An Integrated Safe and Fast Recovery Scheme from Soft Errors

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

ABSTRACT

An ideal soft error tolerance solution should hide the effect of soft errors from user and provide correct results at expected time. Software solutions are attractive because they can provide flexible reliability without imposing any hardware modifications. Our investigation of state-of-the-art error recovery techniques reveals that they suffer from poor coverage (ability to detect and correctly recover from soft errors). This paper presents InCheck (In-application Checkpointing and Recovery) as an effective, safe and timely software technique for complete error coverage. The key features of InCheck are: verified register preservation, single memory location checkpoints, and safe & timely recovery. To evaluate the effectiveness of InCheck, we performed more than 210,000 fault injection experiments on different hardware components of an ARM cortex53-like processor running MiBench applications. The original and SWIFT-R (state-of-the-art) protected programs suffered from 8000 and 1800 instances of wrong outputs respectively, but when protected by InCheck, there was no failure.

1. INTRODUCTION

Soft errors or transient faults – caused by high energy particles that lead to an unexpected change in the transistor logic – have long been considered as the main reliability challenge for mission-critical aviation applications[25]. However aggressive sub-nano transistor scaling (10nm-7nm) and near-threshold supply voltage is making modern digital circuits that we use in our everyday lives ever more susceptible to external noises. Now even low-energy terrestrial particles [1, 9] like muons' can cause soft errors[23, 24]. In fact, the ITRS 2015 [34] report lists Muon-induced soft errors as a major reliability challenge in both near-term and long-term microprocessor designs. Therefore soft error resilience will soon become a must – even for safety-critical terrestrial applications.

Conventionally, hardware level soft-error resilience techniques have been employed in mission- and safety-critical systems, like spacecrafts and enterprise system [35]. However, hardware solutions come with high costs owing to the need to redesign the system. Software approaches are attractive, as they can provide a flexible, and affordable solution. They are especially useful for mixed-critical systems [**?**], where flexible software techniques can provide reliability based on task requirements. For instance, in automobile applications where the safety-critical tasks (braking) and non-critical tasks (entertainment systems) share the underlying microprocessor, software techniques are preferable as they can meet high-reliability requirements for critical tasks, while non-critical tasks get the best performance from underlying microprocessor. Although existing software techniques can meet the reliability requirements for medium- and less-critical tasks, they do not provide high error resiliency (detection+recovery) demanded by high-critical tasks [5].

An ideal error resilient scheme should provide complete, effective and timely recovery from soft errors. Many of the software-level fault tolerance techniques are *incomplete*, because they provide error detection and assume some sort of checkpoint/roll-back [7, 27, 33, 31, 16, 14, 9] for recovery. Restarting a program from beginning is the simplest rollback recovery strategy. However, re-starting is not applicable in many cases, i.e, long running, real-time and interactive applications [32], and is accompanied by a high error recovery latency – expected recovery latency can be half of the program execution time. These problems can be alleviated by building full-system checkpoints (preserving the whole memory and register stats) during the execution of a program [2, 12]. However, to solve the problem of latent errors (errors which may happen before checkpointing and will be detected after checkpointing) frequent checkpoints are required, which imposes unacceptable performance overhead to the system [8, 19, 3].

Software forward-recovery techniques [15, 22, 17] which execute three versions of computations and perform majority-voting between the results can potentially solve the problem of latent errors in a more efficient way than frequent full system checkpointing. The recovery-latency of voting-based techniques is small (around 10 machine instructions), but they cannot provide reliability demanded by high-critical applications mainly due to considerable single-points-of-failure introduced by voting [26, 29]. Moreover, coarse-grained forward-recovery schemes [15, 22] demand more than 3x memory overhead. However, since the memory subsystems are usually ECC-protected in many modern microprocessor, applying redundancy on memory is a waste.

In-application fault tolerant techniques can potentially eliminate the need for full-system checkpointing and memory replication, while providing efficient and timely error handling by combining both error detection and recovery within the application itself. Unfortunately, the existing in-application error tolerant schemes are significantly weaker than their underlying detection schemes, due to the vulnerability added by their complex error recovery routines. For instance, SWIFT-R [17] was proposed to provide recoverability to SWIFT (an error-detection only technique) [16] by adopting forward recovery strategy. SWIFT-R divides the program registers into 3 redundant sets, executes 3 versions of each computational instruction and performs majority-voting amongst register operands of memory and control-flow instructions before their execution. Surprisingly, our analysis of SWIFT-R reveals that it has about 16x more vulnerability than SWIFT! This is because: **i)** SWIFT-R transformation requires considerably more unprotected (memory) instructions than SWIFT due to its high register pressure, **ii)** SWIFT-R has software vulnerable windows larger than SWIFT, as it replaces light (in terms of machine instructions) error detection checks of SWIFT with heavier voting operations, and **(iii)** SWIFT-R provides unsafe recovery by blindly masking the effect of certain errors on registers that may have already propagated into memory or may even have altered the program control-flow.

