CHITIN: A Comprehensive In-thread Instruction Replication Technique Against Transient Faults

Hwisoo So*, Moslem Didehban†, Jinhyo Jung*, Aviral Shrivastava‡, Kyoungwoo Lee*
*Dependable Computing Lab, Yonsei University, Seoul, South Korea. {Shs7719,Jinhyo.Jung,Kyoungwoo.Lee}@yonsei.ac.kr
†Cadence Design Systems, San Jose, California. Moslem@cadence.com
‡Compiler Microarchitecture Lab, Arizona State University, Tempe, AZ. Aviral.Shrivastava@asu.edu

***Abstract*—Soft errors have become one of the most important design concerns due to drastic technology scaling. Software-based error detection techniques are attractive, due to their flexibility and hardware independence. However, our in-depth analysis reveals that the state-of-the-art techniques in the area cannot provide comprehensive fault coverage: i) their control-flow protection schemes provide incomplete redundancy of original instructions, ii) they do not protect function calls and returns, and iii) their instruction scheduling leaves many vulnerabilities open. In this paper, we propose CHITIN – code transformations for soft error resilience that adopts the load-back checking scheme of nZDC, an improved version of SWIFT-like control-flow protection scheme, and a contiguous scheduling of the original and redundant instructions to dramatically improve the vulnerability from soft errors that disrupt the control-flow. Our fault injection experiments demonstrate that CHITIN can reduce more than 89% of the silent data corruptions in the state-of-the-art solutions.**

I. Introduction

Transient faults or soft errors, unexpected temporary bit flips in the transistor caused by energetic particles, electromagnetic interference, electrical noises, etc., have been identified as one of the main sources of hardware malfunctions in modern microprocessors [1]–[3]. Traditionally, soft errors were a concern only in high-altitude settings such as satellites and airplanes. However, due to continued device scaling and higher integration, soft errors are now a serious reliability concern even in terrestrial settings [4], [5], and are considered as a first-class suspect in many system-level failure scenarios [6]–[9].

Hardware-level solutions, such as ARM Cortex-R dual/triple core lock-step microprocessors [10], [11], have been frequently used to mitigate the problem of soft errors. However, they require modifications in the design, and are inapplicable to existing off-the-shelf processors. Software-level solutions are more desirable as they can be applied on any past, present or future processor, and can be applied opportunistically only when required [12], [13]. Some of the most popular software-level protection techniques can be classified as in-thread instruction replication techniques [12]–[20]. These techniques replicate the assembly-level instructions and execute the original and redundant (shadow) execution streams inside the same thread, so that an error would make a difference between the architectural state of the two streams. Intermittent checking among the original and shadow streams is performed to detect errors.

However, our in-depth analysis reveals that the state-of-the-art in-thread replication techniques suffer from several limitations. SWIFT (SoftWare Implemented Fault Tolerance) [12], the most cited technique, has flaws in its checking algorithm for stores and branches. nZDC (near Zero Data Corruption) [13], another popular technique, resolves these problems but still fails to provide complete redundancy for control-flow instructions. Furthermore, both do not prescribe any transformation for function calls and returns, and their instruction scheduling strategy leaves them open to several more vulnerabilities. These limitations make the techniques more vulnerable to control-flow errors, which are less likely to be masked compared to data-flow errors [21]. Consequently, such drawbacks make existing schemes ineffective for mission-critical applications running on embedded microprocessors.

To address the above mentioned drawbacks, we propose CHITIN[1] – CompreHensive In-Thread INstruction replication. The contributions of CHITIN are:

i) Complete redundancy for control-flow instructions: Previous techniques for control-flow instructions either let errors in the original branch instruction propagate to corrupt the checking instruction, or fail to detect the faults that illegally transfers the control-flow to another block. CHITIN protects the control-flow instructions by completely separating the branch and its corresponding error checking instructions to different (original or shadow) instruction streams.

ii) Protection for function calls and returns: To the best of our knowledge, CHITIN is the first in-thread replication scheme to provide a comprehensive transformation ruleset for function call and return instructions. Utilizing a signature-based method with a redundant link register, CHITIN guarantees the correct execution of these instructions.

iii) Contiguous instruction scheduling: The instruction scheduling strategies of previous works generate frequent equal-point-of-executions, points in which the architectural states of the original and shadow streams are identical. Naturally, errors causing jumps between two equal-point-of-executions in the same basic block cannot be detected. CHITIN uses a contiguous original-shadow instruction scheduling strategy, which minimizes the number of equal-point-of-executions in a basic block, and therefore reduces the number of undetected control-flow errors.

