A software-level Redundant MultiThreading for Soft/Hard Error Detection and Recovery

Hwisoo So*, Moslem Didehban†, Aviral Shrivastava†, Kyoungwoo Lee*
*Department of Computer Science, Yonsei University, Seoul, Korea
Email: {Shs7719,Kyoungwoo.Lee}@yonsei.ac.kr
†Compiler Microarchitecture Lab, Arizona State University, Tempe, AZ
Email: {Moslem.Didehban, Aviral.Shrivastava}@asu.edu

***Abstract*—In this work, we investigate the potential of software-only RMT (Redundant MultiThreading) schemes for soft and hard error detection and recovery. We first implement and evaluate the error protection capability of basic software level triple redundant multithreading (STRMT) and analyze its vulnerability. Then we introduce FISHER (FlexIble Soft and Hard Error Resiliency) as a software RMT scheme which can achieve high degree of error resiliency and does not suffer from STRMT vulnerability holes. FISHER executes three threads and rather than having a centralized voting mechanism, it distributes and intertwines error detection and recovery operations between redundant threads. We performed 135,000 soft/hard error injection experiments on different hardware components of an ARM cortex53-like μ-architecturally simulated microprocessor. The results demonstrate that FISHER can reduce programs failure rate by around 42$\times$ and 26$\times$ compared to original and basic STRMT-protected versions of programs, respectively.**

I. Introduction

Advances in semiconductor technology have made computer-based systems as an indivisible part of virtually all aspects of human life. While some of these systems are inherently error tolerant, some cannot tolerate any erroneous behavior. Hardware malfunctions have been considered as one of the main reasons behind unexpected program behaviors and system failures. Hardware faults are classified into two categories: a) *transient faults or soft errors* i.e., temporarily bit flip errors – caused by high-energy particles, voltage violations, and other electromagnetic interference, and b) *permanent faults or hard errors* caused by process variation, thermal stress, or oxide wear-out. It has been predicted that overall system failure rate due to hardware errors grows continuously mainly due to the ever-increasing level of integration in different layers of computer-based systems, i.e., more transistors per core, more cores per chip and more chips per system [1], [2].

Software level error resilience schemes are promising because they can boost the reliability of execution even for commercial-off-the-shelf microprocessors. Redundancy is the key strategy behind error resilience schemes. While software error resilience transformations can apply redundancy in various abstraction levels (e.g. instruction-level [3] or thread-level [4], [5]), in this work we concentrate on thread-level or Redundant MultiThreading (RMT) schemes. The main idea of thread-level error mitigation schemes is to statically create redundant copies of main execution thread and achieve error resilience by dynamically checking the results of register operands of memory operations. Since redundant threads can potentially be executed on physically different hardware cores for a multicore microprocessor, they can detect the manifestation of both soft and hard errors. Many of the existing of software-based RMT error resilience schemes assume that the microprocessor memory subsystem (TLBs, caches and main memory) is ECC-protected and only focus on *soft error detection* on microprocessor core components.

We also presume that memory subsystem is protected and we investigate the potential of **S**oftware-level **T**riple RMT (STRMT) for *soft/hard error detection and recovery*. We first implement and analyze the error coverage capability of a basic STRMT scheme – a straightforward extension of state-of-the-art software-level thread duplication and error detection solutions. The basic STRMT runs three redundant threads on different cores of a multi-core microprocessor and performs majority voting operations between store register operands immediately before the execution of each store. The results of microprocessor-wide single bit soft/hard error injection experiments demonstrate that significant amount of errors ($\sim$12%) cause failure (wrong output) in basic STRMT protected programs!

Furthermore, we propose FISHER as a software-level thread triplication error resilience scheme which eliminates basic STRMT single-point-of-failures by distributing and intertwining error detection, diagnosis and recovery operations between main and redundant threads. To detect the manifestation of permanent errors, FISHER double-checks for errors in the computations after each error detection and recovery. To eliminate vulnerabilities introduced by the single-instance memory write operations, both redundant threads independently verify the results (not operands) of main thread memory write operations. As a result, FISHER's sphere of protection encompasses all program instructions (i.e., computational, memory and control flow operations) and can handle both soft and hard error effectively. Micro-architectural level error injection experiments show that FISHER improves the reliability of basic STRMT by $\sim$26$\times$!

II. Basic STRMT

Basic STRMT is a straightforward implementation of software-level thread triplication and voting for error detection and recovery. In basic STRMT, we simply follow prevalent rules of state-of-the-art thread detection schemes [4]–[6], but we apply triplication rather than duplication and perform voting rather than error checking. Main idea here is to mask the manifestation of single error by performing majority voting between redundant versions of store register operands immediately before execution of memory write operations.

