EXPERT: Effective and Flexible Error Protection by Redundant Multithreading

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

***Abstract*—Resiliency is a first-order design concern in modern microprocessor design. Compiler-level Redundant MultiThreading (RMT) schemes are promising because of their capability to detect the manifestation of hardware transient and permanent faults. In this work, we propose EXPERT, a compiler-level RMT scheme which can detect the manifestation of hardware faults in all hardware components. EXPERT transformation generates a checker thread for program main execution thread. These redundant threads execute simultaneously on two physically different cores of a multi-core processor. They perform mostly same computations, however, after each memory write operation 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, execution continues. Otherwise, the error flag will be raised. Our processor-wide statistical transient and permanent fault injection experiments show that EXPERT error coverage is ∼65× better than the state-of-the-art scheme.**

I. Introduction

The ever-increasing use of computer-based systems has made hardware unreliability as an important microprocessor design concern. Main sources of hardware unreliability are a) transient faults like temporarily bit flip errors – caused by high-energy particles, crosstalk, voltage violations, and other electromagnetic interference, and b) permanent faults caused by process variation, thermal stress, or oxide wear-out[1]. Redundancy is the most efficient strategy to detect the manifestation of hardware errors. Researchers have proposed redundancy-based error mitigation schemes in different layers of system design/implementation stack, ranging from application-level to circuit-level schemes. Among all of the proposed schemes, compiler level approaches are particularly interesting because of their inter/intra application flexible protection. In other words, the protection offered by compiler-level error detection schemes can be tuned based on the application criticality, i.e., error detection can be turned on for critical applications/tasks in mixed-critical environments, or even just for error-sensitive segment of an application execution.

Existing redundancy-based compiler-level error detection approaches can be divided into i) in-thread replication and ii) redundant multithreading (RMT) schemes based on the granularity of the replication units. In in-thread error detection schemes (SWIFT[1], nZDC[2], Clover[3], InCheck[4] and NEMESIS[5]), the replication unit is assembly instructions inside an execution thread while in RMT schemes (SRMT[6], DAFT[7] and COMET [8]) the replication unit is the whole execution thread. Since redundant instructions in in-thread error detection schemes share the underlying microprocessor components, they fail to detect hardware permanent faults.

The key idea behind existing thread-level error detection schemes is to create two copies of application main thread, so called leading and trailing threads, and execute them in parallel. During the execution, leading thread sends critical data (like the register operands values of shared memory write operations) to trailing thread for error detection. The trailing thread receives critical values and checks them against its own redundantly computed ones. If there is no mismatch, the leading thread proceeds and submits the results, i.e., writing data to the shared memory, otherwise the error flag will be raised. Existing RMT schemes [6, 8, 7] have been considered as effective solutions for hardware unreliability and researchers enhanced their applicability to HPC [9] and even GPUs [10] domains. However, our error coverage analysis (explained in section II-B) reveals severe protection holes in state-of-the-art RMT schemes. We realized 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 just computational/arithmetical operations of a program.

In this work, we present EXPERT, a compiler-level fault detection scheme which provides microprocessor-wide transient and permanent fault detection from fail-continue faults (the computations are erroneous, but execution continues normally). The main idea of EXPERT is to run two slightly different versions of an application thread, named Main and Checker threads, on physically different cores of a multicore processor. Main thread performs all programs instructions and updates the memory state. Checker thread, on the other hand, performs all computations and memory read operations redundantly, but executes no memory write operation. Instead it verifies the correctness of main thread computations as well as write operations by loading back the main thread written value from the memory and checks it against its own locally computed value. EXPERT transformation does not suffer from vulnerable input replication process and frequent unprotected

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

978-3-9819263-0-9/DATE18/©2018 EDAA

memory write operations. However, it requires thread synchronization on memory write operations to guarantee coherence memory accesses for redundant threads and facilitates check-after-write error detection policy. To reduce the number of such synchronization points, we propose memory operation packing optimization. This compile-time optimization identifies independent successive memory (read/write) operations an groups them together. Then, rather than synchronizing threads on all memory write operations inside a pack, we just need to synchronize them one time for the whole pack.