Our statistical transient fault injection experiments on a Verilog description of OpenRISC microprocessor show that CHITIN can reduce the number of undetected failures of SWIFT and nZDC by around 14.8x and 9.0x, respectively.

[1]pronounced kītn, is the main constituent of exoskeletons.

978-3-9819263-5-4/DATE21/©2021 EDAA

TABLE I
INSTRUCTION CATEGORIES AND THE FOLLOWING TRANSFORMATION RULES OF SWIFT, NZDC, AND CHITIN

Instruction Category		In-Thread Replication Techniques		
		SWIFT [12]	nZDC [13]	CHITIN [THIS WORK]
Arithmetic & Logical Operations		Operation replication with shadow registers assigned for redundancies of register operands		
Memory Operations	Load	Operation replication with shadow registers assigned for redundancies of register operands		
	Store	Pre-store checking *(Vulnerable against post-check pre-store error)*	Post-store load-back checking	
Control Operations	Jump	Signature-based checking		
	Conditional Branch	Signature-based checking *(Vulnerable against wrong-direction error)*	Signature-based checking *(Vulnerable against inter-block control-flow error)*	Signature-based checking completely independent of the branch
	Function Call and Return	*(Not implemented)*		Signature-based checking utilizing the redundant link register

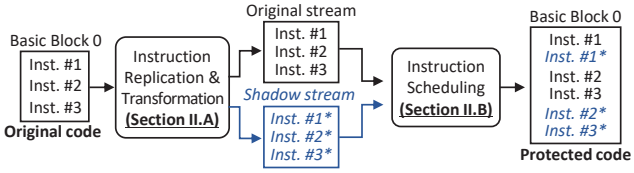


Fig. 1. In-thread replication replicates an execution stream and schedules the original and shadow streams in the same thread.

II. RELATED WORKS

In-thread instruction replication schemes consist of replicating instructions in the original stream based on their functionality, and scheduling the original and replicated execution streams into one thread, as illustrated in Figure 1. In-thread replication schemes aim to detect errors by spotting the discrepancies the errors cause between the two streams. The most cited in-thread replication scheme is SWIFT [12], which replicates the instructions and the corresponding registers into two execution streams without replicating the data memory. Recent in-thread replication schemes [13]–[20] are largely based on SWIFT, and most of them focus on lowering the performance penalty of SWIFT. nZDC [13], on the other hand, pointed out some vulnerabilities in SWIFT and attempted to resolve them.

A. Instruction transformation

To protect the system against soft errors, in-thread replication transforms the replicated instructions to form a redundant execution stream that accurately repeats the functionalities of the original stream. Table I summarizes the classification of instructions and the corresponding transformations of representative in-thread replication techniques, SWIFT and nZDC.

1) Arithmetic and logical instructions: Arithmetic and logical instructions update the destination register with their results. In-thread replication creates replicas of such instructions, replacing the register operands with shadow registers so that the replicated instructions provide redundant results. For this purpose, in-thread replication partitions the general purpose registers into original and shadow registers. The original instructions and registers form the original execution stream, and the replicated instructions and shadow registers form the shadow execution stream, which should be identical to the original stream if the system executes both streams correctly.

2) Memory instructions: The memory subsystem is usually protected by hardware solutions such as parity code or error correction code, and therefore popular in-thread techniques do not replicate the data memory space. They rather focus on preventing errors in corrupted store instructions from propagating to the memory subsystem.

As load instructions only update the data registers, they can be replicated just as arithmetic and logical instructions as described in Table I. However, replicating store instructions is more complicated, since its result should be stored in the memory subsystem. Therefore, SWIFT-based schemes [12], [14]–[20] compare the original and shadow data and address registers of the store to detect potential errors. The store instruction is executed in the original stream after the checking ensures its correctness.

nZDC [13] pointed out a vulnerability in these schemes; soft errors that occur after the checking but before or during the execution of store could corrupt memory without being noticed. Accordingly, nZDC proposed load-back checking as an alternative to pre-store checking. Load-back checking executes the store with original registers, and then loads the stored value back with the shadow address register to check the correctness of the executed store.