As Figure 1 depicts, basic STRMT runs two extra redundant threads (marked as Redundant Thread 1 and Redundant Thread 2 in Figure 1), for each main execution thread. These three redundant threads run on physically different cores and execute

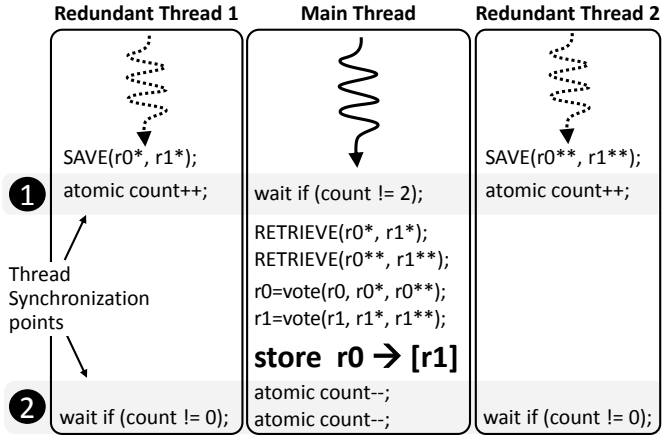

Fig. 1: Basic STRMT error resilience strategy.

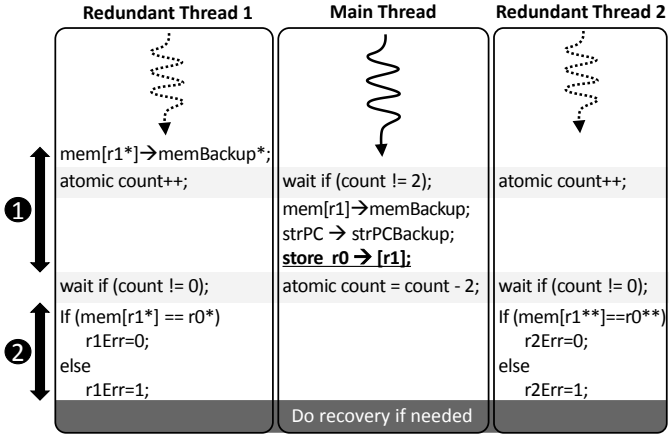

Fig. 2: FISHER error detection process.

mostly identical computations, but only the main thread performs program memory write instructions. Redundant threads send out the results of their computations, i.e., the values of store data and address register operands (`r0*` and `r1*` for redundant thread 1 and `r0**` and `r1**` for redundant thread 2) to the main thread through a shared memory buffer write operations (`SAVE`). Main thread retrieves the results of redundant threads computations from the shared buffer (`RETRIEVE`), and performs 2-of-3 majority-voting between results and then executes the store instruction[1]. Basic STMRT requires two synchronization points for correct execution: one immediately before main thread collects redundant threads results (marked as ①), and one immediately after the execution of store (marked as ②). The first synchronization barrier is required to guarantee that the RETRIEVE operations in main thread take place after that redundant threads wrote their values into the shared buffer. The second synchronization is required to provide consistent memory view for redundant threads. If we eliminate the second synchronization point, chances are that redundant threads perform a conflicting memory operation (i.e., load from the same memory location as store target address) before the main thread updates the memory. In such cases, redundant threads will read the wrong (not updated) memory value and the program may produce wrong output.

To evaluate the effectiveness of basic STRMT, we conduct statistical fault injection experiments (detailed explained in section IV) on different benchmark programs protected by basic STRMT. The results reveal that a significant number of (around 12%) of injected errors lead to program failure (wrong output or silent data corruption)! We analyze the results and realize that while basic STRMT can correct (mask) the manifestation of data flow errors which alter computations (visualized as down-flow arrows in Figure 1), it suffers from four main vulnerability windows:

(i) Main thread write operations. If an error affects the execution of a main thread store instruction, it remains undetected because checking operations are placed before store.

(ii) Redundant threads SAVE operations. To send the results of their computations to the main thread, redundant thread shall write their results to a shared memory buffer. This happens through write (store) instructions that are annotated as `SAVE` operations in Figure 1. Errors affecting effective address of these memory write operations are also vulnerable because they will lead to writing data to a wrong memory location.

