EXPERTISE: An Effective Software-level Redundant Multithreading Scheme against Hardware Faults

HWISOO SO, Yonsei Universty, Republic Of Korea
MOSLEM DIDEHBAN, Arizona State University, USA
YOHAN KO*, Yonsei University, Republic Of Korea
AVIRAL SHRIVASTAVA, Arizona State University, USA
KYOUNGWOO LEE, Yonsei Universty, Republic Of Korea

Error resilience is the primary design concern for safety- and mission-critical applications. Redundant Multi-Threading (RMT) is one of the most promising soft and hard error resilience strategies because it does not require additional hardware modification. While the state-of-the-art software RMT scheme can achieve a high degree of error protection, our detailed investigation revealed that it suffers from performance overhead and insufficient fault coverage. This paper proposes EXPERTISE, a compiler-level RMT scheme that can detect the manifestation of hardware faults in all processor components. EXPERTISE transformation generates a checker-thread for the main execution thread. These redundant threads are executed simultaneously on two physically different cores of a multicore processor and perform almost the same computations. After each memory write operation is committed by the main-thread, the checker-thread loads back the written data from the memory and checks it against its own locally computed values. If they match, the execution continues. Otherwise, the error flag is raised. In order to evaluate the effectiveness of the proposed solution, we performed soft and hard error injection experiments on all the different hardware components of an ARM Cortex53-like μ-architecturally simulated microprocessor. Based on statistical fault injection campaigns, we have found that EXPERTISE provides 188× better fault coverage with 27% faster performance as compared to the state-of-the-art scheme.

CCS Concepts: • **Software and its engineering → Software fault tolerance**; • **Computer systems organization** → *Reliability*; • **Hardware** → *Transient errors and upsets*; **Error detection and error correction**.

Additional Key Words and Phrases: soft error, transient fault, redundant multithreading

ACM Reference Format:
Hwisoo So, Moslem Didehban, Yohan Ko, Aviral Shrivastava, and Kyoungwoo Lee. 2022. EXPERTISE: An Effective Software-level Redundant Multithreading Scheme against Hardware Faults. *ACM Trans. Arch. Code Optim.* 19, 4, Article 53 (September 2022), 25 pages. https://doi.org/10.1145/3546073

1 INTRODUCTION

Advances in semiconductor technology have integrated electronic systems in virtually all aspects of human life. Many modern applications, including autonomous cars, electricity distribution, and

*Corresponding author

Authors' addresses: Hwisoo So, shs7719@yonsei.ac.kr, Yonsei Universty, 50 Yonsei-ro, Seodaemun-gu, Seoul, Republic Of Korea, 03722; Moslem Didehban, moslem.didehban@asu.edu, Arizona State University, 660 S Mill Ave, Tempe, Arizona, USA, 03722; Yohan Ko, yohan.ko@yonsei.ac.kr, Yonsei University, 1 Yonseidae-gil, Wonju, Gangwon-do, Republic Of Korea, 26493; Aviral Shrivastava, aviral.shrivastava@asu.edu, Arizona State University, 660 S Mill Ave, Tempe, Arizona, USA, 03722; Kyoungwoo Lee, kyoungwoo.lee@yonsei.ac.kr, Yonsei Universty, 50 Yonsei-ro, Seodaemun-gu, Seoul, Republic Of Korea, 03722.

1544-3566/2022/9-ART53
https://doi.org/10.1145/3546073

embedded medical devices, are considered safety- or mission-critical applications because even small errors in their computations can lead to tragic consequences. Hardware faults are one of the main reasons for erroneous computations. Hardware faults are of two types: transient and permanents faults. Transient faults or soft errors are temporary malfunctions of computing devices that do not cause permanent damage to the microprocessor circuitry. In contrast, permanent faults or hard errors [1] have a lasting impact on the microprocessor functionality. The overall system failure rate due to hardware faults grows continuously due to the ever-increasing level of integration in different layers of computer-based systems, that is, more transistors per core, more cores per chip, and more chips per system [3, 40].

Although hardware solutions to protect from soft and hard errors like ARM Triple Core Lock-Step (TCLS) Cortex-R5 microprocessors [17] and HERMES [5, 10] exist, software-level error resilience schemes [7, 29, 49, 51] are desirable. This is because that such software-based protection schemes can be applied to all past, present, and future computing hardware, and also protect the parts of applications that require error resilience selectively. Furthermore, recent neutron beam testing experiments have demonstrated the effectiveness of software-level error resilience solutions [2, 18].

Among the existing techniques to protect the computation from hardware errors, redundant multithreading (RMT) approaches are desirable. They can provide a robust defense against hard and soft errors by utilizing the core redundancy of multicore microprocessor. The main idea of such techniques is to create two copies of the application's main thread, so-called leading- and trailing-threads, and execute them in parallel. The leading-thread sends critical data (such as the register operands values of shared memory write operations) to the trailing-thread for error detection. The trailing-thread receives critical values and checks them against its redundantly computed ones. If there is no mismatch, the leading-thread proceeds and submits the results, that is, writing data to the shared memory. Otherwise, the error flag is raised. Existing RMT schemes [29, 49, 51] have been considered as practical solutions for hardware unreliability, and researchers have enhanced their applicability to high-performance computing (HPC) [16] and even graphics processing unit (GPU) [14] domains.

However, our error coverage analysis revealed severe protection holes in the state-of-the-art RMT schemes. We observed that frequent and unprotected input replication operations (takes place on all shared memory read and system call operations) and shared memory update operations restrict the protection of existing RMT schemes to the computational/arithmetical operations of a program. Although they considerably improve the error detection capability, our extensive microprocessor-wide fault injection experiments revealed critical vulnerabilities.

We present EXPERTISE, a compiler-level protection scheme that provides microprocessor-wide transient and permanent fault detection from fail-continue faults (the computations are erroneous, but execution usually continues). We ran two slightly different versions of an application thread, named the main- and checker-threads, on physically different cores of a multicore processor. The main-thread performs all the program instructions and updates the memory state; in contrast, the checker-thread performs all computations and memory read operations redundantly but executes no memory write operation. Instead, it verifies the correctness of main-thread computations and write operations by loading back the main-thread written value from memory and checking it against its own locally computed value, based on the load-back checking from the previous in-thread replication schemes [6–9]. EXPERTISE separates the store execution and load-back checking process into individual threads and orchestrates two threads to cooperate. Owing to the replicated memory read operations and load-back checking, EXPERTISE transformation does not suffer from a vulnerable input replication process and frequent unprotected memory write operations.

[1] In this paper, we use the terms transient faults and soft errors as well as permanent faults and hard errors, interchangeably.

Table 1. Comparison with existing software-level solutions for hard and soft error detection

	Technique	Scope of Protection	Control-flow Error Detection	Error Detection Latency	Fault Coverage Evaluation Scheme	Additional Notes
En/decoding	ED4I [32]	Functional units and buses	YES	~10^9 instructions	Functional simulator	- Limited applicability -~3× runtime overhead
	ANCode [11], ANBCode [38]	Microprocessor datapath and memory	NO	~10^6 instructions	Functional simulator	- ~10× runtime overhead
	Delta Encoding [21]	Microprocessor datapath and memory	NO	~10^3 instructions	Functional simulator	- ~5× runtime overhead
Multithreading	SRMT [49]	Microprocessor datapath	YES	~10^2 instructions	Functional simulator	- ~4× runtime overhead - Frequent spof[a]
	GPU RMT [14, 48]	GPU datapath	YES	~10^2 instructions	Not evaluated	- 1-126× runtime overhead
	EXPERTISE (This work)	Microprocessor datapath and private caches	YES	~10^2 instructions	Cycle-accurate simulator	- ~3× runtime overhead

[a] spof: single-point-of-failure

However, the load-back checking of EXPERTISE is still vulnerable in the case of a silent store [25], which attempts to write the same value to a memory location already present. If a soft error corrupts the address of the silent store, load-back checking cannot detect the corruption of the store operation [9]. EXPERTISE eliminates such vulnerabilities by adopting the silent store avoidance from the previous in-thread replication scheme [9] and extending it into RMT. Further, we reduced the performance overhead of the software-level RMT scheme by store packing optimization. EXPERTISE requires thread synchronization on memory write operations to guarantee coherence memory accesses for redundant threads and facilitates a check-after-write error detection policy. To reduce the number of such synchronization points, we identify independent successive store operations and group them at compile time. Then, we add just one synchronization point to a pack instead of a single memory write operation. Our analysis reveals that EXPERTISE can provide 188× better fault coverage and 27% faster performance than the state-of-the-art software-level RMT scheme.

2 SOFTWARE SOLUTIONS FOR HARDWARE ERROR PROTECTION

Existing software-level solutions for hard and soft error detection can be divided into i) program data encoding/decoding and ii) redundant multithreaded execution, as described in Table 1.

2.1 Software encoding/decoding

Researchers have thoroughly investigated software-level data and computation encoding/decoding for hard and soft error protection [11, 21, 31, 37, 38]. The ED4I source-code transformation [31] generates a functionally equivalent AN-coded version of the program. In the AN-coded version, all program data and computations are multiplied by a constant factor A, after which the ED4I executes original and encoded versions of the program and compares their outputs for error detection – the final output of the AN-coded version of the program should be A-multiples of the original ones; however, these software encoding/decoding techniques are inefficient because of the long error detection latency, significant performance overhead, and limited fault coverage. The error detection latency is defined as the time between the occurrence of an error and its detection. If the error detection latency is short, the system can have a longer response time to the safe mode. Furthermore, because of the program dynamicity (interaction with a user and other applications during the execution), error detection on the final program output is not practical in many cases. However, the error detection latency of the ED4I scheme is long because it is proportional to program execution time.

Second, software encoding/decoding techniques incur huge performance overhead. The schemes presented in [11, 38] improve the efficacy of ED4I by utilizing more advanced coding mechanisms, for example, ANB and ANBD encoding, and addressing practical issues, that is, overflow through an intermediate-level compiler transformation. Unfortunately, such implementation of program encoding/decoding accompanies significant performance overhead (6× to more than 60× as reported in [38] and up to 250× reported in [21]). To overcome these limitations, [21] introduced delta-encoding, which is a source-to-source transformation that combines AN-encoding with instruction duplication. However, software-only delta-encoding transformation still requires around 4.08 × execution time to achieve its maximum fault coverage, compared to unprotected architectures.

Finally, software encoding/decoding techniques suffer from a lack of fault coverage. AN-based encoding schemes can fail to protect the system against errors in logical and memory operations [37] and hard-to-detect control-flow errors [21]. Furthermore, they significantly reduce the range of computations. For example, an unsigned char variable (8 bits) typically holds any value between 0 and 255 (256 valid numbers). However, after applying an AN-code with factor 4, it reduces the number of valid numbers by 4 – from 256 to 64! Therefore, AN-based encoding/decoding solutions are not practical for programs that use a wide range of values.

2.2 Software spatial redundant execution

Software spatial redundant techniques rely on the inherent core redundancy presented in modern multi/many-core microprocessors to achieve error protection. They run two functionally identical versions of an application on two separate cores and check the results for error detection. Thread-level redundancy-based schemes can provide adaptive and flexible resilience [16].