3) Control-flow instructions: Since control-flow instructions such as jumps and branches update the program counter (PC) immediately, it is hard to directly replicate such instructions. In-thread replication schemes rather adopt CFCSS (Control-Flow Checking by Software Signatures) [22], reserving some general purpose registers to function as a redundant PC. A common implementation constitutes of assigning signatures to each basic block, using one register to hold the current signature, and another register to hold the signature difference of the current and next block [12]. Upon entering a block, the signature register is updated with the difference, and compared with the signature of the block to check the validity of the control-flow.

However, the state-of-the-art in-thread replication techniques make mistakes in implementing their versions of CFCSS, exposing them to vulnerabilities. SWIFT makes a signature update instruction depend on the branch that it should protect. This allows a single error to corrupt both the direction of a branch and its checking signature, making SWIFT vulnerable to wrong-direction control-flow errors. nZDC uses a constant uniformly as the correct signature for every block, and in turn misses some errors that jump from one block to another block.

4) Function calls and returns: Function calls and returns directly update the PC. Since both the original and shadow streams share the PC, soft errors on these instructions can corrupt the two streams simultaneously. Although previous in-thread replication schemes protect function parameters by replicating them with shadow registers, the schemes provide no information on how to replicate the function calls and returns. A recent scheme [21] suggests a signature-based protection for

these instructions. However, it does not protect the link register, which is essential for the correct execution.

B. Instruction scheduling

The replicated and original instructions should be carefully scheduled in the same thread to avoid malfunctions. For example, the instructions should be placed so that they do not break any dependencies or change the semantics of the program. Previous schemes adopt the alternating instruction scheduling strategy. This strategy schedules one shadow instruction right before or after the corresponding original one, and the system alternates between executing original and shadow instructions.

Poor scheduling can also cause the system to be vulnerable to unwanted jump errors, in which the control-flow unintentionally jumps from one execution point to another. With the example of alternating scheduling, every point after the execution of a pair of original and shadow instruction is an equal-point-of-execution, meaning that there is no difference in the two execution streams at the point. Unwanted jump errors between two such points in the same block do not cause any difference between the two execution streams, and hence cannot be detected. Alternating scheduling is especially vulnerable since it induces frequent equal-point-of-executions [23].

III. Our Solution: CHITIN

We introduce CHITIN, which overcomes the limitations of previous in-thread replication schemes and presents a comprehensive set of rules for instruction replication, transformation, and scheduling. CHITIN adopts the instruction transformation of nZDC for arithmetic, logical, and memory instructions as it provides sufficient fault coverage. For control-flow instructions, CHITIN builds on the transformation of SWIFT, which is efficient but incomplete. CHITIN also provides transformation for function calls and returns and a contiguous scheduling strategy to extend its error coverage. In each subsection, we describe the problems that exist in previous techniques, and explain how CHITIN resolves the vulnerabilities. The rightmost column in Table I summarizes the instruction transformation of CHITIN as compared to previous techniques.

A. Transformation for compare and branch instructions

To protect a branch instruction, SWIFT creates a replicate compare instruction with shadow registers, and updates the signature difference register based on the result of the shadow compare. However, if the branch is not taken, SWIFT updates the signature difference register to indicate not taken, regardless of the result of the shadow compare. Therefore, if a soft error alters the branch direction from taken to not taken, the error cannot be detected by SWIFT. This vulnerability is shown in Figure 2 (a). In this example, the branch is either taken (T) or not taken (NT), based on the result of the compare. Assume that the values of original registers R0 and R1 are equal in this example, and therefore the branch should be taken. Accordingly, shadow registers R0* and R1* are equal, and SWIFT updates the signature difference register with the signature of block T (sig(T)). Then, if a fault corrupts the branch direction from T to NT as marked by ① (e.g., an error changes R1), the signature difference register is overwritten

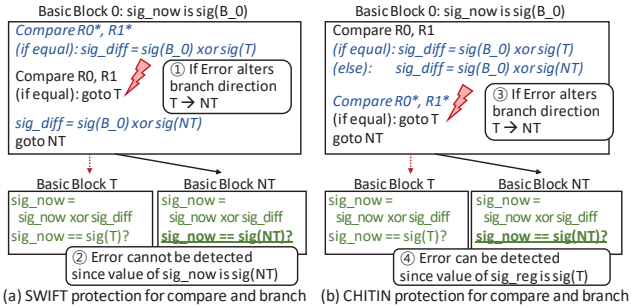


Fig. 2. While SWIFT cannot detect errors that corrupt the branch direction from taken to not taken, CHITIN detects the errors by isolating the signature update and the branch.

by the signature of block NT (sig(NT)). Consequently, even if control-flow wrongly jumps to NT block, the error cannot be detected, as marked by ②.