To evaluate the effectiveness of EXPERT, we performed microprocessor-wide statistical transient and permanent fault injection experiments on a μ-architecturally simulated ARM-cortex A53 like dual-core microprocessor. The results show that on an average EXPERT transformation can achieve around 65$\times$ better error detection than state-of-the-art.

II. Background and Motivation

A. Why redundant multithreading for error detection?

There are two main strategies for hardware permanent and transient fault detection: one is to utilize some data encoding/decoding scheme[11, 12], and second is to execute redundant computations on two physically separate cores/processors[6, 10, 13]. The software implementation of the former comes with a significant performance overhead (6$\times$ to more than 60$\times$ as reported in [12]) and suffers from limited fault detection capability. In second strategy, two functionally identical versions of an application are executed on two separate cores, and their results will be checked for error detection. The frequency and position of error checking operations are crucial and determines error detection properties like performance degradation, error detection capability, and error detection latency – the time between the error occurrence and the error detection. An ideal error detection scheme should be able to detect the manifestation of all errors quickly after their occurrence with the minimum performance overhead. Short error detection latency is important especially in checkpoint/rollback systems because it affects recovery latency and the amount of waste computations. Checking the final outputs of redundant executions can provide high error coverage with minimum error detection overhead in some cases. However, in general, due to programs dynamicity (interaction with a user and other applications during the execution), such strategy is not practical in many cases.

Positioning error detectors at the boundary of program I/O points seems reasonable because it prevents errors from propagating to the outside of the sphere of protection. PLR or Process-Level Replication [13] proposes dynamic process replication and performs error checking for system call arguments. As mentioned in [2], PLR suffers from undetected errors in cases that the system call arguments are pointers. The reason is that even if the redundant computed pointers are correct, the data which is stored in the memory may not be. Nevertheless, in cases that system call arguments are not pointers, PLR can provide effective error detection. However, since existing process-level redundancy schemes like PLR [13] and [14] (runtime overhead improved version of PLR) are based on run-time whole process replication, they cannot provide intra-application flexible protection.

Fine-grained error detection on all program memory operations has been used in several state-of-the-art soft error detection schemes like nZDC [2]. The main advantage of such schemes is flexible error protection, low error detection latency, and microprocessor-wide protection. nZDC provides high-level of soft error protection by replicating instructions inside a thread and placing error detectors after the memory write instructions. Although such scheme eliminates the chance of latent errors in checkpointing/rollback systems and provides fine-grained protection controllability, it fails to detect the manifestation of hard errors. Thread-level redundancy based schemes can provide flexible soft and hard error protection. Software-based redundant multithreading (SRMT) scheme proposed in [6] is the first software-only fine-grained multithread scheme which can detect both soft and hard errors.

SRMT considers all shared memory read accesses and return values of system calls as inputs to the redundant threads. SRMT simply copies such values from the leading thread to the trailing one and duplicates computations. SRMT considers non-stack memory read/write accesses and system calls as output operations and checks the redundant computed register operands of such operations. To provide fail-stop failure mode, SRMT leading thread does not submit volatile store operations till the trailing thread verifies the value of their register operands. Note that memory read operations are treated as both SRMT sphere-of-protection input and output operations – the address register operand should be checked, and the loaded value from memory should be copied from the leading thread to the trailing one. To facilitate the data communication between leading and trailing threads, SRMT proposes a circular software queue.

Figure 1 shows SRMT transformation for a simple snippet of code. For the `load` instruction, the leading thread first sends the value of load address register operand (`r4`) to the trailing thread and then executes memory read operations. The `sendBuf(r4)` function shown in the figure is responsible for sending the value of load address register operand to the trailing thread. Then, once leading thread receives requested data from memory, it sends the loaded data for the trailing thread by calling `sendBuf(r0)` function. Trailing thread, on the other hand, receives data from the buffer by calling `recvBuf()` and compares that against its own locally computed value for the load address register operand (`r4`). If they match, trailing thread reads the next value from the buffer and places that in the corresponding register(`r0`). Both threads execute computational instructions redundantly. Once the leading thread reaches a memory write operations, it sends the values of data and address registers of store (`r0` and `r4`) to 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.