Traditional thread-level redundancy schemes execute the original and replicated threads and compare their result with hardware support. These schemes execute these threads on simultaneous multithreading (SMT) processor [34, 36, 46] or chip multiprocessor (CMP) [13, 30]. Fingerprinting-based schemes [22, 42, 43] optimize the hard-wired result comparison of threads on the CMP-based schemes by summarizing the execution results of each thread into hash-based fingerprints [43]. However, these techniques require additional hardware modification to support their features and the following hardware costs. For the sake of simplicity, this paper focuses on the software-based schemes since they can be applied to the general processors without additional modifications.

The software-based redundant multithreading (SRMT) scheme [49] is the first software-only multithread scheme that can potentially detect both soft and hard errors. SRMT executes two redundant threads, leading-thread and trailing-thread, on two different physical cores. The leading-thread is responsible for shared memory read and write accesses, where the shared memory is protected by hardware-level solutions such as error correction code (ECC). In contrast, the trailing-thread is responsible for the detection of errors in the register values that the leading-thread used for shared memory read and write operations. To satisfy the above-mentioned functionality, SRMT exploits a circular software queue for communication between the leading-thread and the checker-thread. The leading-thread enqueues the register values for the input of shared memory read and write accesses (address for a load instruction, and address and data for a store instruction) to the queue before the execution of such instructions. When the trailing-thread reaches the corresponding execution point, it dequeues the register values from the queue, and compares them with its locally calculated register values. Because the trailing-thread does not execute the load instruction, the leading-thread also needs to send the loaded value after the load instruction to the trailing-thread.

Fig. 1 shows the SRMT transformation for a simple snippet of code. In this example, both threads execute computational instructions redundantly, except for the load and store instructions. For the load instruction, the leading-thread first sends the value of the load address register operand (r_4) to the trailing-thread and then executes the memory read operation. The `sendBuf` (r_4) function

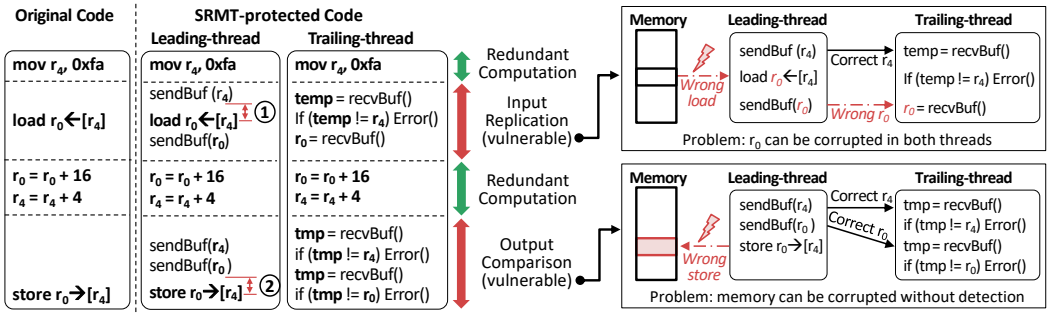


Fig. 1. SRMT transformation duplicates the original program execution thread. Only the leading-thread performs memory operations. The trailing-thread validates the correctness of memory instruction register operands while the execution of memory operations remains unprotected.

shown in the figure is responsible for sending the value of the load address register operand to the trailing-thread. Then, once the leading-thread receives the requested data from the memory, it sends the loaded data for the trailing-thread by calling the `sendBuf(`r_0`)` function. The trailing-thread receives the value of the load address register operand from the buffer by calling `recvBuf()` and checks the load address by comparing the received value against its own locally computed value for the load address register operand (r_4). If they match, the trailing-thread considers that the load address is correct, reads the next value from the buffer, and places it in the corresponding data register (r_0). Once the leading-thread reaches a memory write operation, it sends the values of data and address registers of the store (r_0 and r_4) to the trailing-thread for error detection and then performs store instruction. Similar to load memory address register value checking, the trailing-thread retrieves the leading-thread computed store data and register values from the buffer and checks them against its own redundantly computed values.

Our detailed analysis of fault coverage of SRMT transformation reveals two major vulnerability windows. First, it is vulnerable to input replication process. In redundancy-based error detection strategies, input replication provides the same inputs for redundant computations. The input replication process can be considered as a single point of failure in redundancy-based error detection schemes. If any error affects the input data before (or during) the input replication, the error remains undetected. This is because both redundant executions start their computations with the same wrong data and produce the same wrong output. One of the main disadvantages of SRMT transformation is frequent input replication operations that occur after all shared memory read accesses and system call operations. In SRMT transformation, the leading-thread performs shared memory read and system call operations and sends the results (the loaded value from the memory or system call return values) to the trailing-thread. Therefore, if any error occurs during such single-instance operations, they will propagate to the trailing-thread, leading to a user-visible failure. For instance, assume an error that permutes the effective address calculation of the leading-thread load instruction shown in Fig. 1 (marked as ①). Because of this error, the wrong value is loaded into the load instruction destination register (r_0). Then, the leading-thread sends the faulty value to the trailing-thread, and both threads continue their execution with the same wrong value in the register r_0. Note that this fault not only corrupts the register (r_0) in both threads equally; it is also not detected by the checking instructions of trailing-thread because it compares the value in the queue that is inserted before the fault. Faults on pipeline stage register bits while processing load operations, memory address generation units or load/store units are examples of errors that may remain undetected in the SRMT scheme because of the frequent input replication process.

Second, SRMT transformation is vulnerable during the output comparison process. Output comparison is defined as the process of checking the results of redundant computations for error detection and committing results if there is no discrepancy. In SRMT transformation, a shared memory write operation is considered as an output comparison; therefore, SRMT should detect incorrect store instructions. As shown in Fig. 1, the leading-thread sends the value of data and address registers (r_0 and r_4) to the trailing-thread before the execution of the store instruction, and the trailing-thread compares the register values from the leading-thread against its locally calculated register values to ensure that the store instruction is executed with the correct input. However, the errors that occur after checking the results and during the result submission process (marked as ② in Fig. 1) will directly affect the memory write operation without being detected. This is because trailing-thread does not verify the execution of the store directly, but it only checks the old snapshot of the register values that leading-thread sent before the execution of store instruction. Note that soft errors that occur in the vulnerable period (marked as ② in Fig. 1) does not affect the execution of `sendBuf(`r_0`)` and `sendBuf(`r_4`)`. Therefore, trailing-thread considers that the received values of r_0 and r_4 from the leading-thread are correct, regardless of the correctness of the execution of the original store instruction. Examples include errors in pipeline data path registers while processing store instructions, store effective address functional units, store buffers, and even store register operands (after sending their data to the trailing-thread and before being read for memory write operation). Moreover, because store operations can stay in the microprocessor store buffer for a long time, they have a considerably high exposure time, leading to a high chance of corruption. Note that hardware-level memory protection such as ECC cannot cover this vulnerability because the system cannot be aware of the corruption of the store instruction, and the system calculates the ECC bits based on the corrupted store instruction.

In this study, we mainly concentrate on the SRMT scheme. However, all existing software-only RMT schemes suffer from the above-mentioned vulnerable intervals. For instance, the DAFT [51] technique improved the performance overhead of SRMT by applying an in-thread instruction duplication scheme for the entire use-def chain of register operands of volatile memory write accesses, rather than verifying their correctness in the trailing-thread. Therefore, DAFT-protected programs need no synchronization between redundant threads and execute faster. Not only does DAFT suffer from all SRMT protection holes, but its fault coverage is limited to soft errors. The COMET [29] scheme also improves the performance overhead of SRMT by applying several optimizations, including in-lining `sendBuf()` and `recvBuf()` functions, and reducing the number of such functions with packing store data and memory register values together. Several schemes [14, 48] have applied the SRMT error detection strategy to GPUs. Furthermore, RedThreads [16] takes advantage of SRMT flexible error detection and accomplishes programmer-tunable protection by providing programming-language support for applying partial SRMT in HPC applications. Overall, because all existing software-level redundant multithreaded schemes mainly attempt to improve the performance overhead of SRMT, they suffer from SRMT protection holes. The only exceptional solution is process-level redundancy (PLR) [41], which replicates the application into redundant processes and compares the external results of them. However, PLR suffers memory overhead due to the memory replication and can only be applied in a process-level granularity [51].

3 OUR APPROACH

3.1 Basic idea of EXPERTISE

We present EXPERTISE, a compiler-level RMT approach that eliminates the input replication and output comparison vulnerability windows of state-of-the-art schemes. The main design goal of EXPERTISE is to provide processor-wide transient and permanent fault detection. More specifically,

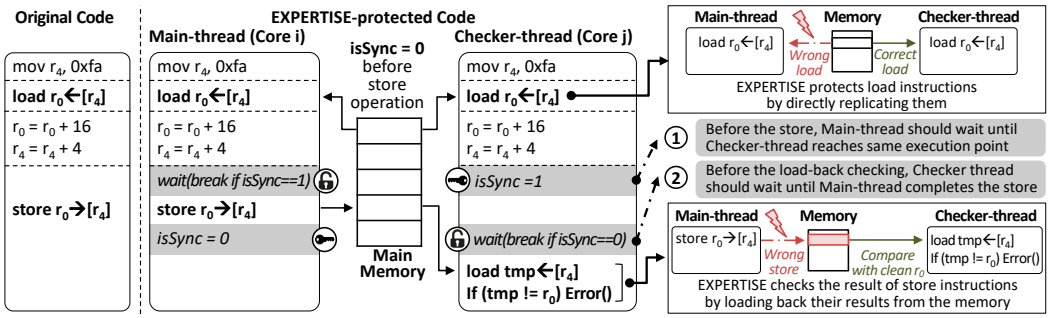


Fig. 2. EXPERTISE transformation runs two copies of a program thread and synchronizes them on store operations. The main-thread performs store, and the checker-thread verifies the correct execution of the store by loading the written value from the memory and checking it against the locally-computed value.

we consider the single bit-flip model including flip-to-1 and flip-to-0 [19, 33] for transient faults and the single stuck-at fault model [12] for permanent faults.[2] EXPERTISE targets single-threaded applications running on a multicore processor in which the memory subsystem (excluding core private caches) is protected by ECC. The EXPERTISE transformation assigns a checker-thread to the main application thread. The checker-thread is executed on a different core than that executing the main execution thread. The key idea here is to orchestrate the main- and checker-threads in such a way that after each memory write operation is committed by the main-thread, the checker loads the written value from the memory back and checks that against its own locally computed value. The major features of EXPERTISE are as follows.