While the transformation of nZDC does not share this vulnerability, it suffers from a different problem. Since nZDC keeps the correct signature of every block as the same constant, it cannot detect errors that jump from one block to another. In addition, it requires more than 10 extra instructions to transform one branch instruction, which makes nZDC terribly inefficient.

The vulnerability in SWIFT occurs because the signature update instructions, which should protect the branch instruction, are partially dependent on the branch instruction. To break this dependency, CHITIN updates the signature difference register solely based on the original compare, and decides the direction of the branch based on the result of shadow compare. After the original compare, the signature difference register is updated with the signature of either the taken block or the not taken block, depending on the result of the original compare. CHITIN then replicates the compare with shadow registers and lets the branch instruction depend on the result of the shadow compare. Since the signature updates and the branch instruction depend on different compare instructions, CHITIN no longer suffers from errors that corrupt the direction of a branch. CHITIN also gives a unique signature to every block, resolving the vulnerability of nZDC.

Figure 2 (b) shows the CHITIN transformation for compare and branch. Note that the signature update instructions are dependent on the original compare with R0 and R1, and the branch is dependent on the shadow compare with R0* and R1*. Therefore, even if a soft error corrupts the branch direction from T to NT as marked by ③, the error cannot affect the signature update instructions. After the control-flow incorrectly jumps to block NT, the error is detected since the signature register contains the signature of T (sig(T)), as marked by ④. CHITIN can also detect errors in the opposite direction, corrupting the branch direction from NT to T, in a similar manner.

The signature register must be updated every time control-flow transfers to another basic block. On the other hand, its verification can be occasionally skipped since the corrupted signature register remain invalid across updates [12]. CHITIN therefore inserts checks only when needed. Checks are inserted in basic block with store instructions to avoid wrong memory writes. Checks are also inserted in the first basic block of a

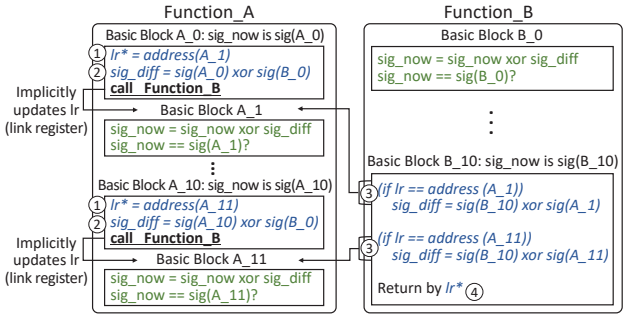


Fig. 3. To protect a function call, CHITIN updates the shadow link register (lr) and the signature difference register (sig_diff). For a function return, CHITIN updates sig_diff based on lr, and changes function return to utilize shadow lr.

loop if the loop does not have any basic block with store. The reason is because if a soft error corrupts the loop condition, the directions of the branch in the loop and its corresponding signature updates can be opposite twice during the loop. In this case, the signature is first incorrect but later gets corrected and CHITIN loses the chance to detect such errors.

B. Transformation for function call and return instructions

Previous in-thread replication techniques do not provide a concrete methodology to protect function calls. CHITIN therefore extends the signature-based control-flow checking to protect function calls and returns. CHITIN considers a function call as a combination of two instructions: a jump to the beginning basic block of the target function and an update of the original link register. To protect a function call, CHITIN updates the signature difference register with the signature value of the first basic block of the function, and updates the shadow link register explicitly. Faulty function calls that jump to illegal blocks are detected by checking the signature register.

