry destination BBs. In each direction-check block, NEMESIS inserts a redundant (D-stream) compare instruction and announces the presence of error only if the redundant compare determines a different path from the original (M-stream) compare instruction. However, if no error is detected, the program control goes to the destination BB by a direct branch which is positioned at the end of direction-check block. NEMESIS CFC mechanism provides complete wrong-direction error detection and maximizes the masking effect of compare instruction.

Figure 6 demonstrates NEMESIS CF transformation for a simple program which has both, single-entry (`BB3` and `BB5`) and multiple-entry destination (`BB4`) basic blocks. In the figure, NEMESIS direction-check blocks are marked as `BB1-3`, `BB1-4`, `BB2-4`, and `BB2-5`. A direction-check block contains four instructions, a D-stream copy of the source BB compare instruction, a conditional branch

instruction to the diagnosis/recovery block, and two redundant direct branches to the destination BB. The condition of the conditional branch instruction in a direction-check block is specified in such way that it will not change the CF of the program if there is no fault. More specifically, if the control flow changes when the branch is not-taken (from `BB1` to `BB3` or from `BB2` to `BB5`), the condition of the redundant conditional branch instruction in the direction-check block is as same as the condition of branch in the source BB. On the other hand, if the CF changes from source to the destination BB when a branch is taken (from `BB1` or `BB2` to the `BB4`), the condition of the conditional branch instruction in the direction-check block will be opposite to the condition of the branch in the source BB. For instance, if the branch in source BB is "b.eq (branch equal)", the conditional branch in the direction-check block will be "b.ne (branch not equal)".

In a fault-free run of a program, the conditional of the conditional branch in direction-check block is always false, and the control flow goes to the destination BB with the first direct branch instruction. However, if soft error alters the direction of the source BB branch, the conditional branch in the direction-check block will change the control flow of the program to the corresponding recovery block. Note that, the second direct branch in the direction-check block (the last instruction) just gets execute if an error affects the execution of first directed branch in a way that it cannot change the control flow of the program.

Control Flow Error Diagnosis/Recovery: The control flow error diagnosis routine is simple because it just needs to check for inter-stream error propagation, and, if that is the case, the error is consider as detectable/not-recoverable. Otherwise, diagnosis routine transfers the control of the execution to the recovery block, where, majority-voting takes place between compare register operands, and the program resumes its error-free execution from the M-stream compare instruction.

4.4 Exceptions and Segmentations Faults

NEMESIS transformation detects/recovers from the manifestation of errors after memory write and branch instructions. However, if a soft error causes an hardware defined exception (i.e., divided-by-zero, illegal instruction opcode, accessing an inaccessible memory location) a fatal hardware trap will raise and program execution never reaches to the NEMESIS detection checks. To recover from such errors, NEMESIS error handling routines should be initiated at the beginning of the exception handling routines with some small modifications. For error detection, all architectural registers should get compared against their shadows. If there is any inconsistency between the state of the redundant registers, it can be presumed that soft error is the reason of the exception. Then NEMESIS performs majority-voting between redundant registers and re-executes the exception-raising instruction. If the exception raises again the normal exception handling mechanism responses to the occurrence of exception. .

Table 1: Simulator (Gem5 SE emulation mode) parameters.

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

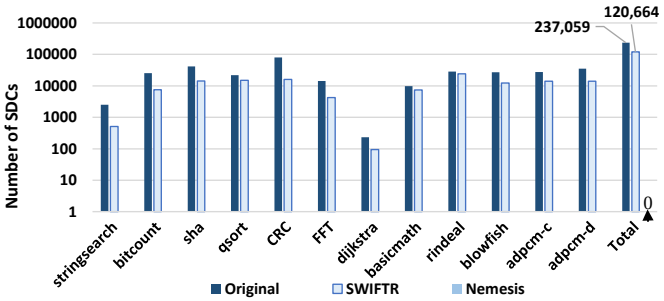

Figure 7: Out of 15 million fault injection experiments (evenly distributed between original, Swift-R and NEMESIS versions of programs), 237K result in SDCs in the ORG program, 120K in the Swift-R program, and 0 in the NEMESIS program.