1) The EXPERTISE transformation eliminates single-point-of-failures of the input replication process. As discussed in Section 2, the input replication process for memory read instructions introduces a protection hole in the existing RMT schemes. EXPERTISE eliminates such vulnerabilities by adopting a memory read instruction duplication strategy. Fig. 2 illustrates the EXPERTISE transformation. In this example, memory read and write instructions and the corresponding checking instructions are indicated in **bold** and instructions for synchronizing main-thread and checker-thread are indicated in *italics* with gray boxes. As shown in Fig. 2, the EXPERTISE transformation replicates memory read instructions as well as computational instructions. With duplicated memory read instructions, EXPERTISE can achieve redundancy of load instructions. The main challenge here is how to provide input replication coherency (how to guarantee that both redundant threads will receive the same data from the shared memory). For instance, consider a case in which the main-thread reads some value from memory, performs computations, and store the updated data back to the memory. Later, once the checker-thread performs the redundant read operation, it receives a different value from what the checker-thread received, which eventually led to false error detection. Therefore, we need a mechanism to ensure that both threads perform their previous memory read operations before writing to the memory. EXPERTISE uses a shared variable (`isSync`) and a busy-waiting mechanism to provide the required ordering between redundant thread executions. The operations required for coherent input replication are marked as ① in the figure. Although it is possible to place such memory barrier operations on different places of the code (i.e., after load and store instructions) to satisfy the input coherency problem, we choose to

[2]EXPERTISE cannot detect multi-bit faults if two redundant threads are corrupted identically. However, multi-bit faults are frequent compared to single-bit faults [24] and redundancy-based scheme that targets single-bit faults can detect most of the multi-bit faults if the scheme can detect single-bit faults sufficiently [35].

put them immediately before store operations because it will give us the minimum number of synchronization points. Generally, number of write operations is lesser than read operations [45].

2) The EXPERTISE transformation does not suffer from a vulnerable output comparison process. The existing software-level multithreaded scheme suffers from a very fragile output comparison/delivery process owing to the indirect store verification that only checks the snapshot of the register values before the store instruction. To solve this vulnerability, EXPERTISE adopts the load-back checking from the existing in-thread instruction replication schemes [6–9] that directly verifies the correctness of the output delivery process to eliminate vulnerability from the output comparison. EXPERTISE extends such comprehensive error checking into RMT by assigning execution of store instruction to the main-thread, and the load-back checking process to the checker-thread. Therefore, EXPERTISE verifies the right execution of a store instruction in the main-thread by loading the main-thread written data from memory in the checker-thread and comparing it against the redundant computed version of data, as shown in Fig. 2. Because this load-back checking directly accesses the result of the store instruction of the main-thread, EXPERTISE can effectively detect a soft error even if the fault corrupts the execution of the store instruction.

However, the load-back checking requires strict order between the original store instruction and the following load-back checking process. While in-thread level schemes can easily accomplish this requirement by the instruction ordering in identical threads, accomplishing such a requirement on RMT requires the communication between two threads. Therefore, The challenge for combining load-back checking and RMT is to ensure that the checking operation in the checker-thread will take place after the write operation in the main-thread. EXPERTISE addresses this problem by inserting a memory barrier after memory write operations (marked ② in Fig. 2). When the main-thread performs a memory write operation, it rewrites the value of the `isSync` flag as 0. Then, the checker-thread breaks the wait and verifies the result (computations and execution of the memory write operation) of the main-thread. The checker-thread accomplishes this by loading the main-thread written value from the memory and checks it against its own locally computed value. Note that in the EXPERTISE error detection strategy, errors altering the effective address of the store instructions are also detected. If the main-thread updates an incorrect memory location, the checker-thread reads the data from the right memory location and detects a mismatch. Further, EXPERTISE also can detect a fault on checker-thread by load-back checking. In this case, the load-back checking can detect mismatch by comparing the result of store instruction from the main-thread that is not affected by the fault and the corrupted locally-computed values of checker-thread.

3.2 Solution for erroneous silent store

Even though post-store detection discussed in Section 3.1 can provide high-level fault coverage, it can still be vulnerable to errors that impact the address of silent stores [9]. A memory write operation is silent if it writes a value in the memory that is presented in the target memory location even before the execution of the store [25]. Therefore, even if it skips over the execution of store instruction for such cases, the result is still correct. Silent stores are frequent in real-world applications, and independent profiling results shown in [20, 26] reveal that up to ∼75% (on average more than ∼ 30%) of dynamic stores in SPEC benchmark programs are silent! In this subsection, we first introduce the vulnerability of EXPERTISE from the address corruption of a silent store. Then, we present the EXPERTISE solution based on the silent store avoidance from the previous in-thread replication scheme [9] that solved the silent store problem of load-back checking. Specifically, EXPERTISE extends this solution into RMT by assigning redundant silent store checkings into main- and checker-threads and combining synchronization signal and transfer of silent store checking result into one variable. For the sake of understanding, we first present two naive solutions to

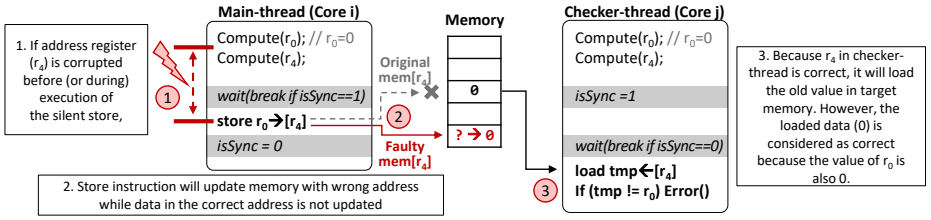


Fig. 3. Load-back checking is vulnerable to an error affecting the memory address of silent stores.

solve the vulnerability issue from silent stores and discuss their drawbacks. Then, we present our solution to solve both the vulnerability of load-back checking in EXPERTISE and the drawbacks of the naive solutions.

1) Load-back checking of the EXPERTISE transformation cannot detect the address corruption of a silent store. Assume that a soft error corrupts the address of a silent store, and the main-thread writes a value in the wrong memory location. Then, the load-back checking mechanism of EXPERTISE presumes no errors in computations because the checker-thread still reads the expected value from memory. Fig. 3 shows the vulnerability of the EXPERTISE transformation in the case of silent stores. In Fig. 3, thread synchronization and error detection operations are indicated in *italics* with gray boxes and **bold**, respectively. Fig. 3 depicts a code snippet that computes r_0 and r_4 registers, and stores r_0 into the memory location (mem[r_4]). Note that in this example, both the results for r_0 computation and the original data in the memory location (mem[r_4]) before the store operation are zero, and therefore the store operation is silent. The vertical line (marked as ①) shows the vulnerable period of register r_4. If any errors alter the value of r_4 on the main-thread during this vulnerable interval, it leads to an unwanted write to a random memory location (marked as ②) through the execution of memory write instruction. However, checker-thread error detection operations cannot detect the manifestation of such an error because the correct value has been sitting in the target memory location (marked as ③). Note that including errors directly affecting the value of register r_4, errors hitting the address calculation of store instruction also remain undetectable. Such undetectable cases include errors on the address (or immediate value) of store instruction in the instruction cache and microprocessor pipeline, errors affecting functional units during the computation for the effective address of store instruction, and errors affecting the load-store unit. The importance of memory write operations, the frequency of silent stores, and the sensitive components during the execution of stores make silent store vulnerability non-negligible.

2) Avoiding silent store can solve the silent store problem of load-back checking, but naively detecting a silent store in one thread induces another vulnerability. Avoiding the execution of silent stores can eliminate the vulnerability of load-back checking in EXPERTISE. The key idea to avoid the execution of silent stores is to detect silent stores dynamically and skip their execution. Note that skipping silent stores can eliminate the vulnerability of load-back checking. It does not affect the original behavior of the program because it does not change any architectural states or memory instances. Intuitive method for compiler-level silent store detection and avoidance is to read the value inside the store's target memory before its execution and check it against the store value register. If they are equal, the store is silent, and its execution can be skipped [9]. We refer to this process as a `silent store checking`. Fig. 4 (a) shows a first-cut silent store checking solution. In this simple solution, the main-thread does not perform the store operation if the value that already exists in the memory is the same as the store value.

This solution can successfully skip the execution of silent stores in fault-free execution, thus avoiding the silent store problem. However, it fails when a fault impacts the address register operand

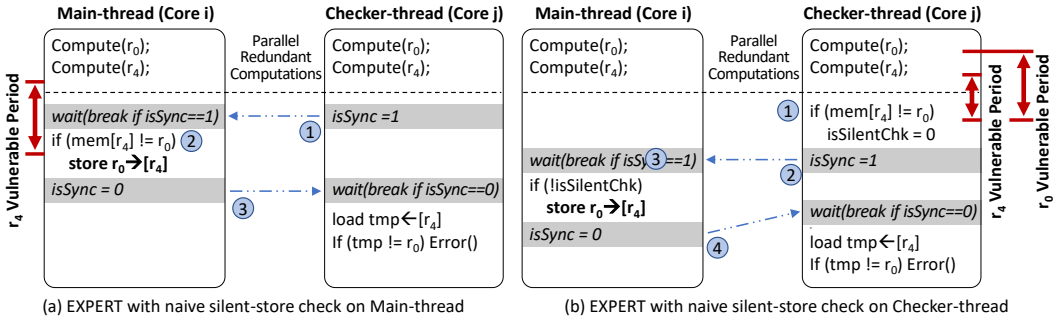


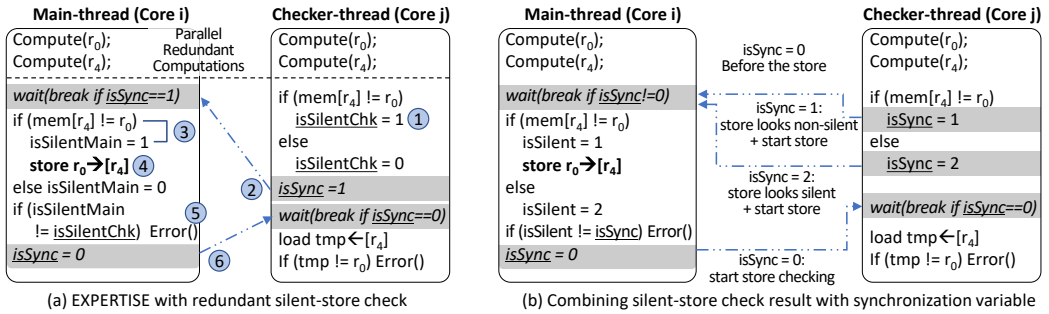
Fig. 4. EXPERTISE transformation with naive silent store checking on the main-thread (a) and checker-thread (b). r_4 is vulnerable in (a) if the original store is silent. In contrast, r_0 and r_4 are vulnerable on (b) if the original store is not silent, but the corrupted register values make the result of the silent store checking silent.

of the silent store checking. Consider a case in which the store operation is silent, and an error has changed its memory address register (r_4 in Fig. 4 (a) to r_4*[3]) before the execution of the silent store checking operation (marked as r_4 `Vulnerable Period`). In this case, an incorrect value (present at the memory location r_4*) will be compared against the store value register, and if they happen to be different (which is more likely), the store operation will update the wrong memory location (r_4*). However, because the store is silent, the checker-thread error detection operations cannot catch the error, the correct value will be loaded from memory, and no mismatch will be observed.