(iii) Main core permanent faults. If a permanent error occurs on the core executing main thread instructions, the error can propagate to the memory. For instance, consider a permanent stuck-at error on an arbitrary bit of physical register holding store data value (`r0` in the Figure). Despite of majority voting operations the written value into the memory would be wrong, simply because store instruction gets its data from register file which is permanently erroneous.

(iv) Main thread control-flow errors. Errors which alter the control flow of the main thread execution in such a way that it skips over the execution of at least one store instruction remain unrecoverable. Similarly, control flow errors jumping over some computations instructions and at least one of the majority-voting operation before the execution of store also remain unrecoverable. Examples are faults in the program counter register (PC, nPC) and the target address of conditional/direct branches.

III. Our Solution: FISHER

To solve the above-mentioned vulnerability intervals, in this section we introduce FISHER as an effective software-only soft and hard error detection and recovery scheme. Similar to the basic STRMT, FISHER adopts forward-recovery strategy to protect the execution of programs against single bit soft and hard errors.

Post-store error checking: FISHER adopts post-store error detection (ED) strategy utilized in the state-of-the-art error detection schemes [3], [7]–[9]. The core idea is that first main thread performs store instructions and then asks redundant threads to check the results. More specifically, after each memory write operation committed by the main thread, both redundant threads load the already written data from the memory and check it against their own locally computed version of store value register. Figure 2 illustrates an example of FISHER transformation for a typical store instruction.

Recovery information preservation: FISHER preserves the value of about-to-write memory location (`mem[r1]`) and store id (`strPC`) into two specific registers prior to the store execution. These information will be used for unrecoverable error diagnosis and memory restoration in the case of error. Store id can be any static value which is uniquely assigned to each store instruction at compile time. We use the corresponding store program counter as store ID. Note that FISHER keeps

[1]SAVE and RETRIEVE operations are indeed store and load instruction. We opt to use different terms to distinguish between program related load and stores and the memory operations required for error detection and corrections.

TABLE I: FISHER error detection and recovery process.

Error Info		Faulty	Action		
r1Err	r2Err	Core	Redundant1	Main	Redundant2
0	0	None	Continue	Continue	Continue
0	1	Core 2	① Preserve AS[a] and continue	Continue	② Restore AS and continue
1	0	Core 1	② Restore AS and continue	Continue	① Preserve AS and continue
1	1	Main Core	① Preserve AS ④ Permanent Check and continue	② Restore Memory and AS ③ Retry Store	④ Continue

[a] AS: Architectural State

recovery information in two specific registers (rather saving them to memory) to avoid executing hard-to-protect memory write operation.

Collaborative error detection and recovery planning: Redundant threads load the data from the store target address and check it against their own locally computed version of store value. If any thread detects a discrepancy, it sets its designated shared error detection flag (`r1Err` and `r2Err` variables for first and second redundant threads, respectively). Once both redundant threads make their decisions about the error, the next execution phase which can be either continuing normal execution or starting recovery process will be determined based on the value of the shared error flags.

Table I shows ED and recovery processes in different situations based on the values of error detection flags. As the first row shows, if both `r1Err` and `r2Err` are set to zero, there is no error in computations and all threads can continue their execution. If only one of the error detection flags (either `r1Err` or `r2Err`) is set one, that implies only one of the redundant threads (the one that its corresponding error detection flag is set to one) has experienced some sort of error. In such situations (second and third rows of Table I), first, the error-free redundant thread preserves its architectural registers in a designated shared memory space and then continues its execution. Then, the faulty thread loads recently-saved architectural state from designated shared memory and copies them into their corresponding registers and continues its execution. Note that in these cases, the main thread can safely continue its execution because the scope of error is limited to the faulty core.

If both error detection flags are set to one (last row of the Table I), main thread/core is erroneous. In this case, the recovery process is more evolved because potentially both memory and register state restorations are required which may not be even possible in all scenarios. The actions required for safe recovery from the errors affecting main thread are as follow: (i) One of the redundant threads (redundant thread 1 in our implementation) saves its architectural state (AS) in the designated shared memory space and informs main thread that AS preserving is done. (ii) Main thread checks if the error is recoverable (next bullet explains this process). If yes, main thread first restores the memory state to the right before the execution of faulty store and then reverts the impact of error from its register file. To restore the memory state, all needed is to write back the memory preserved data (which was preserved right before the execution of store) to the memory. To restore register state, the main thread loads the saved architectural state from shared memory and copies them into their corresponding registers. On the other hand, if error annotated as unrecoverable, then main thread notifies the user and terminates the program execution. (iii) At this point, the impact of error from the execution is reverted and main thread execution can resume from right before the store instruction.