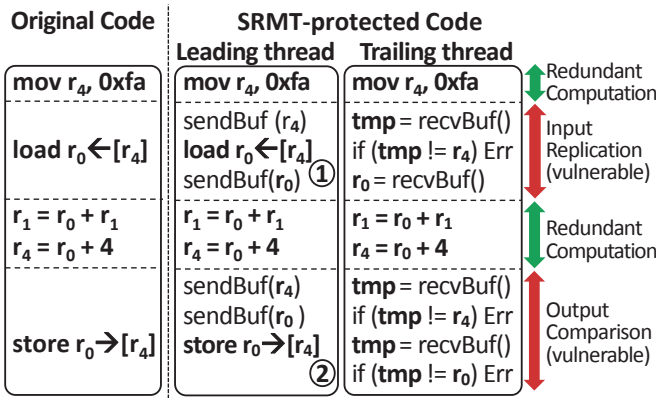

Fig. 1: SRMT transformation duplicates original program execution thread. Only leading thread performs memory operations. Trailing thread validates the correctness of memory instruction register operands while the execution of memory operations themselves remains unprotected.

B. Limitations of SRMT

Our detailed analysis of fault coverage of SRMT transformation reveals two major vulnerability windows:

1) Vulnerable input replication process. In redundancy-based error detection strategies, input replication is the process of providing same inputs for redundant versions of computations. Input replication process can be considered as a single-point-of-failures in redundancy-based error detection schemes. If any error affects the input data before (or during) input replication, the error remains undetected. That is because both redundant executions start their computations with same wrong data and will produce same wrong output. One of the main disadvantages of SRMT transformation is frequent input replication operations which take place after all shared memory read accesses and system call operations. In SRMT transformation, leading thread performs shared memory read and system calls operations and sends the results (the loaded value from the memory or system call return values) to the trailing thread. Therefore, if any error happens during the execution of such single-instance operations, will propagate to the trailing thread and can lead to a user-visible failure. For instance, assume an error which permutes effective address calculation of the leading thread load instruction shown in figure 1 (marked as ①). Because of such error, the wrong value will be loaded into `load` instruction destination register (`r0`). Then the leading thread sends that faulty value to the trailing thread, and both threads continue their execution with same wrong value. Faults on pipeline stage register bits while processing load operations, memory address generation units or load/store units are examples of errors which may remain undetected in SRMT scheme because of frequent input replication process.

2) Vulnerable output comparison process. Output comparison is defined as the process of checking the results of redundant computations for error detection and committing final results if there is no discrepancy. In SRMT transformation, all errors which occur after results checking and during final result submitting process (marked as ② in figure 1) will directly affect the memory write operation and can cause a failure. Examples are errors in pipeline data path registers while processing the store instructions, store effective address functional unit, store buffers and even in store register operands (after sending their data to the trailing thread and before being read for memory write operation). Moreover, since store operations can stay in microprocessor store buffer for a long time, they have considerably higher chance of soft error exposure time and chance to be corrupted.

We particularly concentrate on SRMT scheme in this work. However, all existing software-only RMT schemes suffer from the above mentioned vulnerable intervals. For instance, DAFT [7] technique improved the performance overhead of SRMT by applying 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. However, not only DAFT suffer from all SRMT protection holes, but its error coverage is limited to soft errors. COMET [8] 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. Researches [15, 10] applied SRMT error detection strategy to GPUs. Furthermore, RedThreads [9] takes advantages of SRMT flexible error detection and accomplishes programmer-tunable protection by providing programming-language support for applying partial SRMT in HPC applications. Overall, since all existing software-level redundant multithreaded schemes mainly try to improve the performance overhead of SRMT, they suffer from SRMT protection holes.