3) EXPERTISE eliminates the vulnerability to the one-side silent store checking by redundant silent store checking. The above vulnerability can be avoided by dividing the execution of the store operation and the silent store checking into different threads as shown in Fig. 4 (b). Intuitively, another first-cut solution is to proceed with a silent store checking on the checker-thread, and the main-thread utilizes the result of the silent store checking from the checker-thread to determine whether to skip the store operation. However, this approach fails to protect store operations that are not silent. Assume that a fault has changed the data register (r_0 to r_0*) of the store operation before the execution of the silent store checking operation (marked as r_0 `Vulnerable Period`). If the value of the corrupted data register (r_0*) is the same as that of the data in the target address (r_4), the checker-thread will assume that the store operation is silent. Therefore, the main-thread, which utilizes the result of the silent store checking on the checker-thread, will skip the store operation. This error cannot be observed by load-back checking on the checker-thread because it utilizes the corrupted data register (r_0*) again. Consequently, the target memory location will not be updated, although the original store is not silent. Note that such a failure case can occur in a similar manner when an error changes the memory address register (r_4).

EXPERTISE employs redundancy in silent store checking operations by performing them separately on both the main- and checker-threads. This is because the main reason behind the ineffectiveness of the two first-cut solutions is that there is no redundancy for silent store checkings. Fig. 5 (a) shows the EXPERTISE transformation with a redundant silent store checking. In this example, thread synchronization and error detection operations are indicated in *italics* with gray boxes and **bold**, respectively. Further, the variables for inter-thread communication between the main-thread and checker-thread are underlined. EXPERTISE uses two variables, `IsSilentMain` and `IsSilentChk`, to hold the results of silent store checking operations conducted by the main-

[3]* means erroneous value due to hardware faults



(a) EXPERTISE with redundant silent-store check

(b) Combining silent-store check result with synchronization variable

Fig. 5. EXPERTISE employs the redundancy of silent store detection (a). To avoid the additional communication, EXPERTISE combines the synchronization variable and the result of the silent store checking (b).

and checker-threads, respectively. For correctness, the checker-thread silent store checking should take place before the execution of the store instruction.

Once the checker-thread completes silent store checking, it sets the synchronization flag (marked as ② in Fig. 5 (a)) and main-thread execution continues by performing three conditional branch operations. First, `IsSilentMain` is set if the store is silent inside the main-thread (marked as ③). If the store is not silent in the second condition statement, main-thread advances by the execution of store instruction (marked as ④). Third, the main-thread compares the result of its own silent store checking test (`IsSilentMain`) against the checker-thread (`IsSilentChk`), and raises the error flag in the case of mismatch (marked as ⑤). If they are the same, the main-thread resumes the execution of the checker-thread by setting the value of `isSync` as 0 (marked as ⑥).

The silent store avoidance with redundant silent store checkings eliminates all possible vulnerabilities from the address corruption of silent store operations. Note that this solution also eliminates the vulnerabilities of the naive solutions in Fig. 4 (a) and (b). This is because a single fault in the vulnerable period of the naive solutions (marked as r_0 `Vulnerable Period` and r_4 `Vulnerable Period` in Fig. 4 (a) and (b)) can only corrupt either `isSilentChk` or `isSilentMain`, and comparing the results of the redundant silent store checking (marked as ⑤ in Fig. 5 (a)) can detect this fault.

However, the redundant silent store checkings in Fig. 5 (a) induce another vulnerability against soft errors. This solution requires two variables for inter-thread communication (`isSync` and `isSilentChk`), as underlined in Fig. 5 (a). Without additional hardware support, these variables are located in the memory because both threads should be able to access them. Therefore, write operations for these variables are implemented using store instructions. The problem is that these instructions can also be affected by soft errors. If a soft error corrupts the address of the store instruction for `isSilentChk`, the store instruction corrupts the data in the wrong address. If the corrupted write operation of `isSilentChk` was originally silent (the value of `isSilentChk` that the checker-thread is going to send to the main-thread is the same as the previous value that remains in the memory), the main-thread cannot be aware of this corruption. This is because the value of `isSilentChk` that is not updated owing to the address corruption of the write operation is the same as the value of `isSilentMain`. Note that `isSync` does not suffer such vulnerability, because soft errors that affect a store operation of `isSync` will break the communication rules between the main-thread and checker-thread, which will result in system-visible failures such as an infinite loop. Further, store instructions for `isSync` are never silent.

To resolve the vulnerability from the additional communication variable, we combine `isSync` and `isSilentChk`, as shown in Fig. 5 (b). While the main-thread of the previous solution in Fig. 5 (a)

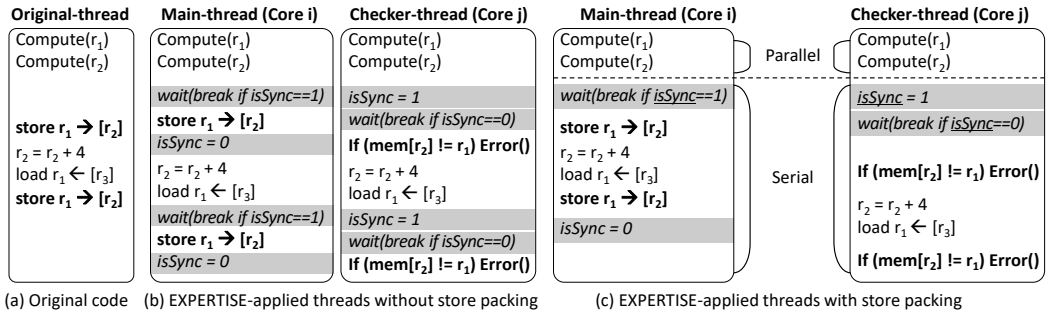


Fig. 6. To apply pure EXPERTISE to the original code (a), as many inter-thread synchronizations as the number of store instructions are required (b). EXPERTISE store packing reduces the number of synchronization points by executing multiple stores and adjacent instructions inside one serial segment of the execution (c).

waits till the checker-thread attains the break condition `isSync==1`, the new solution in Fig. 5 (b) utilizes the break condition `isSync!=0`. Therefore, the checker-thread can awake the main-thread by writing any values except 0 to `isSync`. The new solution utilizes a variety of the value of `isSync` to encode the functionality of `isSilentChk`. When the checker-thread awakes the main-thread, `isSync` is set to 1 if the result of the silent store checking in the checker-thread indicates that the store instruction is not silent. Otherwise, if the result of the silent store checking in the checker-thread indicates that the store instruction is silent, the checker-thread sets `isSync` as 2. The main-thread utilizes the local variable `isSilent` as a result of silent store checking in the main-thread, and the main-thread compares `isSilent` with the received value of `isSync`, to check the validity of redundant results of silent store checkings. Finally, the main-thread sets `isSync` to resume the execution of the checker-thread. Consequently, the solution in Fig. 5 (b) requires only one communication variable `isSync`, which is the same as the original solution without a silent store checking, as shown in Fig. 2.

4 PERFORMANCE OPTIMIZATION

The performance advantage of RMT comes from the observation that redundant threads can be executed on physically different cores on a multicore processor to detect hardware faults. The downside of this assumption is that redundant thread communication (marked in italic with gray in Fig. 5 (b)) is becoming inter-core communication which is considerably slower than intra-core communication. Previous analysis of RMT execution of GPUs reveals that, on average, 80% of the redundant execution overhead is caused by inter-core communication [14]. For the sake of understanding, we first introduce store packing optimization with the basic EXPERTISE without load-back checking, which is discussed in Section 3.1, and discuss the challenges of applying a silent store checking that is discussed in Section 3.2 with the store packing optimization. Second, we present improved store packing optimization with fingerprint-based load-back checking to relieve the conflicts of register values when we apply silent store checking with store packing optimization. Finally, we apply compressed synchronization, which combines the multiple results of silent store checkings and synchronization variables into one shared variable.

4.1 Basic idea of store packing optimization

The main goal of our performance optimization is to reduce the number of inter-core communications based on the main idea of Give-N-Take [47] and Sink-N-Hoist [50]. The number of thread synchronization points in the baseline EXPERTISE is equal to the memory write operations. We

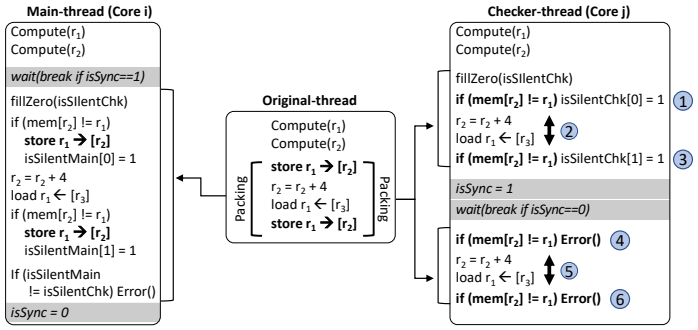


Fig. 7. Directly applying store packing to EXPERTISE cannot ensure the correctness of load-back checking.

apply store packing, which exploits only one synchronization point for a group of store instructions. With the store packing optimization, EXPERTISE executes multiple store instructions in main-thread, then checker-thread executes the corresponding load-back checking instructions with one synchronization point. Fig. 6 (a), (b), and (c) describe an example original code, EXPERTISE-applied code without store packing, and EXPERTISE-applied code with store packing optimization, respectively. Note that in this example, we do not apply silent store avoidance discussed in Section 3.2 to concentrate on the concept of store packing optimization. Because there are two store instructions in the original code, as shown in Fig. 6 (a), two synchronizations are required for the pure EXPERTISE transformation, as shown in Fig. 6 (b). Instead, EXPERTISE merges these two store instructions into a pack with one synchronization, as shown in Fig. 6 (c). Note that the computational instructions between the store instructions are also packed by store packing. Store packing sacrifices parallelism because packed instructions, including computational instructions between stores, should be executed serially. However, it improves the overall runtime performance by reducing the number of thread synchronizations.