Safe stop in presence of unrecoverable errors: Soft errors are unrecoverable if they permute memory address of the store operation in such way that the store updates a wrong memory location different from the preserved (backed up) ones. To deal with such cases, FISHER diagnosis routine checks for validity of the memory backup. It uses previously redundantly preserved memory backups (denoted as memBackup and memBackup* in Figure 2), last store ID (denoted as storePCBackup in Figure 2), the current state of memory (mem[r1]) and the value of store operand registers of main and redundant threads. Other required information of redundant thread is also accessible to the main thread because one of the redundant threads has already saved its AS which includes the value of its registers as well as its memory backup (memBackup*) to the shared checkpointing area.

Results double-checking: One of the redundant threads re-checks the execution of memory write instruction (after recovery) simply by re-loading the written data from the memory and checking it against its own locally computed value. If any discrepancy is observed in this step, the redundant thread raises the detected-unrecoverable error flag and terminates the program execution. This double-checking process after each recovery takes care of main thread unrecoverable permanent errors (third vulnerability hole in basic-STRMT technique).

Thread control-flow errors: Any control-flow error which causes a thread to violate FISHER synchronization will result in deadlock error. If an error changes any thread control to jump to a different block without violating the synchronization, it will be detected by load-back checking, as corrupted one holds different store information from other clean threads. Then FISHER recovery scheme calculates proper control-flow information (in other words, pc vlaue) of corrupted thread from the error-free thread to return the corrupted one to the correct block.

IV. Experimental results

We used LLVM 3.7 compiler [10] to implement basic-STRMT and FISHER transformations. We compiled nine benchmarks from MiBench suite [11] with -O3 compiler optimization flag and generated three binary versions (ORG, Basic STRMT, FISHER) for each program. We implemented basic-STRMT and FISHER as compiler back-end passes for ARMv7-A 32 bit architecture. We used gem5 [12] μ-architectural level simulator and modeled a two-issue in-order dual-core microprocessor in SE mode.

A. Fault Injection Setup

We injected single bit-flip transient and single stuck-at 0/1 permanent faults on sequential elements of different hardware components of the simulated processor while running original, basic-STRMT and FISHER versions of programs. Targeted hardware components are register file, fetch and decode stage pipeline registers, functional units as well as load/store unit. For each component, we inject 500 transient faults and 500 permanent stuck-at faults per each version which makes 5,000

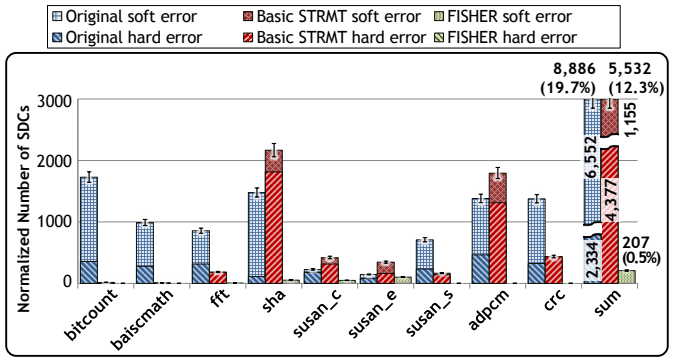

Fig. 3: Normalized soft and hard error failure rate for original, basic STRMT and FISHER programs. Compared to basic STRMT, FISHER reduces the number of normalized SDCs by ∼26×.

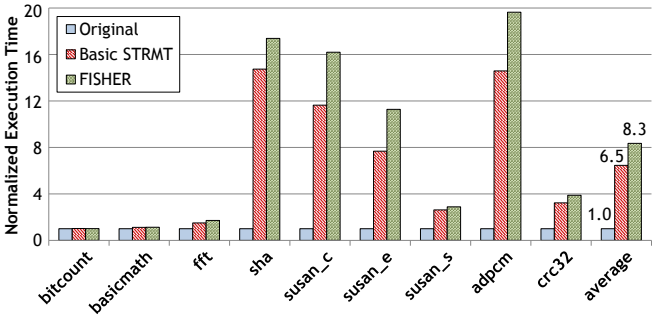

Fig. 4: Performance overhead of basic STRMT and FISHER normalized to the runtime of the original version of program.

(5*500 transient + 5*500 permanent faults) random processor-wide fault injection experiments per each version of programs. Overall, we performed around 135,000 (9 * 5000 * 3) fault injection experiments which provide us with less than 5% error with 95% confidence interval for each component.