III. Our Approach

In this section, we present EXPERT, a compiler-level RMT approach which eliminates input replication and output comparison vulnerability windows of the state-of-the-art schemes. The main design goal of EXPERT is to provide processor-wide transient and permanent fault detection. More specifically, we consider single bit flip model for transient faults and single stuck at fault model for permanent faults. EXPERT targets single-threaded applications running on a chip multicore processor in which memory subsystem (excluding cores private caches) is protected by ECC. EXPERT transformation assigns a checker thread for main application thread. Checker thread will be executed on a different core than the one executing main execution thread. The key idea here is to orchestrate the main and checker threads in such way that after each memory write operation committed by the main thread, the checker loads the written value from memory back and checks that against its own locally computed value. The major features of EXPERT are:

1) EXPERT transformation eliminates single-point-of-failures of input replication process. As mentioned before, input replication process introduces a protection hole in existing RMT schemes. EXPERT eliminates such vulnerabilities by

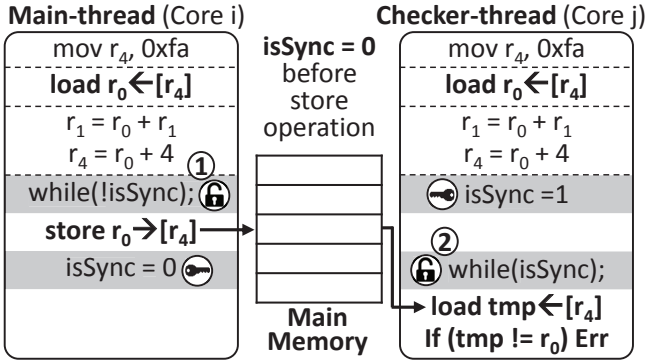

Fig. 2: EXPERT transformation runs two copies of a program thread and synchronize them on store operations. Main thread performs store and the checker thread verifies the correct execution of store by loading the written value from memory and check it against its own locally computed one.

adopting relaxed memory read instruction duplication strategy. Figure 2 illustrates EXPERT transformation. As shown in figure 2, EXPERT transformation replicates memory read instructions as well as computational instructions. The main challenge here is how to provide input replication coherency (how to guarantee that both redundant threads will receive same data from the shared memory). For instance, consider a case that main thread reads some value from memory, performs computations and store the updated data back to the memory. Later on, once the checker thread performs the redundant read operation, it will receive a different value from what the checker thread was received which eventually lead to false error detection. Therefore, we need a mechanism to make sure that both threads have performed their previous memory read operations before writing to the memory. EXPERTS uses a shared variable (`isSync`) and the busy-waiting mechanism to provide required ordering between redundant threads executions. The operations required for coherent input replication is 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 input coherency problem, we choice to put them right before store operations because it will give us the minimum number of synchronization points – Generally, number of write operations is lesser than read operations. The only exception for that is load operations from volatile memory locations. Volatile memory locations are the ones that may get updated by write operations committed from the outside of the application. Memory-mapped IO addresses are examples of volatile memory locations. We synchronize main and checker threads right before volatile memory read accesses, then both leading and checking thread issue three redundant versions of the volatile load instruction and perform 2-of-3 majority voting operation between the results and continue their executions. The reason for the in-thread triplication and voting operations for volatile memory accesses is to provide a coherent input replication process even for frequently updated volatile memory accesses. If an error affects the execution of replicated loads and does not get recovered by majority voting operation, the error will be manifest itself in the successive error checking operations.

2) EXPERT transformation does not suffer from vulnerable output comparison process. Unlike the existing software-level multithreaded scheme which suffers from very fragile output comparison/delivery process, EXPERT transformation verifies the correctness of computations and the output delivery process. EXPERT accomplishes such comprehensive error checking by loading the main thread written data from memory in checking the thread and compares that against the redundant computed version of data. The challenge here is how to make sure that the checking operation will take place after the write operation. EXPERT addresses this problem by inserting a memory barrier after memory write operations (marked as ② in figure 2). Once the main thread performs a memory write operation, it clears `isSync` flag. For checker thread, a clear flag means that it can proceed and verify the result (computations and execution of memory write operation) of the main thread. Checker thread accomplishes that by loading the main thread written value from the memory and checks it against its own locally computed value. Note that in EXPERT error detection strategy, errors altering effective address of the store instructions will also be detected. If because of any reason the main thread updates a wrong location of memory, the checker-thread reads the data from the right memory location and detects a mismatch.