However, simply applying store packing optimization can be inefficient because of the silent store checking discussed in Section 3.2. Fig. 7 shows the conflicts of register values when applying store packing optimization to EXPERTISE without considering the redundant executions of packed instructions on the checker-thread. The transformation of the checker-thread consists of two parts. The first part (marked as ①, ②, and ③) deals with silent store checking and results sent for the two store operations and are performed before synchronization signals (`isSync = 1` and `wait(break if isSync==0)`). The second part is for reading the results of the main-thread from the memory and checking them for error detection (marked as ④ , ⑤, and ⑥). In this example, both the first and second original store instructions access r_1 and r_2 as data and address operands where the packed instructions between them (② and ⑤) modify r_1 and r_2. Therefore, the transformed instructions for the first store instruction (① and ④) should be executed based on the values of r_1 and r_2 before the packed instructions (② and ⑤), and those for the second store instruction (③ and ⑥) should be executed based on the values for r_1 and r_2 after the packed non-store instructions. Naive store packing on EXPERTISE harms the above conditions: i) The load-back checking for the first store in the second part (④) accesses r_1 and r_2 after the packed instructions (②), while the main-thread does the first store based on the values for r_1 and r_2 before the packed instructions (②). ii) Because the packed instructions are executed twice (② and ⑤), the load-back checking for the second store in the second part (⑥) might access the wrong r_1 and r_2 values. In this example, the r_2 value is increased by 8 for the load-back checking of the second store, while the main-thread does the second store with the r_2 value that is increased by 4. Note that naively eliminating redundant packed instructions (②) in the first part can solve this problem, but it harms

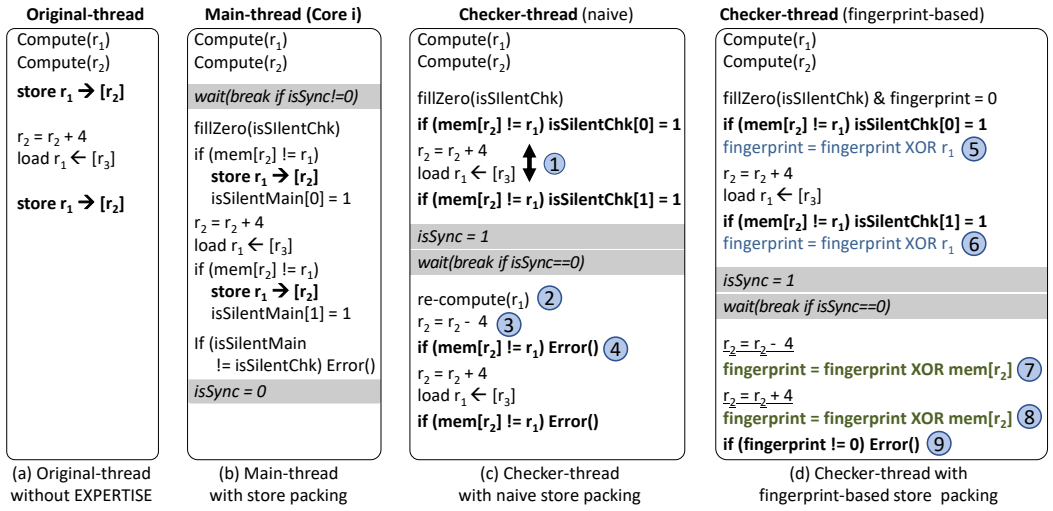


Fig. 8. While the naive store packing requires extra instructions to re-compute the data and address registers, the fingerprinting-based store packing needs to re-compute only the address registers.

the silent store checking of the second store (③) because it accesses the values of r_1 and r_2 that are not updated by the packed instructions (②).

4.2 Store packing optimization with fingerprinting-based load-back checking

Fig. 8 (a) shows an example code before applying EXPERTISE and Fig. 8 (b) illustrates the EXPERTISE-applied main-thread code with store packing optimization. Fig. 8 (c) illustrates the first-cut solution of checker-thread that applies EXPERTISE with store packing optimization. In the first-cut solution (Fig. 8 (c)), packed non-store instructions are executed only in the first part (marked as ①). Because both data and address operand registers of the first store are modified in the first part, their values should be restored before the execution of load-back checking of the first store in the second part (marked as ④). In the code example shown here, the address computations are easily revertible by performing reverse computations (`r2 = r2 - 4`, as marked with ③). For the data register (`r1`), the checker-thread needs to redo the part of the parallel computation required for data register calculation (denoted as `re-compute(r1)`, as marked with ②). Note that restoring data and address registers requires extra computations, but it might be impossible if previous register or memory data to restore data and address were overwritten. For the latter case, store packing should divide the pack, and therefore, the efficiency of store packing decreases.

To minimize redundant computations to apply store packing optimization in the presence of the EXPERTISE silent store checking, we propose a fingerprinting-based store packing optimization that can be combined with the EXPERTISE silent store checking. Fig. 8 (d) illustrates the checker-thread optimized by fingerprinting-based store packing. During the first part of the checker-thread for silent store checking, the checker-thread generates a data fingerprint by XORing all data registers for the silent store checkings (marked as ⑤ and ⑥). Note that the data of silent store checking and the one of the load-back checking should be equal in the absence of error if both checkings target the same store instruction. Therefore, XORing all loaded values in the load-back checking part should be equal to the fingerprint generated during silent store checking. Instead of generating another fingerprint and comparing it to the fingerprint of silent store checking part, we simply XOR all the loaded values in the load-back checking part with the previous fingerprint value (marked as

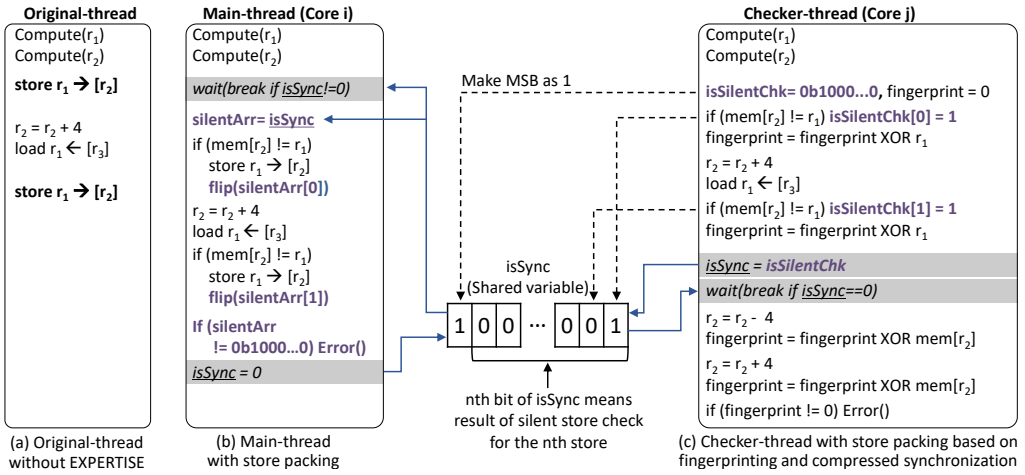


Fig. 9. EXPERTISE compresses the synchronization and results of silent store checkings on the checker-thread into one shared variable to minimize the number of inter-core operations.

⑦ and ⑧). Absence of the error, the final fingerprint value should be 0 because the checker-thread XORed all data values twice. Based on this observation, the checker-thread compares the XORed fingerprint with 0 to detect faults (marked as ⑨).

Consequently, the fingerprinting-based store packing does not need to revert changes in data registers and corresponding redundant computations for the load-back checking. Therefore, EXPERTISE only needs to revert and compute again for the address registers, as denoted by the underlined instruction in Fig. 8 (d). Furthermore, fingerprinting-based store packing has the potential to pack instructions that cannot be packed in basic store packing without silent store checking that is discussed in Section 4.1. Because all instructions inside of the basic store packing will be executed in `main-thread-first-then-checker-thread` fashion, store instructions cannot be packed together if there is `load/store-to-store memory dependency` in the basic store packing. Note that `load-/store-to-store memory dependency` means that there should be no store operation that comes after a memory operation (load or store) and access to the same memory location as previous memory instructions. This constraint is essential in the `main-thread-first-then-checker-thread` fashion because the preceding execution of store operation in the main-thread can overwrite the loaded value in the trailing load instruction in the checker-thread. On the other hand, all instructions in the fingerprinting-based store packing of EXPERTISE are executed in `checker-thread -first-then-main -thread` fashion, except for load-back checking and corresponding redundant instructions for the address. Therefore, store packing of EXPERTISE does not suffer from `load-to-store memory dependency` because load instruction in the checker-thread will be executed before the store instruction in the main-thread.

4.3 Applying compressed silent store checking

As discussed in Section 3.2, additional inter-core communication can induce another vulnerability if the communication process is not protected. In store packing optimization, the checker-thread should send the results of silent store checkings to the main-thread as many as the number of stores in the instruction pack. To eliminate additional vulnerabilities from the additional communication process for sending results of silent store checkings, we extend the idea of combining the synchronization variable and the result of silent store checking in Fig. 5 (b) to compress multiple results

of silent store checkings into one synchronization variable. Fig. 9 (a) shows an example original code, and Fig. 9 (b) and (c) show EXPERTISE-applied main- and checker-threads with compressed inter-core communications, respectively. The modified codes for the compressed communication compared to the main-thread and checker-thread without compressed communication (Fig. 8 (b) and (d)) are written in bold in Fig. 9 (b) and (c). The synchronization variable is underlined, and the instructions for inter-core communications are written in italics with gray boxes in Fig. 9 (b) and (c). In this example, `isSync` performs the roles of the synchronization variable and results of the multiple results of silent store checkings in the checker-thread. The checker-thread encodes the results of silent store checkings in the packed instructions into the bits in the temporal register `isSilentChk` and sends `isSilent` to the main-thread via `isSync`. Specifically, the nth bit of `isSyncSilent` indicates the result of the silent store checking (1 means silent, and 0 means not silent) for the nth store instruction in the pack, as shown in Fig. 9. However, the value of `isSilentChk` and `isSync` can be zero if all of the store instructions in the pack are not silent. This should be avoided because the main-thread waits when the value of `isSync` for communication is 0. Therefore, when the checker-thread sends the value of `isSilentChk` to the main-thread via `isSync`, it always makes the *MSB* (most significant bit) of `isSilentChk` as 1. Note that `isSilentChk` is not a variable for communication; therefore, this solution does not require an additional communication process compared to EXPERTISE without silent store checking.

After notification from the checker-thread, the main-thread copies the value of `isSync` from shared memory into the register (`silentArr` in Fig. 9). Instead of maintaining `isSilentMain` as in the previous example (Fig. 8 (b)) to accumulate the results of silent store checkings, this solution flips the nth bit of the `silentArr` if the nth store is silent. After executing all instructions in the pack, every bit of `silentArr` except MSB should be 0 if the results of the silent store checkings in the main-thread match the ones of the checker-thread. Therefore, the main-thread can detect the mismatch of silent store checkings by checking whether all bits of `isSilentChk` except MSB are 0.

While the compressing the results of silent store checkings to the synchronization variable does not requires additional communication for transferring the results of silent store checkings, it limits the number of store instructions that can be packed into one inter-core communication; If the size of `isSync` is n bits, The maximum number of packed store instructions is n-1. If there are more than n-1 stores to pack, the store packing optimization should split them into multiple packs.

4.4 Algorithm for the store packing optimization

Algorithm 1 shows the algorithm to extract instruction sets that store packing optimization of EXPERTISE can pack together from a basic block. This algorithm parses all instructions in the target block and start packing when it encounters the store instruction by *insidePack*. If the algorithm judges that it can not pack any more instructions, it terminates the current packing by enrolling the current pack with the instructions from the firstly store instruction to the lastly parsed one. Then, it initialize the variables and continues parsing until it parses all instructions in the target block. Note that there can be multiple instruction packs in one basic block.

The algorithm terminates the current packing if it meets the following conditions. i) The current instruction is a store instruction and it overwrites the memory address of the previous store instructions in the pack. ii) The number of store instructions exceeds the maximum stores as discussed in Section 4.3. iii) The current instruction is a load instruction and it accesses the memory address of the previous store instructions in the pack. iv) The current instruction is a jump or branch instruction. v) The current instruction updates a register that should be used as an address register of the previous store instructions in the pack, and this result is not revertible. vi) The current instruction is the last instruction of the basic block.