Realizing these problems, we propose InCheck (INtegrated safe CHECKpointing and Recovery) – a software-only complete, safe & timely recovery scheme from soft errors. InCheck makes light-weight error-free checkpoints at basic block granularity, and safely reverts the program execution to the beginning of last executed basic block using preserved checkpoints. The main features of InCheck are:

i) Verified Register File Preservation. InCheck transformation not only preserves registers' value into memory (no latent error), but also makes sure that the preserving process happened correctly.

ii) Single Memory-location Checkpointing. Rather than checkpointing the whole memory state, InCheck just preserves the state of each memory location temporarily before of each write.

iii) Safe & Timely recovery. Instead of performing recovery regardless of the error propagation scope, InCheck invokes a diagnosis routine which allows recovery only when its safe. The recovery latency of InCheck is negligible as it involves re-execution of just one basic-block's instructions apart from diagnosis and recovery routines.

In order to evaluate the effectiveness of InCheck, we performed about 216,000 micro-architectural single bit-flip fault injection experiments on an ARM Cortex-A53 like simulated processor while running Mibench [10] programs. The results demonstrate that InCheck protected programs never generated a wrong output and in more than 96% of detected faults, it also provides a safe & timely recovery. Though SWIFT-R reduced the overall SDC (Silent-Data Corruptions) count in Original Programs by about 4.3x, it still accounted for about 1800 SDCs in the total injected faults on SWIFT-R protected programs.

2. LIMITATIONS OF PREVIOUS WORKS

2.1 Checkpoint/Rollback Recovery

Traditionally Checkpointing/rollback has been used a recovery strategy in High Performance Computing (HPC) systems. The program execution in such systems is periodically paused to make checkpoints (snapshots of the entire program state including memory footprint and register values) into a safe storage. In case of an error, last checkpoint will be considered as safe process state and the program resumes its execution [2].

Applying full system checkpointing/rollback as error recovery in embedded critical applications in not efficient because: i) Significant recovery latency (millions or billion of instructions depends on the checkpointing interval), ii) Unacceptable performance overhead (Frequent checkpointing is required for latent error recovery[3]) and iii) Need of extra safe storage (around 0.5-1 Giga bytes per checkpoint). In fact, this huge overhead of frequent and multiple checkpoints (required for successful and fast recovery from silent errors) has restricted the usage of such recovery techniques to HPC systems alone [8, 19].

To overcome the limitations of full checkpointing, techniques like Encore[21], Clover[11] and FASER[28] have introduced fast and low-overhead recovery schemes by taking advantage of the idempotent regions of the program codes. Idempotent segments of a code do not have any Write after Read (WAR) dependencies. Therefore, multiple re-executions of such region always produces same result after execution. Nevertheless, Idempotency-based recovery techniques have been designed for non-critical applications and have narrow fault coverage (recovery just from some specific soft errors which affect idempotent-regions of code). For instance, faults which cause a write into the wrong memory address or control flow errors (a fairly large amount of errors) cannot be recovered by these techniques, even if they occur in idempotent region of code. The authors of Encore themselves acknowledged to these limitations. Hence, the ineffectiveness of idempotency-based recovery schemes renders them unsuitable for high-critical applications.

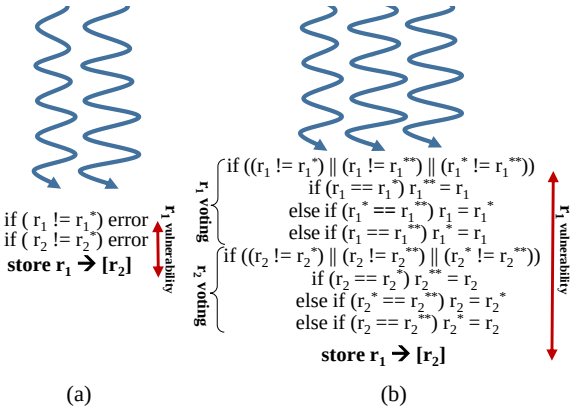

Figure 1: SWIFT-R (part b) software vulnerability interval is considerably more than SWIFT (part a)