IV. Performance optimization

As previous works [10, 8] mentioned, the major performance bottleneck in software-level RMT techniques is coming from synchronization overhead. Particularity, since EXPERT scheme forces threads to be synchronized on all memory write operations, it suffers from considerably more synchronization overhead. To reduce the number of synchronization points, we propose a memory operation packing optimization which in we consider all non-conflict successive memory operations as one operation and just requires one synchronization point. Figure 3 illustrates the main idea of memory operations packing optimization. In the figure, memory operation marked as ②, ③ and ④ are successive and independent memory operations – their memory addresses are known to be different at compile time. Therefore, we can pack them together and synchronize threads just once for the whole pack. However, since there is a memory dependency between last store instruction (marked as ⑤) and the third load operation (marked as ③), the main and checker threads have to be synchronized before the last store instruction. This is necessary to avoid false alarms and providing coherence input replication.

V. Experimental results

Compilation and simulation framework: We have implemented both EXPERT and SRMT transformations as late backend passes in LLVM3.7 compiler infrastructure. We have compiled nine applications from MiBench benchmark suite with -O3 compiler optimization flag. Note that since we did not modify the standard library call source code, we excluded

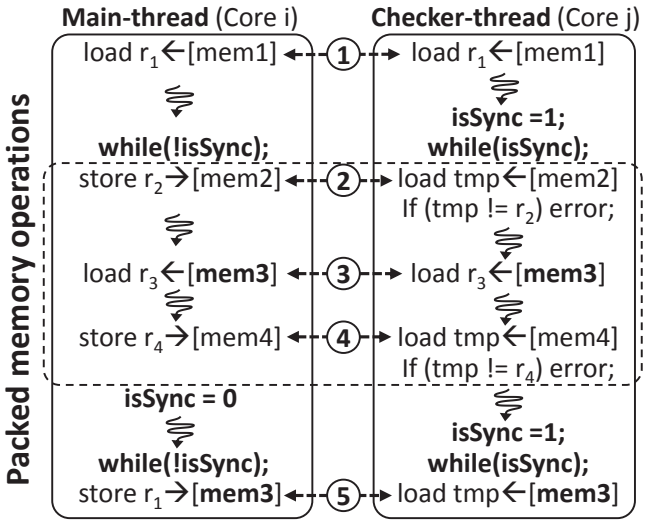

Fig. 3: EXPERT replicated thread synchronization points can be reduced by packing successive independent memory operations.

them from fault injection experiments and performance overhead estimation results shown in this work. We used Gem5 [16] μ-architectural level simulator. The simulator was run in syscall emulation mode and modeled ARMv7-a profile on a 32-bit two-issue in-order dual-core microprocessor.

A. Fault injection setup

We have injected both single bit-flip transient faults and single stuck-at 0/1 permanent faults in the simulated processor while running original and protected versions of programs. We injected errors in 6 main hardware components including the register file, fetch and decode stage pipeline registers, functional units and load/store unit of the simulated microprocessor. For each hardware component, we inject 500 transient faults for each version of program and 100 permanent stuck-at faults. Therefore, 3,000 soft error (2% error with 95% confidence) and 600 hard error (5% error with 95% confidence) injection experiments were conducted per version of program [17]. Overall, we injected 81,000 (9 programs * 3000 processor wide fault injection site * 3 versions for each program) transient errors and 16,200 (9 * 600 * 3) permanent faults on all nine benchmarks and different version of programs.