The function return is more complicated since it can have multiple possible return points, where the correct one is dynamically decided by the link register. CHITIN protects the function return using signatures and a redundant link register. CHITIN updates the signature difference register based on one link register, and executes the return instruction based on the other link register, so that a single error cannot affect both the signature update instructions and the return instruction. Specifically, the original link register is compared against all possible return points, and if a compare result matches, the signature difference register is updated with the signature value of the matched return point. Then, CHITIN executes the function return based on the updated shadow link register.

Figure 3 shows the CHITIN transformation for function calls and returns. In this example, function A calls function B in basic blocks A_0 and A_10, and basic blocks A_1 and A_11 following the function calls are the possible return points of function B. To replicate the function call, CHITIN explicitly updates the shadow link register to be equal to the original link register as marked by ①, and the signature difference register with the signature value of B_0 (sig(B_0)), as marked by ②. To replicate the function return, CHITIN checks all possible return points to correctly update the signature difference register, as marked by ③. Finally, the function returns using the shadow link register (as marked by ④). With this transformation, CHITIN successfully protects function calls and returns in addition to providing redundancy for the link register.

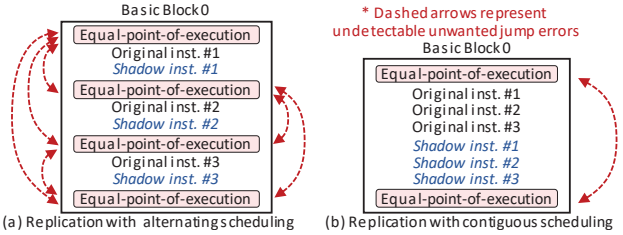


Fig. 4. While the number of equal-point-of-executions is proportional to the number of instructions in the block with alternating scheduling, contiguous scheduling induces only 2 such points regardless of the number of instructions.

C. Contiguous instruction scheduling strategy

Unwanted jump errors between equal-point-of-executions, points at which the original and shadow streams are identical, do not cause any mismatch between the two streams. Signature-based protection schemes can detect such errors between two different blocks, since the signature register contains the signature of the valid block. However, it cannot detect unwanted jump errors between equal-point-of-executions in the same basic block. Alternating scheduling strategy of previous in-thread replication schemes is highly vulnerable against intra-block unwanted jump errors due to the frequent number of equal-point-of-executions it induces. For a basic block with N original instructions, alternating scheduling induces $N+1$ equal-point-of-executions, which in turn opens ${}_{N+1}P_2$ possible cases of undetectable errors. Figure 4 (a) shows a basic block with alternating scheduling, where red dotted arrows represent the possible undetectable control-flow errors.

To reduce the number of equal-point-of-executions in each basic block, CHITIN adopts a contiguous original-shadow instruction scheduling strategy, which contiguously schedules as many instructions of one stream as possible. With this scheduling strategy, the number of equal-point-of-executions is uniformly 2, regardless of the number of instructions in the basic block. Figure 4 (b) shows an example of contiguous scheduling, the corresponding equal-point-of-executions, and the undetectable control-flow errors.

When scheduling the instructions contiguously, the following rules must be kept to preserve program semantics: i) The store instruction should precede the corresponding load-back checking instructions. Otherwise, the load-back checking would load the wrong value. ii) The signature update instructions should precede the corresponding control-flow instructions (jump, branch, function call, and function return). Otherwise, the instructions would alter the program counter immediately and skip the checking instructions. Therefore, CHITIN schedules all instructions in original execution stream first, which deals with register updates for original registers, store instructions, and signature updates for corresponding control-flow instructions. CHITIN then schedules instructions in the shadow stream, which deals with register updates for shadow registers, load-back checking for corresponding store instructions, and control-flow instructions. This is summarized in Figure 5.

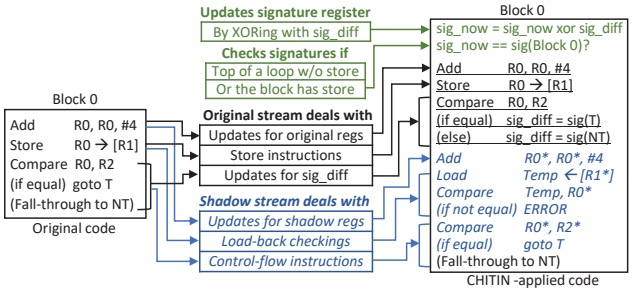


Fig. 5. Applying contiguous scheduling to a block, with the original stream underlined, and shadow stream italicized.