5. EXPERIMENTAL METHODOLOGY

We implemented NEMESIS and Swift-R techniques as late back-end passes in LLVM 3.7 compiler infrastructure [17] for an ARMv8-a ISA. We compiled twelve benchmarks from MiBench benchmark suite [12] with *-O3* compiler optimization flag. For each program we produced three versions, Original, Swift-R and NEMESIS. We did not modify the standard library functions and we exclude them from all of fault injection and performance overhead evaluation results. We performed extensive fault injection experiments on a ARM cortex-A53 like simulated microprocessor. We used gem5[4] a cycle accurate simulator with configuration detail shown in table 1.

Fault model and fault sites: We inject single bit-flip per execution as our fault model. We injected faults on different bits of various hardware components including general purpose integer register file, pipeline decoder and instruction queue registers, integer functional units and load-store unit buffers.

In an attempt to cover all cases, we randomly inject 100,000 faults for each version of a program per fault site. For each version of a program, we injected 400,000 (4 * 100,000) faults in four hardware components. Overall, we inject 14,400,000 (400,000 * 3 * 12) fault injection experiments. These extensive fault injection experiments provide us more the 95% confidence interval with less than 0.1% error rate, which is 10x less error rate than previous works [7, 10, 22].

For each fault injection experiment, a target component and a (bit, cycle) are randomly selected. Once the simulator reaches the target fault injection cycle, simulation is paused and the selected bit is flipped, then, the simulation run resumes its execution till simulation terminates or the allowable simulation time gets over. The result of each simulation run is classified into one of the following category:

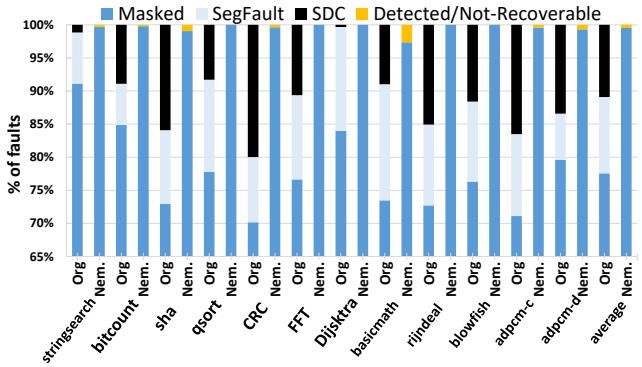

Figure 8: NEMESIS protected code increases the percentage of masked faults by 22.5%, and completely eliminates SDCs.

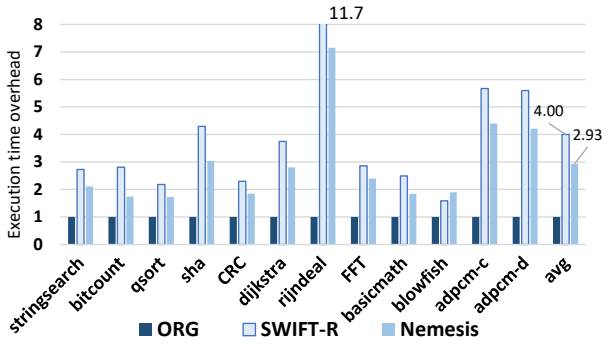

Figure 9: Nemesis-protected programs, on an average run 25% faster than Swift-R protected ones.

Masked: The final output of the program is correct. Note that faults, which are successfully recovered by Swift-R and NEMESIS techniques, also count as masked faults. **Failed:** Program terminates normally, but, the output is incorrect. This is the case of Silent Data Corruptions or SDCs. **SegFault:** Program terminates by generating some symptoms such as segmentation fault or simulation time is over. **Detected/Not-Recoverable:** In this case, program terminates by announcing the presence of a unrecoverable error. This type of outcome can only happen while the fault is injected during the execution of an NEMESIS-protected program.