2.2 Coarse-Grain Forward Recovery

Forward recovery schemes which work based on the triplication and voting strategy, eliminate the need for checkpointing and can provide timely recovery. Forward-recovery can take place at coarse-grain (like process/task/thread replication) or fine-grain (assembly-level instruction replication) modular redundancy. State-of-the-art coarse-grain techniques like PLR (Process-level Redundancy [22]) apply redundancy at process-level, and perform voting between the redundantly-computed system call arguments at system-call boundaries. However, since the errors can affect a program without manifesting into the system call arguments, PLR approach is not effective for critical applications. For instance, if a system call (like `fwrite`) takes a memory pointer and size as arguments, there can be errors in the actual data referenced by the pointer even if the arguments (redundantly computed pointers and size) are equal. In addition, software voting operations and the execution of system call themselves are the single-points-of-failures in such techniques.

2.3 Fine-Grain Recovery

Fine-grained assembly-level techniques [17, 16, 9, 14, 7, 13, 28, 30, 27] are the most related techniques to the work presented in this paper. They are very popular as they can provide potentially high-level of reliability. This is because such techniques are implemented as machine-level instructions, and have the ability to effectively check for the errors which may cause a failure, i.e, in memory operations and control-flow direction. Unfortunately, the existing complete fine-grained techniques, SWIFT-R [17] and FASER [28], provide recovery from some errors with a significantly higher failure rate that the error detection schemes.

2.3.1 SWIFT-R: Unsafe Recovery

The idea of providing the error detection and recovery at assembly-level was introduced in SWIFT-R [17] research paper. SWIFT-R (SWIFT+Recovery) is developed based on the well-known SWIFT[16] technique. It executes three versions of program's computational instructions with different sets of registers and performs 2-of-3 majority-voting between redundant registers' values before memory and control-flow instructions to mask the effect of error from computations. However, SWIFT-R not only imposes significant performance overhead to SWIFT, but also increases SWIFT's failure rate because of three main reasons:

1) SWIFT-R transformation demands considerably

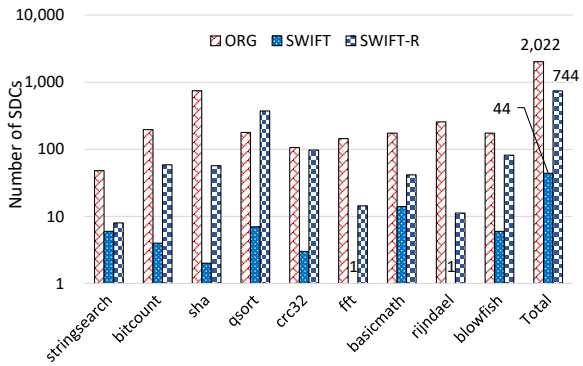

Figure 2: SWIFT-R protected programs experienced more than 16x failure than SWIFT-protected ones!

more unprotected instructions than SWIFT. SWIFT-R requires about two-thirds of programmer available registers for error recovery, while SWIFT reserves about half of registers for error detection. This extra register reservation forces compiler to generate more memory instructions which cannot get protected by SWIFT and SWIFT-R transformations [7]. To quantify the effect of register reservation on programs, we compiled Mibench[10] programs with LLVM 3.7[6] infrastructure for ARMv8-A Architecture which has 32 general purpose integer registers[1]. We found out that SWIFT-R imposes more than 4x memory operation than SWIFT. Note that, FASER [28], a SWIFT-based backward-recovery, also share this problem with SWIFT-R.

2) SWIFT-R suffers from considerably larger software vulnerability window than SWIFT. SWIFT-R increases software vulnerability window (interval between value checking and actual usage of that value) of SWIFT, because it replaces light error-detectors (1-2 machine instructions), with expensive majority-voting operations (8-10 machine instructions). Figure 1 illustrates the software vulnerable window of SWIFT and SWIFT-R for a simple memory write operation. SWIFT just checks the store's value(`r1`) and address(`r2`) registers, against their shadows (`r1*` and `r2*`) before the actual memory write instructions. Therefore, there is a small interval between the checking and using register values, which if error hits the registers in this interval, error may remain undetected. SWIFT-R, on the other hand, requires majority-voting between three redundant registers values computed for stores value (`r1`, `r1*` and `r1**`) and address(`r2`, `r2*` and `r2**`) registers. Clearly, since the software implementation of the 2-of-3 majority-voting needs more machine instructions than just error checking, software vulnerable interval of SWIFT-R is larger than SWIFT. Besides of the longer duration of each software vulnerable window, the number of such intervals in SWIFT-R is also considerably bigger than SWIFT, because SWIFT-R demands more memory (therefore voting) operations. Note that these software vulnerable intervals are considerable, because they exist before all memory, control-flow and function calls instructions.