An exception is when a store overwrites data in memory that is accessed by previous load or store in the same basic block. With contiguous scheduling, the shadow memory instruction would access a value different from the original even in the absence of soft errors. Therefore, in the case of a write-after-read or write-after-write dependency in a basic block, CHITIN splits the contiguous scheduling, generating an additional equal-point-of-execution.

IV. Experiments

We conducted statistical fault injection experiments on an OpenRISC-based microprocessor to evaluate the fault coverage and performance overhead of CHITIN.

Benchmarks and binaries: We compiled 7 applications of Mibench [24] test suite with the following versions: no protection (Original), SWIFT, nZDC, and CHITIN. Note that since SWIFT and nZDC do not provide transformation rules for function calls and returns, we implemented both techniques by re-initializing the signature register at the top basic block of each function and each return point to avoid the false alarm due to the invalid signature register. We also implemented two additional versions of CHITIN in a step-by-step manner to see the effects of each sub-technique. The first version is CHITIN & alternating & naiveFunction that only replaces the transformation for control-flow instructions of nZDC as discussed in Section III-A. The second is CHITIN & naiveFunction that applies contiguous scheduling of Section III-C to CHITIN & alternating & naiveFunction. Finally, CHITIN is the comprehensive version that applies transformation for function calls and returns of Section III-B to CHITIN & naiveFunction.

Fault injection setup: We modified synthesizable Verilog codes of Mor1kx cappuccino microprocessor, which is the latest version of the OpenRISC1000 processor family, to flip random bits on hardware components. The modified Verilog code was simulated on an Icarus Verilog simulator [25]. For each execution, we randomly selected a bit on fault sites of a component and an execution tick to inject the fault. A fault site is a bit in a hardware component in which a fault can be inserted. The list of hardware components, fault sites for each component, and the number of faults, and the corresponding margin of errors with 95% confidence interval based on [26] are listed in Table II. With 7 benchmarks, 6 versions, and 10,800 faults per binary, we injected a total of 453,600 faults.

Failure classification: We only considered silent data corruption (SDC) as the failure case, since other cases can be noticed by the system. We consider a faulty run as SDC if the system reaches a valid endpoint of application without crash or timeout (runtime exceeds 2x of the fault-free run), and the output is different from that of a fault-free run.

TABLE II
The number of injected faults per component

Component	Fault sites	# of faults	Margin of error*
Register File	1,024	6,000	1.27%
Fetch/Decode Unit	200	1,200	2.83%
Decode/Execute Unit	216	1,300	2.72%
Execute/Control Unit	183	1,100	2.95%
Write/Back Unit	32	300	5.66%
Arithmetic Logic Unit	36	300	5.66%
Load-Store Unit	101	600	4.00%
Total	**1,792**	**10,800**	

*Margin of error with 95% confidence interval [26]

TABLE III
SDC classification of CHITIN

Failure sources	# of scaled SDCs
Intra-block unwanted jump error	9.3 (17.4%)
Address corruption of silent store	10.1 (18.8%)
Non-store to store alteration	8.5 (15.9%)
Inter-block unwanted jump error	12.3 (23.0%)
System call corruption	13.3 (24.8%)
Total	**53.6 (100%)**

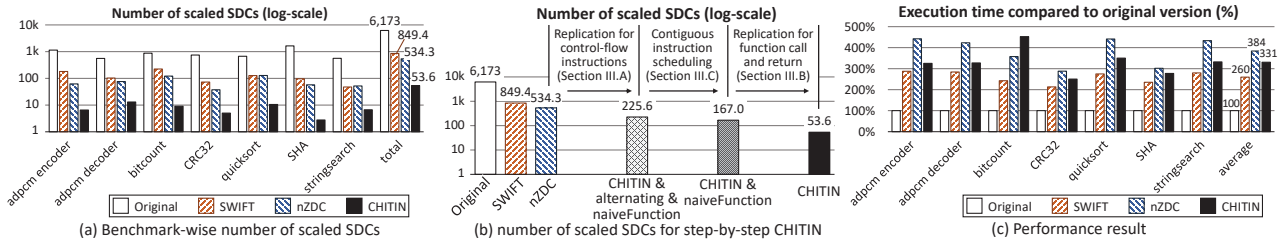
Comparison methodology: We inject one fault in every execution regardless of the configuration, but the performance degradation caused by different protection methods increases the period that the application is exposed to soft errors [27]. To fairly compare the fault coverage, we scaled the number of SDCs on each binary by multiplying it with the normalized execution time [27], [28].