ALGORITHM 1: Pack Extraction Algorithm of Store Packing Optimization

```
Input: Original Program Basic Blocks.
Output: List of Packed Instructions.
insidePack = FALSE; firstPackedStore = NULL; lastPackedStore = NULL;
memoryAddrList = NULL; addrRegList = NULL; packList = NULL; numOfStores = 0;
for Each instruction Inst in Current Basic Block do
    if insidePack == FALSE then
        if isStore(Inst) then
            firstPackedStore = Inst; lastPackedStore = Inst; insidePack = TRUE;
            memoryAddrList.insert(memoryAddr(Inst)); addrRegList.insert(addrReg(Inst));
            terminatePacking = FALSE; numOfStores = 1;
    else
        if isStore(Inst) then
            if memoryAddr(Inst) is in memoryAddrList or numOfStores >= maximumStoresInPack then
                terminatePacking = TRUE;
            else
                memoryAddrList.insert(memoryAddr(Inst)); lastPackedStore = Inst; numOfStores += 1;
        else if isLoad(Inst) then
            if memoryAddr(Inst) is in addrList then
                terminatePacking = TRUE;
        else if isJumpOrBranch(Inst) then
            terminatePacking = TRUE;
        if Inst modifies x for x in addrRegList then
            if Inst is not revertible then
                terminatePacking = TRUE;
        if Inst is the last Instruction in the Basic Block then
            terminatePacking = TRUE;
        if terminatePacking then
            packList.insert(pack(from firstPackedStore to lastPackedStore));
            insidePack = FALSE; firstPackedStore = NULL; lastPackedStore = NULL;
            memoryAddrList = NULL; addrRegList = NULL; numOfStores = 0;
            if isStore(Inst) and Inst is not packed in the current pack then
                Repeat iteration with Inst again;
```

The presented store packing algorithm does not change the order of existing instructions. The elaborate code placement optimization that re-orders the instructions to minimize the number of non-store instructions in the store packs and places the synchronization instructions to the proper location can further reduce the performance overhead of EXPERTISE.

5 EXPERIMENTAL METHODOLOGY

5.1 Architecture and benchmark setup

To test the runtime performance and the fault coverage, we compiled nine programs from the MiBench benchmark suite [15][4] with the LLVM 4.0.0 compiler infrastructure [23] with the O3 optimization flag to minimize the number of store instructions [28]. For each program, we generated three binaries: **i) `ORG`:** original unprotected version of programs, **ii) `SRMT`:** This set of programs is protected by SRMT transformation [49], and **iii) `EXPERTISE`:** This set of binaries is protected by EXPERTISE transformation which includes silent store detection, store packing, fingerprinting-based error detection, and compressed inter-core communication between the main-thread and checker-thread introduced in this work. To apply the protection schemes, we modified the compiler to reserve registers before applying the protection schemes. EXPERTISE requires two reserved registers to hold the address of the synchronization variable (`isSync`) and the temporal register for checking (tmp in Fig. 2), and the silent store handling of EXPERTISE requires one more register to hold the result(s) of silent store checking(s). SRMT requires two registers for holding the address and

[4]We modified crc32 benchmark to call crc32buf, instead of crc32file that calls fread function for every iteration.

Table 2. gem5 simulator Configuration

Parameter	Value
CPU Model	ARMv7-A 32-bit dual-core in-order processor
Pipeline	Two issue/4-stage
# of FUs	2Int, 1Mul, 1Div, 1Float, 1Mem, 1Misc
L1 D/I-Cache	64KB (2-way) / 32KB (2-way), hit latency:2
L2 Cache	2MB (8-way), hit latency: 10

index of the circular queue that works as the buffer from the leading-thread to the trailing-thread, and one more register as a temporal register for checking (temp in Fig. 1). After generating the assembly files with the register reserving, the Python script generates the protected versions. All of the inter-core communications in SRMT and EXPERTISE are via the shared L2 cache.

In addition to the above three versions, we also generated two versions for the detailed fault coverage and performance evaluation. The first version is **EXPERTISE w/o silent store avoidance[44]**, which includes the load-back checking and store packing of EXPERTISE but does not include silent store checking. We exploit this version to evaluate the fault coverage improvement from the silent store checking and the additional runtime overhead for applying silent store checking. The second one, **EXPERTISE w/o store packing optimization**, excludes the store packing optimization to evaluate the performance improvement from the store packing optimization.

We evaluated the error detection capability of SRMT and EXPERTISE version of binaries by statistical microprocessor-wide fault injection experiments on gem5 μ-architectural cycle-accurate simulator [1]. Note that the results of such fault injection campaigns show the same trend as neutron beam testing experiments [4]. We configured a gem5 simulator similar to the cortex-A53 dual-core dual-issue microprocessor. Table 2 shows the simulated microprocessor parameters. Considering that we worked with unmodified standard library calls, we excluded them from all fault injection experiments and our simulation-based performance overhead estimation.

To measure the real-world runtime overhead of EXPERTISE-protected programs, we also run them on a real device (REVVL 5049W) equipped with a Cortex-a53 ARM v8 in-order microprocessor with a 32kb private data and instruction cache as well as 1024kB shared L2 cache. To execute the EXPERTISE-protected programs on the device, we inserted data memory barrier (dmb) instructions right before sending `isSync` value and right after escaping waiting. We executed each original and EXPERTISE-protected program on the device ten times and estimated the average runtime of 8 trials except for the maximum and minimum execution times. Since it is hard to exclude the execution time on the unmodified standard library functions on the real device setup, the estimated runtime on the device includes such functions, which are excluded for simulation-based experiments. In addition, we did not use the standard input of miBench [15] for real performance measurement since their fast execution on the device makes runtime estimation inaccurate. Instead, we increased the input data size up to several orders of magnitude.

5.2 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 the original, `SRMT`, `EXPERTISE w/o silent store avoidance`, and `EXPERTISE` versions of the programs. We injected errors into five main hardware components of each core, including the register file, fetch and decode stage pipeline registers, functional units[5] and load/store unit. For each hardware component, we inject 500 transient faults and 500 permanent stuck-at faults per version unless the hardware component was not sufficiently utilized by the program. As a result, our fault injection

[5]Functional units consist of two integer units, multiplier, floating-point unit, memory unit (read), and memory unit (write).

experiments injected up to 20,000 (10 components* 500 fault injections * 2 types of fault * 2 cores for multithreading) random processor-wide for each version of programs[6]. Overall, we performed 394,000 fault injection experiments, which yielded an error of less than 5% with a 95% confidence interval for each component per version of the program [27].

Transient Fault Injection: To inject transient faults in a target component, we first created a trace file per component, which included all instructions and the corresponding cycle in which they utilized the component as well as their corresponding values. Next, we randomly selected a bit b from the target component c and a fault injection cycle t from the trace file. Next, we started the simulation and whenever the execution reached cycle t, we modified the value (data) associated with b in c. Finally, we let the simulation run resume with the corrupted data until the program permanently terminates or the allowable simulation time (2× more than error-free simulation time). **Permanent Fault Injection:** Permanent fault injection is almost similar to transient fault injection, but we permanently alter all data that utilize the targeted component in such a way that the selected bit b in the target hardware component is always set to either zero or one.
Output Classification: We classified the output of each fault injection simulation run into: **1) Masked:** The program terminates normally, and the output is correct. **2) Failure/SDC:** The program operates and terminates normally, but the output is incorrect. **3) System-Level Symptoms:** This outcome occurs when the program encounters a fatal error, that is, segmentation faults, crashes, or timeout (simulation time reaches its limit).

Similar to [6, 8, 39], we considered the impact of runtime and hardware overhead of protected schemes on reliability estimation by multiplying the absolute number of SDCs by a scaled factor. The scaled factor should be composed of two components: hardware overhead (β) and performance overhead (α). Because all programs protected by `EXPERTISE` require the same amount of hardware overhead (1 extra core), we need to consider the hardware overhead component. However, considering the hardware component overhead by choosing β as 2 and injecting the same number of faults for both original and protected ones can be dangerous because two cores run different threads (master or checker), and their behavior can be different. As follows, instead of scaling the number of SDCs by $\beta = 2$, we injected the same number of faults to each core for protected versions. In other words, we injected 2× faults to the protected versions compared to the unprotected versions.

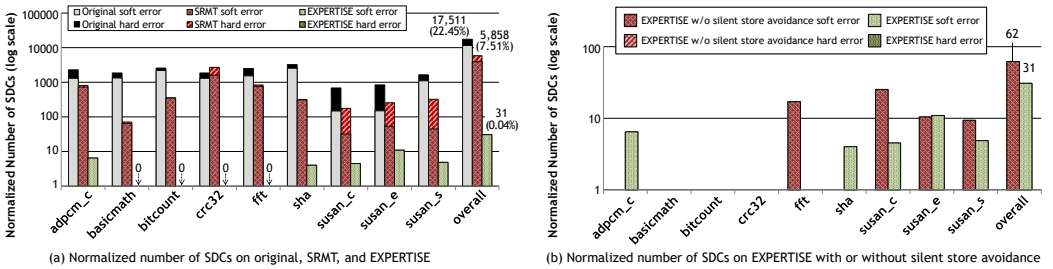
6 EXPERIMENTAL RESULTS

6.1 Fault coverage

Fig. 10 (a) shows the scaled number of SDC for ORG, SRMT, and EXPERTISE versions of the programs. In the Figure, Y-axis represents the normalized number of SDCs, and X-axis shows the benchmarks. Note that the rightmost set of bars (annotated by total) represents the sum of all scaled SDCs for all benchmarks. Fig. 10 (a) indicates that of the 78,000 fault injection experiments on the original version of programs, 17,511 (∼22.45%) lead to SDCs. SRMT-protected versions of programs reduced 5,858 scaled-SDCs (∼7.51%)[7]. On the other hand, the EXPERTISE-protected version of programs ends up with 31 scaled-SDCs, translating to more than 99.95% error coverage.

To analyze the effectiveness of the silent store checking and avoidance, we also injected the same number of soft and hard errors as the number of errors injected to EXPERTISE to the EXPERT w/o silent store avoidance versions. Fig. 10 (b) shows the scaled number of SDC for EXPERTISE w/o silent store avoidance (referred to as `EXPERTISE-silent`) and EXPERTISE. As shown in Figure Fig. 10 (b), applying silent store checking and avoidance to the load back checking of EXPERTISE

[6]Since an original program only utilizes one core, the number of injected faults is up to 10,000 for original versions
[7]While we injected 157,000 faults for SRMT and EXPERTISE-protected programs, we calculated the percentage of SDCs by dividing the number of faults for original ones, since additional faults are due to the area overhead



(a) Normalized number of SDCs on original, SRMT, and EXPERTISE

(b) Normalized number of SDCs on EXPERTISE with or without silent store avoidance

Fig. 10. (a) EXPERTISE transformation reduces the failure rate of SRMT[49] by around 188×. (b) The silent store avoidance of EXPERTISE can effectively reduce the vulnerability from the corruption of silent stores.

can reduce the total scaled number of SDC from 62 to 31. Specifically, EXPERTISE dramatically reduces the SDCs from the benchmarks that have frequent silent store instructions. For example, the ratio of silent stores in the benchmarks `susan_corners` and `fft` are 42.7% and 24.9%, and silent store checking of EXPERTISE reduces the number of SDCs from 25 to 5 and from 17 to 0, respectively. On the other hand, for the benchmarks such as `adpcm_c` and `sha`, the scaled numbers of SDCs are 0 for EXPERTISE w/o silent store avoidance while the numbers are not 0 for EXPERTISE. Note that the faults that induce SDCs in the EXPERTISE versions of those benchmarks can also cause SDCs for the EXPERTISE w/o silent store avoidance versions. Still, our randomly selected faults for the EXPERTISE w/o silent store avoidance did not include such faults.