3) SWIFT-R unsafe recovery eliminates the effect of error from registers even though memory or control flow is faulty. If an error happens during SWIFT vulnerability window, it probably corrupts the memory state and/or alters the program control-flow. SWIFT can still detect such errors, if the mismatch between redundant registers reaches to the successive error detectors. However, since SWIFT-R and FASER recovery schemes, try to blindly recover from any discrepancy (without determining the scope of error propagation), they can mask the effect of error in registers while the memory state or the program execution path is still erroneous.

[1]Implementation of SWIFT-R and FASER on an ARMv7 microprocessor is problematic because only 16 user visible registers are available.

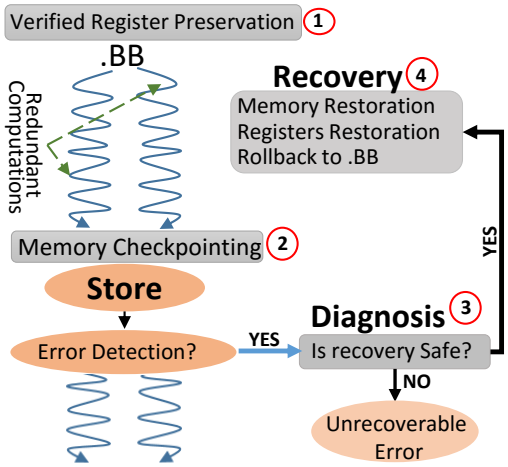

Figure 3: InCheck data-flow error recovery Overview

In order to quantify SWIFT-R's negative impact on the failure rate, we performed 54k fault injection experiments on Register File of a simulated processor while running SWIFT and SWIFT-R protected programs. The details of the simulator configuration and fault injection set-up are presented in section 4. As figure 2 demonstrates, SWIFT-R transformation caused around 16x more failure (SDC) than SWIFT! *Therefore, recovery from errors is a delicate process which if not performed correctly, will worsen system reliability and execution time.*

3. OUR APPROACH: INCHECK

3.1 InCheck Overview

In this paper, we propose a safe and timely recovery scheme which in combination with nZDC (**n**ear **Z**ero silent **D**ata **C**orruption[7]) as the underlying error detection scheme, will serve as a complete, safe and timely error handling strategy. InCheck's error handling process can be divided into two main parts: DF (Data-Flow) and CF (Control-Flow) error recovery. DF error recovery (shown in figure 3) consists of four main steps: 1) Verified Register Preservation. It takes place at the beginning of each basic block and stores the value of live registers into the checkpointing area of memory. InCheck makes sure that no error can cross the preservation phase by checking the the process of preservation itself in-addition to checking the reserved registers' values. 2) Single Memory-location checkpointing. Right before each store instruction, InCheck preserves the data of about-to-be-updated memory location into a specific register. This register will further be used for memory restoration in the case of errors. 3) Checks for Safe Recovery. InCheck diagnosis routine checks if the error is recoverable. This is necessary because state recovery is not always possible. For instance, errors which cause a write into a memory location that is different from the one that the back-up load reads from, cannot be recovered because the backup itself is not valid. 4) Timely Recovery. The program execution is resumed from the beginning of basic block after the Memory and Register file state are restored to fault-free state that was present at the beginning of that basic block. Timely recovery was possible as overall operations needed for diagnosis and recovery are implemented within 100 instructions.

Control-flow error recovery is similar to DF error recovery,

however, the challenge is to determine from where the program re-execution should be restarted. InCheck CF diagnosis routine separates wrong-direction (errors which alter the direction of branch) CF errors from wrong-target ones(errors which cause an illegal jump), and provides recovery for the former and safe-stop for latter.

3.2 Verified Register File Preservation

InCheck saves the value of live registers into a designated memory locations, called register preservation area, at entrance of each basic block. Just error-free registers should be preserved and the preservation process should be error-free, otherwise, a failure may be observed. InCheck validates the correctness of preservation process by applying checking-load strategy [7] – It loads back the saved register value from the register preservation area and checks that against the shadow register. Note that the program counter register (PC) always considered as live and get preserved. The PC preservation is crucial for control-flow errors (described in section 3.6) detected in fan-in basic blocks where potentially multiple re-execution points exists.