A. Result: Fault Coverage

Figure 6 (a) shows the benchmark-wise results of the fault injection experiments for original, SWIFT, nZDC, and CHITIN. The X-axis represents the benchmarks, and the Y-axis represents the number of scaled SDCs in log-scale. The original version shows 6,173 SDCs. SWIFT and nZDC reduce this number to 849.4 and 534.3, respectively. Notably, CHITIN only shows 53.6 scaled SDCs on an average, which is 14.8x and 9.0x better than SWIFT and nZDC, respectively.

To evaluate the effect of each sub-technique in CHITIN, we also injected faults on two other versions of CHITIN as shown in Figure 6 (b). At first, replacing the control-flow transformation of nZDC by the one of CHITIN reduces the scaled SDCs of nZDC by around 58%. Applying contiguous scheduling in addition to the control-flow transformation reduces this number by 26%. Finally, employing CHITIN transformation for function calls and returns again reduces the scaled SDCs by 68%. This result demonstrates that each sub-technique of CHITIN effectively reduces the vulnerabilities it targets.

Around 0.07% of the faults still cause SDCs in CHITIN-applied applications. Table III shows the five sources of SDCs in CHITIN. i) Intra-block unwanted jumps between the remaining equal-points-of-executions are not detected. ii) The address corruption of a silent store cannot be detected by load-back checking. A silent store occurs when the data in the target address of a store is already equal to the data of the store instruction [29]. This vulnerability of load-back checking is already discussed by Didehban et al. [30]. iii) Faults that alter



(a) Benchmark-wise number of scaled SDCs (b) number of scaled SDCs for step-by-step CHITIN (c) Performance result

Fig. 6. CHITIN achieves the best error coverage as compared to the state-of-the-art techniques, while maintaining its performance penalty in-between them.

a non-store instruction to a store but does not induce a mismatch between two streams cannot be detected. iv) Unexpected jumps that make the control-flow return to the top of the previous basic block might not be detected, since signature difference cannot determine the direction; the signature difference from the first block to the second one is the same as the difference value from second to the first. v) Faults that corrupt the execution of system calls cannot be detected.

B. Result: Performance Overhead

Figure 6 (c) shows the execution runtime of each configuration. The X-axis represents the benchmarks, and the Y-axis represents the execution runtime normalized to the original runtime. On average, applying SWIFT, nZDC, and CHITIN to an application induce 1.6x, 2.8x, and 2.3x additional execution time overhead, respectively. CHITIN is slower than SWIFT for two reasons: i) CHITIN inserts an additional signature check for every loop without a store, and ii) CHITIN transformation for function return induces significant slowdowns, especially in benchmarks with many function calls. For example, *bitcount*, which has six functions with around 20 possible return points each, shows the highest performance degradation in CHITIN.

V. Conclusion

We present a comprehensive in-thread instruction replication, CHITIN. CHITIN judiciously combines nZDC and SWIFT with enhanced protection for compare and branch instructions. Furthermore, CHITIN provides directions to protect function calls and returns, using the shadow link register. Finally, the contiguous scheduling of CHITIN reduces vulnerability of intra-block unwanted jump errors. Statistical fault injection experiments show that the number of SDCs in CHITIN is 6.3% and 10.0% compared to SWIFT and nZDC, respectively.

VI. Acknowledgements

This work was partially supported by funding from National Science Foundation Grants No. CNS 1525855, CPS 1646235, CCF 1723476 - the NSF/Intel joint research center for Computer Assisted Programming for Heterogeneous Architectures (CAPA), NRF-2016H1A2A1909470 (Global PH.D. Fellowship Program, NRF, the Ministry of Education), NRF-2015M3C4A7065522 (Next-generation Information Computing Development Program, NRF, MSIT), 2014-3-00035 (High Performance and Scalable Manycore Operating System, IITP, MSIT), and Samsung Electronics Co., Ltd..

References

[1] R. C. Baumann, "Soft errors in advanced semiconductor devices-part I: the three radiation sources," *IEEE TDMR*, vol. 1, no. 1, 2001.