6. EVALUATION AND ANALYSIS

6.1 NEMESIS transformation is effective

Figure 7 shows the number of fault injection runs that resulted in SDCs for unprotected (ORG), Swift-R and NEMESIS versions of the programs. The figure shows that of the 5 million runs, the original program results in SDC ∼237K times. The Swift-R version ends up with ∼120K times occurrence of SDCs. As compared to these, NEMESIS protected programs do not cause any SDCs. [1]. Note that the Y-axis of the graph, or `Number of SDCs` is in exponential scale and that the last bar in each graph shows the total number of experiments that resulted in SDC. We can see that fault injection in NEMESIS-protected programs never resulted in SDCs. This implies that NEMESIS error detection is able to detect all injected faults. The NEMESIS diagnosis routine always correctly distinguished recoverable errors from unrecoverable ones, and the recovery routine always eliminated the effect of error completely. NEMESIS error detection is able to detect all errors because it checks for the results of critical instructions rather than their operands, and it covers infrequent corner cases (described in section 4.1).

In comparison with original versions of programs, Swift-R protected programs, on average, produced 35%, 73%. 86% and 89% less failures for faults injected in load-store unit, register file, pipeline registers and functional units, respectively. Surprisingly, in some cases such as fault injection in the load-store unit for programs like basicmath and rijndeal, Swift-R transformation actually increases the number of failures! The reason is that in comparison with original codes, Swift-R transformation dramatically increases the number of memory operations for register-hungry programs and leaves them unprotected. NEMESIS transformation also increases the number of memory operations, but NEMESIS protects them either by triplication (read operations) or checking their results (write operations).

Figure 8 shows the fault injection experimental result distribution for original and NEMESIS-protected versions of the programs. It can be seen that very significant percentage of faults are masked, even in original programs (on average ∼77%), and NEMESIS recovery scheme increases the number of masked faults, on average, by about 22.5%. Almost half of these additional masked faults were generated exceptions and were covered by initiating NEMESIS recovery routine in the beginning of the exception handling routines. However, a very limited number of injected faults, on average ∼0.42% of total fault injection experiments (∼3% of the detected faults), lead to unrecoverable errors.

6.2 Performance overhead

Figure 9 shows the execution time overhead for Swift-R and NEMESIS versions of the programs. Compared to the original code, Swift-R and NEMESIS transformations, on average, increase the execution time of the program by about 4X and 2.9X, respectively.
The NEMESIS-protected programs run, on average, 25% faster than Swift-R protected ones, because they execute less number of instructions. For instance, for each load operation, NEMESIS transformation adds two extra redundant instructions while Swift-R transformation requires a majority-voting, which needs 4 machine instructions in our implementation, and two extra move instructions. The number of extra instructions added for memory write operation protection, for both NEMESIS and Swift-R transformations is 8 machine instruction. However, since NEMESIS transformation skips over silent stores instructions, the total number of dynamic memory write instructions in NEMESIS-protected programs is on average 18% less than Swift-R ones. For each compare instruction, Swift-R transformation requires two voting operations (so a total of 8 instruc-

[1]We show the absolute number of failures rather than fault coverage or the percentage of masked faults because as [24] reveals such metrics may cause protection overestimation

tions) while NEMESIS transformation requires just 3 extra instructions, as described in section 4.3. Note that the Swift-R and NEMESIS register reservation for redundant instructions, impose significantly high overhead in some register-hungry applications like rijndeal.

The performance overhead results reported for SWIFT-R transformation are higher than previous works. The reason is that we exclude the number of cycles that a program spends in standard library calls from our execution time estimation, because we didn't modify such functions. However, similar works like [10] assumes that library calls are protected but they do not consider the overhead of library calls protection which causes significant runtime overhead underestimation.

7. CONCLUSIONS

We present NEMESIS a software level redundancy-based technique which checks the results of memory write operations and the direction of branch instructions in a program. If any violation is detected, NEMESIS diagnosis and recovery schemes undo the effect of error from processor and memory state, and the program can continue its execution. Fault injection results show that NEMESIS-protected programs do not incur any SDC and run 25% faster than Swift-R protected programs.