While EXPERTISE transformation can effectively remove the failure cases on the address corruption of silent store operation, EXPERTISE still cannot detect very few soft errors on the fetch unit if they corrupt the operation code and alter any non-store operation to the store one. Specifically, the non-store to store alteration can lead to SDC if i) the corrupted instruction was originally silent, and as a result, the clean execution of the instruction does not alter any register state, or ii) the load instruction of one thread is converted to the store instruction before the other thread executes the corresponding load operation, and the latter one just receives the result of the altered store operation. These failures occur because EXPERTISE statically inserts the error detection codes in compile time for static store instructions, and therefore it cannot protect the altered store instructions that are not originally store instructions in compile time. Further, we observed one rare undetected control-flow error on EXPERTISE.

6.2 Runtime overhead

Fig. 11 shows our simulation-based execution time overhead estimation for `SRMT` and `EXPERTISE` transformations. On average, `SRMT` and `EXPERTISE` transformations increase the program's execution time by an average of ~3.81× and ~3.00×, respectively. Around 3× performance degradation may seem high at first glance. However, almost all similar works impose the same or even more performance overhead. For instance, Intel SRMT paper [49] reports ~4× and AMD GPU software-level redundant multithreading paper [14] shows ~6× performance degradation for inter-group RMT after applying different optimizations, which are similar to the evaluated performance degradation of SRMT in our experiments as shown in Fig. 11. Note that Intel SRMT technique suffers from many single-point-of-failures, and there is no error injection evaluation for AMD RMT approach. The main point of the results shown in Fig. 11 is that the runtime overhead of software-level redundant multithreading schemes varies significantly (up to ~6×) across different benchmarks. For instance, consider `adpcm_c` and `bitcount` applications with `EXPERTISE` protection overhead around 6.5× and 1.1×, respectively.

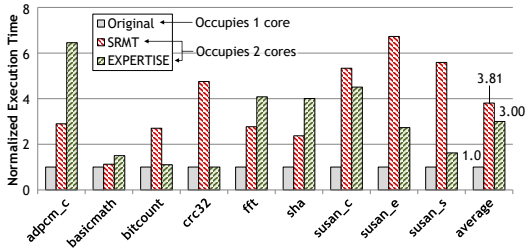


Fig. 11. EXPERTISE-protected programs run around 27% faster than SRMT-protected ones.

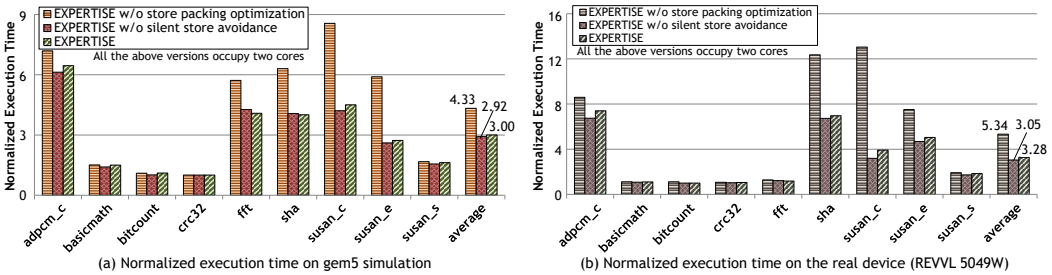


Fig. 12. On average, the store packing optimization speeds up the EXPERTISE-protected programs 44% and 44% on (a) gem5 simulation and (b) real device, respectively.

To verify the performance improvement by store packing optimization and the runtime overhead from the silent store checking, we measured the performance of EXPERTISE w/o store packing optimization and EXPERTISE w/o silent store avoidance versions. Figure 12 (a) shows our simulation-based execution time estimation for EXPERTISE w/o store packing optimization, EXPERTISE w/o silent store avoidance, and EXPERTISE. By applying store packing optimization, the runtime of EXPERTISE can be reduced by 4.33 to 3.00 on average, which is 27% faster than the average runtime of SRMT. While performance improvement of store packing optimization is significant in benchmarks `fft`, `sha`, `susan_c`, and `susan_e`, improvements from the other benchmarks are negligible. The magnitude of improvements from the store packing optimization is proportional to the reduced inter-core communication by the optimization. Fig. 13 shows the relationship between the speedup by store packing optimization and the corresponding reduced inter-core communication for the synchronization. For the benchmarks such as `fft`, `sha`, `susan_c`, and `susan_e`, store packing optimization reduces more than 50% communication and gets higher than 40% speedup. On the other hand, the impact of store packing optimization is insignificant for other benchmarks since the packing only reduces a small number of inter-core communications.

On the other hand, the silent store avoidance of EXPERTISE does not incur significant runtime overhead, as shown in Fig. 12 (a). The dominant source of runtime overhead is the inter-core communication to synchronize the main- and checker-threads. For a benchmark `fft`, EXPERTISE is slightly faster than the EXPERTISE w/o silent store avoidance since they suffer from the different data dependencies. EXPERTISE w/o silent store avoidance cannot pack two store instructions although they access the different addresses if the load instruction between them accesses the address of the latter store instruction. On the other hand, EXPERTISE cannot pack two store instructions although they access the different addresses if the load instruction between them accesses the address of the former store instruction. In Fig. 12 (a), the store packing of EXPERTISE and EXPERTISE w/o silent store avoidance for the benchmarks `adpcm_c`, `basicmath`, `bitcount`,

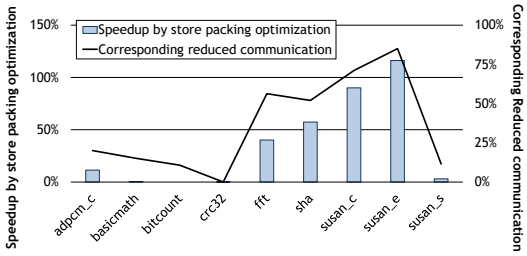


Fig. 13. The performance improvement of store packing optimization is proportional to the reduced number of inter-core communication by the optimization.

and `crc` are identical, so we can assume the runtime overhead due to the additional instructions for the silent store avoidance.

Fig. 12 (b) shows the results of runtime measurement on the real device (REVVL 5049W). The tendencies of the runtime overhead on the device (Fig. 12 (b)) are similar to the ones on the gem5 simulation (Fig. 12 (a)), except for the benchmark `fft`. Around 93% runtime of `fft` is spent on the library functions that EXPERTISE does not modify, and the results on Fig. 12 (b) did not exclude the execution time on such functions for the measurement. The speedup of the store packing optimization is 63% on the device, while it is 44% on the simulator. We assume that the communication cost on the real device is more expensive than the simulation, and therefore the impact of the optimization is higher correspondingly.

7 LIMITATIONS

7.1 EXPERTISE on multithreaded applications

EXPERTISE should be carefully applied when the application accesses shared memory location since main-thread and checker-thread access memory location at different points of time. At first, both threads execute original load instructions in parallel. Secondly, before main-thread executes the store instruction, checker-thread accesses the target location of the store for silent store checkings, and then main-thread accesses it after checker-thread notifies main-thread. Thirdly, after main-thread completes the store instruction, checker-thread should access the target location for load-back checking. If another thread changes the data in the target location between the access of two threads, EXPERTISE will detect a mismatch between two threads and consider it an error.

7.2 Circular wait due to the faults on synchronization variable

EXPERTISE utilizes one synchronization variable `isSync`, and the faults that corrupt the value of `isSync` can induce circular wait between main- and checker-threads. Since `isSync` locates in the shared memory that is protected by parity or ECC, the corruption of `isSync` can only occur if i) a soft or hard error corrupts the store instruction to `isSync` and ii) a hard error repeatedly affect the load instruction from `isSync`. A store instruction to `isSync` is responsible for waking up another thread and transferring the results of silent store checkings. Therefore, if a fault corrupts such store instruction that another thread cannot wake up, both threads will wait for each other, and the application cannot continue the execution. A hard error on the load-related bits can also incur such circular wait if the load instruction from `isSync` for the waiting another thread always loads the false value so that the thread cannot awake[8]. Note that these faults on `isSync` cannot result in the SDCs, and additional solutions such as watchdog can detect them.

[8]Since EXPERTISE inserts load instructions from isSync for waiting and utilizes the loaded value for silent store checking, a single failure of a load instruction from isSync only results either waiting one more iteration or detectable error.

7.3 Protecting system calls

Experiments in Section 5 and 6 excluded fault injection on the standard library functions that our compiler cannot access and modify. Most of the instructions in these functions can be protected by EXPERTISE if the compiler can access them, except system calls that should not be repeated on both threads. Applying EXPERTISE to these system calls requires either kernel modification or the addition of custom system calls for main- and checker-threads. Otherwise, system call emulation [41] can be applied to EXPERTISE; non-repeatable system calls of main-thread are executed, and ones of checker-thread are emulated. Suppose the system call emulation is not applicable. In that case, the inputs of the non-repeatable system call should be compared before the execution, and the results should be distributed from the thread that executes it to another thread.

8 CONCLUSIONS

We present a software-controlled redundant multithreading scheme, EXPERTISE, to protect applications against hardware transient and permanent errors. EXPERTISE improves the hardware-level reliability based on the post-store error detection strategy and ensures that all writes to the memory are executed as expected. Because the post-store load-back checking is vulnerable to address corruption in silent stores, our EXPERTISE solution avoids the execution of silent stores by dynamically detecting the silent store in the main-thread and checker-thread. Our solution also reduces the number of thread synchronization points by applying store packing optimization, which packs multiple store instructions into one synchronization. Furthermore, our solution relieves register conflicts induced from store packing optimization with a silent store checking, using fingerprinting-based load-back checking. Finally, our solution compresses inter-core communications for redundant silent store checkings to avoid additional vulnerabilities from the additional unprotected communication process. Our extensive fault injection experiments show that EXPERTISE outperforms state-of-the-art redundant multithreading techniques in terms of both reliability and performance.

ACKNOWLEDGMENTS

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), Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korean government (MSIT) (No. 2021-0-00155, Context and Activity Analysis-based Solution for Safe Childcare), National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. RS-2022-00165225), and Samsung Electronics Co., Ltd. (FOUNDRY-202108DD007F).

REFERENCES

[1] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The gem5 simulator. *ACM SIGARCH Computer Architecture News* 39, 2 (2011), 1–7.

[2] Matthew Bohman, Benjamin James, Michael J Wirthlin, Heather Quinn, and Jeffrey Goeders. 2018. Microcontroller compiler-assisted software fault tolerance. *IEEE Transactions on Nuclear Science* 66, 1 (2018), 223–232.