Transient fault injection: we randomly select a bit and a cycle from a program fault space and inject a single bit flip error in the first instruction which utilizes that component. Particularly, we first make a per-component trace file which includes all instructions and the corresponding cycle time that they utilize the component, as well as their corresponding value. Then we randomly select a bit and an entity of the trace file, start the simulation, and whenever the execution reaches to the selected random cycle time, we modified the value(data) associated with the selected bit for the instruction which is utilizing that hardware component. For instance, for transient fault injection in load/store unit, we make a trace of all cycles that the load/store unit entries are occupied with program memory operations. Then we select an entry in trace file and a bit between 0 to 63 (address and data in each entry of load/store unit are 32-bit wide for each). We start the simulation till the selected entry cycle. After that, we pause the simulation a flip the selected bit in the associated data. Then we let the simulation run resumes till the program permanently terminates, or the allowable simulation run gets over. **Permanent fault injection:** We randomly select a specific bit in one of the cores hardware components for each simulation run. Then we permanently alter all data that utilize the targeted component in such way that the selected bit in the data is always zero/one. **Failure mode and comparison metric:** We classify the output of each fault injection simulation run into: i) SDC (Silent Data Corruption): faulty program terminates normally, but produce wrong output, and ii) Others: Program output is correct (fault was masked), or the injected fault lead to program crash, segmentation fault or infinite loops. We focus on SDC cases because on rest error is either benign or detected by the operating system. To show the real error detection capability of EXPERT and SRMT, we use normalized SDC which calculated as the absolute number of SDC multiply by a correction factor. The correction factor varies between benchmarks and is proportional to the performance and hardware overhead[2]. The correction factor crucial because as research [18] reveals using traditional fault coverage metric (the number of failed cases divided by the total number injection experiments) as a comparison metric for error detection transformations which prolong the execution time and/or demand more hardware resources will cause severe protection overestimation. The main reason is that considering same error rate for both original programs (run fast and just utilize one core) and protected ones (run slow and occupy two cores) is not fair.

B. Fault coverage

Figure 4 shows the normalized number of SDC for unprotected (ORG), SRMT and EXPERT versions of the programs. The figure indicates that of the 32.4k fault injection experiment runs, and the original programs result in 7,061 SDCs. The SRMT-protected version of programs ends up with 1,310 times occurrence of SDCs. As compared to these, the number of normalized SDC in EXPERT protected programs is just 20. We explored the failed cases of SRMT and realized SDCs in EXPERT are the result of faults which directly affected the execution of a memory operation while they are utilizing pipeline registers or address generation units. However, SRMT can detect the manifestation of almost all faults injected in the register file. The reason is that even for faults happening on the register file vulnerability window (explained in section II-B), still they will be detected by successive checks! That is because the discrepancy between leader and trailer threads registers will manifest itself in the upcoming checks. EXPERT transformation, on the other hand, can detect faults in all microprocessor components because its coverage includes the

[2]We consider the hardware overhead as 2x for both SRMT and EXPERT transformations, because of extra core required for redundant thread.

Fig. 4: EXPERT error detection coverage is ∼65x better than SRMT.

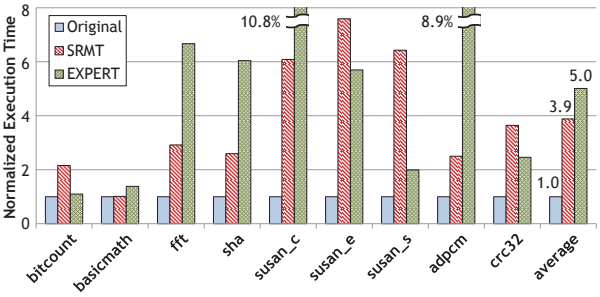

Fig. 5: EXPERT-protected programs run 30% slower than SRMT-protected ones.

execution of memory operations as well as the computational instructions. Only in some rare cases (bitcount and adpcm benchmarks), EXPERT transformation failed to detect the faults. We investigated these cases and realized that in all of them injected faults directly alter the effective address of a silent memory write operations – the data of store instruction was already presented in the store target memory location before execution of store. In those cases, the checker thread reads the data from the un-updated memory location, however, since the loaded data matches to the checker-thread computed value, the errors remains undetected.