Normalized SDC as comparison metric: Similar to [13], we consider the impact of execution time and hardware overhead of protected schemes on reliability estimation by multiplying the absolute number of SDCs by a correction factor which is proportional to the performance and hardware overheads of the protected schemes. For original versions of the programs Normalized SDC is equal to the number of SDCs. For basic STRMT and FISHER, the hardware overhead is 3 since these transformations require two extra cores for the execution of redundant threads and their execution overhead is also extracted from their corresponding performance overhead.

B. Fault Coverage

Figure 3 depicts the results of permanent and transient fault injection experiments for unprotected (ORG), basic STRMT and FISHER versions of the programs. In the Figure, Y-axis represents the normalized number of SDCs and X-axis shows the benchmarks. Note that in the figure the rightmost set of bars (annotated by sum) represents the sum of all normalized SDCs for different versions of programs. Results indicate that out of the 45,000 fault injection experiment (22,500 transient and 22,500 permanent faults) on the original version of programs, around 20% of them result in SDC. Applying basic-STRMT error resilience scheme can reduce the normalized number of SDC down to around 12%. As compared to these, the normalized SDC for FISHER protected programs is only ∼0.5% which is about 26× resilience improvement compared to basic STRMT. In fact, only one 6 transient errors have led to failure in FISHER-protected programs, which after normalization grows to 207 (∼0.5%).

C. Performance Overhead

Figure 4 shows execution time overhead for basic STRMT and FISHER transformations. On an average, basic STRMT and FISHER transformations increase the programs execution time by on around ∼6.5× and ∼8.3×, respectively. The performance overhead numbers of FISHER are inline with previous works. For instance, [4] and [9] report ∼4× and ∼5× execution time overhead just for error detection. Runtime overhead of FISHER heavily depends on program characteristics. For example, the average performance overhead for bitcount, basicmath, crc and susan(s) programs which in them memory write operations comprise less than 1% of total executed instructions, is only around 2.2× for FISHER transformation. On the other hand, for programs with a high number of store operations (e.g., adpcm(c), sha, susan(c) and susan(e)) the average runtime overhead is 16× for the FISHER-protected version of the programs.

V. Conclusions

We introduced FISHER, a compiler level thread triplication and forward error detection and recovery scheme. FISHER distributes error detection checks to enable faulty core identification and safe error recovery.

VI. Acknowledgements

This work was partially supported by funding from NS-FCCF 1055094 (CAREER); by funding from NRF-2016H1A2A1909470 (Global PH.D. Fellowship Program, NRF, the Ministry of Education); by funding from NRF-2015M3C4A7065522 (Next-generation Information Computing Development Program, NRF, MSIT); by funding from 2014-3-00035 (High Performance and Scalable Manycore Operating System, IITP, MSIT).

References

[1] S. Borkar, "Designing reliable systems from unreliable components: the challenges of transistor variability and degradation," *IEEE Micro*, vol. 25, no. 6, 2005.
[2] J. Henkel *et al.*, "Reliable on-chip systems in the nano-era: Lessons learnt and future trends," in *DAC*, 2013.
[3] M. Didehban *et al.*, "InCheck: An in-application recovery scheme for soft errors," in *DAC*, 2017.
[4] C. Wang *et al.*, "Compiler-managed software-based redundant multi-threading for transient fault detection," in *CGO*, 2007.
[5] Y. Zhang *et al.*, "DAFT: decoupled acyclic fault tolerance," *IJPP*, vol. 40, no. 1, 2012.
[6] K. Mitropoulou *et al.*, "COMET: communication-optimised multi-threaded error-detection technique," in *CASES*, 2016.
[7] M. Didehban *et al.*, "nZDC: A compiler technique for near zero silent data corruption," in *DAC*, 2016.
[8] M. Didehban and A. Shrivastava, "A compiler technique for processor-wide protection from soft errors in multithreaded environments," *IEEE Transactions on Reliability*, vol. 67, no. 1, pp. 249–263, 2018.
[9] H. So *et al.*, "Expert: Effective and flexible error protection by redundant multithreading," in *DATE*, 2018.
[10] C. Lattner *et al.*, "LLVM: A compilation framework for lifelong program analysis & transformation," in *CGO*, 2004.
[11] M. R. Guthaus *et al.*, "Mibench: A free, commercially representative embedded benchmark suite," in *WWC*, 2001.
[12] N. Binkert *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, 2011.
[13] H. Schirmeier *et al.*, "Avoiding pitfalls in fault-injection based comparison of program susceptibility to soft errors," in *DSN*, 2015.