[2] P. E. Dodd and L. W. Massengill, "Basic mechanisms and modeling of single-event upset in digital microelectronics," *IEEE TNS*, vol. 50, no. 3, 2003.

[3] M. Snir *et al.*, "Addressing failures in exascale computing," *The International Journal of HPCA*, vol. 28, no. 2, 2014.

[4] E. Ibe *et al.*, "Impact of scaling on neutron-induced soft error in SRAMs from a 250 nm to a 22 nm design rule," *IEEE TED*, vol. 57, no. 7, 2010.

[5] "International Technology Roadmap For Semiconductors 2.0 - Executive Report," *[Online] http://www.itrs2.net/itrs-reports.html*, 2015.

[6] H. Qi, S. Ganesan, and M. Pecht, "No-fault-found and intermittent failures in electronic products," *Microelectronics Reliability*, vol. 48, no. 5, 2008.

[7] V. Sridharan and D. Liberty, "A study of DRAM failures in the field," in *ACM/IEEE SC*. IEEE, 2012.

[8] A. Haggag *et al.*, "Reliability/yield trade-off in mitigating "no trouble found" field returns," in *IOLTS*. IEEE, 2015.

[9] A. Haggag *et al.*, "Mitigating "No trouble found" component returns," in *IRPS*. IEEE, 2015, pp. 3C–5.

[10] X. Iturbe *et al.*, "A Triple Core Lock-Step (TCLS) ARM® Cortex®-R5 Processor for Safety-Critical and Ultra-Reliable Applications," in *DSN Workshop*. IEEE, 2016.

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

[12] G. Reis *et al.*, "SWIFT: Software implemented fault tolerance," in *CGO*. IEEE, 2005.

[13] M. Didehban and A. Shrivastava, "nZDC: A compiler technique for near Zero Silent Data Corruption," in *DAC*. IEEE, 2016.

[14] G. A. Reis *et al.*, "Automatic instruction-level software-only recovery," *IEEE micro*, vol. 27, no. 1, 2007.

[15] J. Yu *et al.*, "Esoftcheck: Removal of non-vital checks for fault tolerance," in *CGO*. IEEE Computer Society, 2009.

[16] S. Feng *et al.*, "Shoestring: probabilistic soft error reliability on the cheap," in *ACM SIGARCH CAN*, vol. 38, no. 1. ACM, 2010.

[17] D. S. Khudia *et al.*, "Efficient soft error protection for commodity embedded microprocessors using profile information," *ACM SIGPLAN Notices*, vol. 47, no. 5, 2012.

[18] K. Mitropoulou *et al.*, "DRIFT: Decoupled compiler-based instruction-level fault-tolerance," in *LCPC Workshop*. Springer, 2013.

[19] D. S. Khudia and S. Mahlke, "Harnessing soft computations for low-budget fault tolerance," in *MICRO*. IEEE, 2014.

[20] I. Laguna *et al.*, "IPAS: Intelligent protection against silent output corruption in scientific applications," in *CGO*. IEEE, 2016.

[21] Z. Zhang *et al.*, "Path sensitive signatures for control flow error detection," in *LCTES*. ACM, 2020.

[22] N. Oh *et al.*, "Control-flow checking by software signatures," *IEEE transactions on Reliability*, vol. 51, no. 1, 2002.

[23] N. Oh *et al.*, "Error detection by duplicated instructions in super-scalar processors," *IEEE Transactions on Reliability*, vol. 51, no. 1, 2002.

[24] M. R. Guthaus *et al.*, "Mibench: A free, commercially representative embedded benchmark suite," in *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*. IEEE, 2001.

[25] S. Williams, "Icarus verilog," *On-line: http://iverilog.icarus.com*, 2006.

[26] R. Leveugle *et al.*, "Statistical fault injection: Quantified error and confidence," in *DATE*. IEEE, 2009.

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

[28] H. So *et al.*, "EXPERT: Effective and flexible error protection by redundant multithreading," in *DATE*. IEEE, 2018.

[29] K. M. Lepak *et al.*, "Silent stores and store value locality," *IEEE Transactions on Computers*, vol. 50, no. 11, 2001.

[30] M. Didehban *et al.*, "NEMESIS: A software approach for computing in presence of soft errors," in *ICCAD*. IEEE, 2017.