References

[1] Guillaume Aupy et al. On the combination of silent error detection and checkpointing. In *PRDC*, 2013.

[2] Gordon B Bell et al. Characterization of silent stores. In *Parallel Architectures and Compilation Techniques*. IEEE.

[3] David Bernick et al. Nonstop® advanced architecture. In *DSN*. IEEE, 2005.

[4] Nathan Binkert et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 2011.

[5] Jonathan Chang et al. Automatic instruction-level software-only recovery. In *DSN*. IEEE, 2006.

[6] Marc A de Kruijf et al. Static analysis and compiler design for idempotent processing. ACM, 2012.

[7] Moslem Didehban and Shrivastava Aviral. nZDC: a compiler technique for near Zero Silent data Corruption. In *DAC*, 2016.

[8] Elmootazbellah N Elnozahy et al. Checkpointing for peta-scale systems: A look into the future of practical rollback-recovery. *TDSC*, 2004.

[9] Chad Farnsworth et al. A soft-error mitigated microprocessor with software controlled error reporting and recovery. *IEEE Transactions on Nuclear Science*, 2016.

[10] Shuguang Feng et al. Shoestring: probabilistic soft error reliability on the cheap. In *ACM SIGARCH Computer Architecture News*. ACM, 2010.

[11] Shuguang Feng et al. Encore: low-cost, fine-grained transient fault recovery. In *MICRO*. ACM, 2011.

[12] Matthew R Guthaus et al. Mibench: A free, commercially representative embedded benchmark suite. In *Workload Characterization*. IEEE.

[13] Siva Kumar Hari et al. Low-cost program-level detectors for reducing silent data corruptions. In *DSN*. IEEE, 2012.

[14] IRC. ITRS 2.0-Executive Summary. `http://www.itrs2.net/itrs-reports.html`, 2015.

[15] Dmitrii Kuvaiskii et al. Elzar: Triple modular redundancy using intel avx (practical experience report). In *DSN*, 2016.

[16] Ignacio Laguna et al. IPAS: Intelligent protection against silent output corruption in scientific applications. In *CGO*, 2016.

[17] Chris Lattner et al. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO 2004*.

[18] David Lin et al. Effective post-silicon validation of system-on-chips using quick error detection. *TCAD*, 2014.

[19] Qingrui Liu et al. CLOVER: Compiler directed lightweight soft error resilience. In *ACM SIGPLAN Notices*. ACM, 2015.

[20] Nahmsuk Oh et al. ED4I: Error Detection by Diverse Data and Duplicated Instructions. *IEEE Transactions on Computers*, 51, 2002.

[21] George A Reis et al. SWIFT: Software Implemented Fault Tolerance. In *CGO*. IEEE, 2005.

[22] George A Reis et al. Automatic instruction-level software-only recovery. *IEEE micro*, 2007.

[23] Felipe Restrepo-Calle et al. Selective SWIFT-R. *Journal of Electronic Testing*, 29, 2013.

[24] Horst Schirmeier et al. Avoiding pitfalls in fault-injection based comparison of program susceptibility to soft errors. In *DSN*. IEEE, 2015.

[25] Aviral Shrivastava et al. Quantitative analysis of control flow checking mechanisms for soft errors. In *DAC*. IEEE, 2014.

[26] Alex Shye et al. PLR: A software approach to transient fault tolerance for multicore architectures. *TDSC*, 2009.

[27] Boeing Stuart and Russell Urzi. A research agenda for mixed-criticality systems.

[28] Ramtilak Vemu and Jacob Abraham. CEDA: Control-Flow Error Detection using Assertions. *IEEE Transactions on Computers*, 60(9):1233–1245, 2011.

[29] Ramtilak Vemu et al. ACCE: Automatic Correction of Control-flow Errors. In *International Test Conference*. IEEE, 2007.

[30] Cheng Wang et al. Compiler-managed software-based redundant multi-threading for transient fault detection. In *CGO*. IEEE, 2007.

[31] Yun Zhang et al. DAFT: Decoupled Acyclic Fault Tolerance. *International Journal of Parallel Programming*, 40, 2012.