C. Performance overhead

Figure 5 shows execution time overhead for SRMT and EXPERT transformations. On an average, SRMT and EXPERT transformations increase the programs execution time by on an average ∼3.9x and ∼5x, receptively. The performance overhead numbers presented in this work are consistent with prior works. For instance, original SRMT paper[6] reported ∼4x for Spec2000 Benchmarks and state-of-the-art RMT schemes for GPU applications [10], reported ∼6x performance degradation for inter-group RMT after applying different optimizations.
The main point of the results shown in figure 5 is that the runtime overhead of both SRMT and EXPERT schemes is heavily dependent on program characteristics. In computation hungry programs like bitcount, basicmath and crc, the execution time overhead for both SRMT and EXPERT is around 1.5x-3x. On the other hand, for memory-intensive programs with many shared memory operations like susan (corners), susan (e) and adpcm, both SRMT and EXPERT suffer from significant performance overhead. The overhead is mainly because of the frequent data communication and synchronization points between redundant threads. For applications that consist of many shared memory load operations and a few store instruction like susan (smoothing), EXPERT-protected applications run faster than SRMT-protected ones. Note that the proposed memory packing optimization (explained in section IV) for EXPERT programs is always beneficial and on an average reduces the overhead of EXPERT by 30%.

VI. Conclusions

We presented EXPERT, a compiler-level RMT soft/hard error detection scheme. EXPERT significantly (∼65x) improves the error coverage of state-of-the-art RMT schemes by providing full execution protection rather than just computational operations protection.

VII. Acknowledgements

This work was partially supported by funding from NSF CCF 1055094 (CAREER); by global PH.D fellowship program through the NRF funded by the Ministry of Education (NRF-2016H1A2A1909470); by next-generation information computing development program through the NRF funded by the Ministry of Science, ICT, and Future Planning (NRF-2015M3C4A7065522).

References

[1] G. Reis *et al.*, "SWIFT: Software Implemented Fault Tolerance," in *CGO*. IEEE, 2005.
[2] M. Didehban *et al.*, "nZDC: a compiler technique for near Zero Silent data Corruption," in *DAC*. ACM, 2016.
[3] Q. Liu *et al.*, "Compiler-Directed Soft Error Detection and Recovery to Avoid DUE and SDC via tail-DMR," *TECS*, vol. 16, no. 2, 2016.
[4] M. Didehban *et al.*, "Incheck: An in-application recovery scheme for soft errors," in *DAC*. ACM, 2017.
[5] M. Didehban *et al.*, "NEMESIS: A software approach for computing in presence of soft errors," in *IEEE ICCAD*, 2017.
[6] C. Wang *et al.*, "Compiler-managed software-based redundant multi-threading for transient fault detection," in *CGO*, 2007.
[7] Y. Zhang *et al.*, "DAFT: Decoupled Acyclic Fault Tolerance," *International Journal of Parallel Programming*, 2012.
[8] K. Mitropoulou *et al.*, "Comet: communication-optimised multi-threaded error-detection technique," in *CASES*. ACM, 2016.
[9] S. Hukerikar *et al.*, "Redthreads: An interface for application-level fault detection/correction through adaptive redundant multithreading," *IJPP*, 2016.
[10] M. Gupta *et al.*, "Compiler techniques to reduce the synchronization overhead of gpu redundant multithreading," in *DAC*, 2017.
[11] N. Oh *et al.*, "ED4I: Error Detection by Diverse Data and Duplicated Instructions," *IEEE TOC*, vol. 51, no. 2, 2002.
[12] U. Schiffel *et al.*, "Anb-and anbdmem-encoding: detecting hardware errors in software," *SAFECOMP*, 2010.
[13] A. Shye *et al.*, "PLR: A software approach to transient fault tolerance for multicore architectures," *IEEE TDSC*, vol. 6, no. 2, 2009.
[14] Y. Zhang *et al.*, "Runtime asynchronous fault tolerance via speculation," in *CGO*. ACM, 2012.
[15] J. Wadden *et al.*, "Real-world design and evaluation of compiler-managed gpu redundant multithreading," in *ISCA*. IEEE, 2014.
[16] N. Binkert *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, 2011.
[17] R. Leveugle *et al.*, "Statistical fault injection: quantified error and confidence," in *DATE*. IEEE, 2009.
[18] H. Schirmeier *et al.*, "Avoiding pitfalls in fault-injection based comparison of program susceptibility to soft errors," in *DSN*, 2015.