[3] Shekhar Borkar. 2005. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *Ieee Micro* 25, 6 (2005), 10–16.

[4] Athanasios Chatzidimitriou, Pablo Bodmann, George Papadimitriou, Dimitris Gizopoulos, and Paolo Rech. 2019. Demystifying soft error assessment strategies on arm cpus: Microarchitectural fault injection vs. neutron beam experiments. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 26–38.

[5] Lawrence T Clark, Dan W Patterson, Chandarasekaran Ramamurthy, and Keith E Holbert. 2016. An embedded microprocessor radiation hardened by microarchitecture and circuits. *IEEE Trans. Comput.* 65, 2 (2016), 382–395.

[6] Moslem Didehban, Sai Ram Dheeraj Lokam, and Aviral Shrivastava. 2017. InCheck: An in-application recovery scheme for soft errors. In *Design Automation Conference (DAC), 2017 54th ACM/EDAC/IEEE*. IEEE.

[7] Moslem Didehban and Aviral Shrivastava. 2016. nZDC: A compiler technique for near Zero silent Data Corruption. In *Proceedings of the The 53st Annual Design Automation Conference on Design Automation Conference*. ACM, 1–6.

[8] Moslem Didehban and Aviral Shrivastava. 2018. A Compiler Technique for Processor-Wide Protection From Soft Errors in Multithreaded Environments. *IEEE Transactions on Reliability* 67, 1 (2018), 249–263.

[9] Moslem Didehban, Aviral Shrivastava, and Sai Ram Dheeraj Lokam. 2017. NEMESIS: A software approach for computing in presence of soft errors. In *Computer-Aided Design (ICCAD), 2017 IEEE/ACM International Conference on*. IEEE, 297–304.

[10] Chad Farnsworth, Lawrence T Clark, Anudeep R Gogulamudi, Vinay Vashishtha, and Aditya Gujja. 2016. A soft-error mitigated microprocessor with software controlled error reporting and recovery. *IEEE Transactions on Nuclear Science* 63, 4 (2016), 2241–2249.

[11] Christof Fetzer, Ute Schiffel, and Martin Süßkraut. 2009. AN-encoding compiler: Building safety-critical systems with commodity hardware. In *International Conference on Computer Safety, Reliability, and Security*. Springer.

[12] J Michael Galey, Ruth E Norby, and J Paul Roth. 1964. Techniques for the diagnosis of switching circuit failures. *IEEE Transactions on Communication and Electronics* 83, 74 (1964), 509–514.

[13] Mohamed Gomaa, Chad Scarbrough, TN Vijaykumar, and Irith Pomeranz. 2003. Transient-fault recovery for chip multiprocessors. In *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*. IEEE, 98–109.

[14] Manish Gupta, Daniel Lowell, John Kalamatianos, Steven Raasch, Vilas Sridharan, Dean Tullsen, and Rajesh Gupta. 2017. Compiler techniques to reduce the synchronization overhead of gpu redundant multithreading. In *Design Automation Conference (DAC), 2017 54th ACM/EDAC/IEEE*. IEEE.

[15] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*. IEEE, 3–14.

[16] Saurabh Hukerikar, Keita Teranishi, Pedro C Diniz, and Robert F Lucas. 2018. Redthreads: An interface for application-level fault detection/correction through adaptive redundant multithreading. *International Journal of Parallel Programming* 46, 2 (2018), 225–251.

[17] Xabier Iturbe, Balaji Venu, Emre Ozer, and Shidhartha Das. 2016. A Triple Core Lock-Step (TCLS) ARM® Cortex®-R5 Processor for Safety-Critical and Ultra-Reliable Applications. In *Dependable Systems and Networks Workshop, 2016 46th Annual IEEE/IFIP International Conference on*. IEEE, 246–249.

[18] Benjamin James, Heather Quinn, Michael Wirthlin, and Jeffrey Goeders. 2019. Applying Compiler-Automated Software Fault Tolerance to Multiple Processor Platforms. *IEEE Transactions on Nuclear Science* (2019).

[19] J Karlsson and P Lidén. 1990. *Transient fault effects in the MC6809E 8-bit microprocessor: A comparison of results of physical and simulated fault injection experiments*. Technical Report. Technical Report 96, Department of Computer Engineering, Chalmers University of Technology, Göteborg, Sweden.

[20] Daya Shanker Khudia, Griffin Wright, and Scott Mahlke. 2012. Efficient soft error protection for commodity embedded microprocessors using profile information. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 99–108.

[21] Dmitrii Kuvaiskii and Christof Fetzer. 2015. Delta-encoding: Practical Encoded Processing. In *Proceedings of The 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2015)* (Rio de Janeiro, Brazil). IEEE Computer Society.

[22] Christopher LaFrieda, Engin Ipek, Jose F Martinez, and Rajit Manohar. 2007. Utilizing dynamically coupled cores to form a resilient chip multiprocessor. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*. IEEE, 317–326.

[23] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*. IEEE.

[24] Kyoungwoo Lee, Aviral Shrivastava, Minyoung Kim, Nikil Dutt, and Nalini Venkatasubramanian. 2008. Mitigating the impact of hardware defects on multimedia applications: a cross-layer approach. In *Proceedings of the 16th ACM international conference on Multimedia*. 319–328.

[25] Kevin M Lepak, Gordon B Bell, and Mikko H Lipasti. 2001. Silent stores and store value locality. *IEEE Trans. Comput.* 50, 11 (2001), 1174–1190.

[26] Kevin M Lepak and Mikko H Lipasti. 2000. *On the value locality of store instructions*. Vol. 28. ACM.

[27] Régis Leveugle, A Calvez, Paolo Maistri, and Pierre Vanhauwaert. 2009. Statistical fault injection: Quantified error and confidence. In *Proceedings of the Conference on Design, Automation and Test in Europe*. European Design and Automation Association.

[28] Guilherme E Medeiros, Felis T Bortolon, Ricardo Reis, and Luciano Ost. 2018. Evaluation of compiler optimization flags effects on soft error resiliency. In *2018 31st Symposium on Integrated Circuits and Systems Design (SBCCI)*. IEEE, 1–6.

[29] Konstantina Mitropoulou, Vasileios Porpodas, and Timothy M Jones. 2016. COMET: communication-optimised multi-threaded error-detection technique. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. ACM.

[30] Shubhendu S Mukherjee, Michael Kontz, and Steven K Reinhardt. 2002. Detailed design and evaluation of redundant multi-threading alternatives. In *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on*. IEEE, 99–110.

[31] Nahmsuk Oh, Subhasish Mitra, and Edward J McCluskey. 2002. ED4I: error detection by diverse data and duplicated instructions. *IEEE Trans. Comput.* 51, 2 (2002).

[32] Nahmsuk Oh, Philip P Shirvani, and Edward J McCluskey. 2002. Error detection by duplicated instructions in super-scalar processors. *IEEE Transactions on Reliability* 51, 1 (2002).

[33] Joakim Ohlsson, Marcus Rimen, and Ulf Gunneflo. 1992. A Study of the Effects of Transient Fault Injection into a 32-bit RISC with Built-in Watchdog.. In *FTCS*. 316–325.

[34] Steven K Reinhardt and Shubhendu S Mukherjee. 2000. *Transient fault detection via simultaneous multithreading*. Vol. 28. ACM.

[35] George A Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I August. 2005. SWIFT: Software Implemented Fault Tolerance. In *Proceedings of the international symposium on Code generation and optimization*. IEEE Computer Society, 243–254.

[36] Eric Rotenberg. 1999. AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. In *Fault-Tolerant Computing, 1999. Digest of Papers. Twenty-Ninth Annual International Symposium on*. IEEE, 84–91.

[37] Ute Schiffel. 2010. Hardware error detection using AN-codes. (2010).

[38] Ute Schiffel, André Schmitt, Martin Süßkraut, and Christof Fetzer. 2010. ANB-and ANBDmem-encoding: detecting hardware errors in software. In *International Conference on Computer Safety, Reliability, and Security*. Springer, 169–182.

[39] Horst Schirmeier, Christoph Borchert, and Olaf Spinczyk. 2015. Avoiding pitfalls in fault-injection based comparison of program susceptibility to soft errors. In *Dependable Systems and Networks (DSN), 2015 45th Annual IEEE/IFIP International Conference on*. IEEE.

[40] Muhammad Shafique, Siddharth Garg, Jörg Henkel, and Diana Marculescu. 2014. The EDA challenges in the dark silicon era: Temperature, reliability, and variability perspectives. In *Proceedings of the 51st Annual Design Automation Conference*. ACM.

[41] Alex Shye, Joseph Blomstedt, Tipp Moseley, Vijay Janapa Reddi, and Daniel A Connors. 2009. PLR: A software approach to transient fault tolerance for multicore architectures. *IEEE Transactions on Dependable and Secure Computing* 6, 2 (2009), 135–148.

[42] Jared C Smolens, Brian T Gold, Babak Falsafi, and James C Hoe. 2006. Reunion: Complexity-effective multicore redundancy. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 223–234.

[43] Jared C Smolens, Brian T Gold, Jangwoo Kim, Babak Falsafi, James C Hoe, and Andreas G Nowatzyk. 2004. Fingerprinting: Bounding soft-error detection latency and bandwidth. *ACM SIGOPS Operating Systems Review* 38, 5 (2004), 224–234.

[44] Hwisoo So, Moslem Didehban, Yohan Ko, Aviral Shrivastava, and Kyoungwoo Lee. 2018. EXPERT: Effective and flexible error protection by redundant multithreading. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2018*. IEEE, 533–538.

[45] Zhenyu Sun, Xiuyuan Bi, and Hai Li. 2012. Process variation aware data management for STT-RAM cache design. In *Proceedings of the 2012 ACM/IEEE international symposium on Low power electronics and design*. 179–184.

[46] TN Vijaykumar, Irith Pomeranz, and Karl Cheng. 2002. Transient-fault recovery using simultaneous multithreading. In *ACM SIGARCH Computer Architecture News*, Vol. 30. IEEE Computer Society, 87–98.

[47] Reinhard von Hanxleden and Ken Kennedy. 1994. Give-N-Take—A balanced code placement framework. *ACM SIGPLAN Notices* 29, 6 (1994), 107–120.

[48] Jack Wadden, Alexander Lyashevsky, Sudhanva Gurumurthi, Vilas Sridharan, and Kevin Skadron. 2014. Real-world design and evaluation of compiler-managed GPU redundant multithreading. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*. IEEE.

[49] Cheng Wang, Ho-seop Kim, Youfeng Wu, and Victor Ying. 2007. Compiler-managed software-based redundant multi-threading for transient fault detection. In *Proceedings of the International Symposium on Code Generation and Optimization*. IEEE Computer Society.

[50] Yi-Ping You, Chung-Wen Huang, and Jenq Kuen Lee. 2007. Compilation for compact power-gating controls. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 12, 4 (2007), 51–es.

[51] Yun Zhang, Jae W Lee, Nick P Johnson, and David I August. 2012. DAFT: Decoupled acyclic fault tolerance. *International Journal of Parallel Programming* 40, 1 (2012), 118–140.