3.3 Single Memory-Location Checkpointing

InChecks introduces a novel and efficient method for memory checkpointing, and rather than saving the whole memory sate or be rely on idempotent region of codes[21, 11, 28], it just makes a back-up from the memory location which is going to be overwritten. The main intuition behind this strategy is that since memory subsystem can be protected by ECC, there is no need to save whole memory state. However, ECC is ineffective if erroneous data or address is sent along write command. Therefore, the previous value of about-to-update memory location is needed for memory restoration.

InCheck provides memory checkpointing by inserting a load instruction (back-up load) from the exact address as the following memory write instruction into an specific register, named MBR (Memory Back-up Register). InCheck transformation forces compiler to break down basic blocks with potentially conflicting memory operations (multiple write and read operations from the same memory location) into sub basic blocks without such memory dependencies. This basic block purification is required for recovery from basic blocks with conflicting memory operations, because InCheck on-the-fly single memory location checkpointing strategy just provide backup for one memory location.

3.4 Checks for Safe Recovery

One of the feature of InCheck which distinguish it from related techniques is its diagnosis routine, which is the essence of safe recovery. Basically, if error affects the execution of redundant computations or error detection instructions (shown is figure 3) the error is always recoverable. However, if error impacts the execution of store instruction in such way that the effective memory address get modified, the error should be considered as unrecoverable, since the data will be written into an unknown (unbacked-up) memory location.

Figure 4 shows an example of InCheck data-flow error detection and diagnosis. The first `load`(left side of the figure, before `store`) is the "back-up load" which performs on-demand memory checkpointing. The error detection takes place after the `store` instruction, and program control goes to diagnosis routine in the case of mismatch. First, diagnosis routine checks for errors on the computation of `store` value register (`r1`). If a mismatch is observed the error gets marked as recoverable, and the program jumps to the recovery routine (not shown in figure). Second, diagnosis routine checks for mismatch on the `store` address register, which depends on error occurrence-time may or may not be recoverable. If error occurs on address register `r2` before the back-up load, the error consider as recoverable. This is because the back-up for wrongly updated memory is available, and therefore, memory restoration is possible. However, if the error happens after the execution of back-up load, since the value of MBR is not the previous value of the wrongly updated memory, the recovery is not possible. To determine the occurrence time of error, diagnosis routine loads the data back from the memory location with the same address as `store` into a temporary register (`temp`) and compares that against `store` value register (`r1`). If same, it assumes that error has modified the address of both back-up load and the `store` in the same way (back-up is valid) and is recoverable. Third, diagnosis routine compares the value of MBR to `temp`. If different, it implies that the `store` has written incorrect data into right memory location. This type of error is also recoverable because memory back-up is valid. Fourth, diagnosis checks for errors on detection instructions which are just false alarms and easily recoverable. False alarms can be checked by repeating the error detection instructions. Ultimately, if none of the above situation was true, the diagnosis routine declares the error as detected/Unrecoverable and terminates the execution of program.

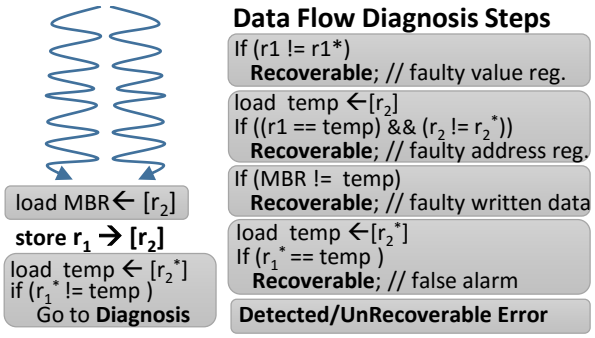

Figure 4: An example of InCheck data-flow diagnosis

3.5 Timely Recovery

In the last phase of InCheck error handling, the actual recovery takes place by performing memory and register file restoration and re-executing the program from the beginning of the basic block. First, the memory state will be restored back into the same state as before memory write instruction by writing the MBR register into the memory right target location. Then the error-free live registers will loaded back from the register preservation area to the corresponding registers, and ultimately the main program execution resumes. Since the first two steps of InCheck (safe register preservation and memory checkpointing) should be executed in all (erroneous or error-free) cases, the recovery latency of InCheck is equal to the execution time of diagnosis and recovery routines and replied instructions (instructions from the beginning of basic-block till the error detection point). Overall, since the diagnosis and recovery routines and the average basic block size are small, the recovery latency is always negligible (less than 300 instructions).

3.6 Control-Flow Error Recovery

InCheck employees nZDC control-flow(CF) checking mechanism, but in addition to nZDC CF error detection checks (positioned close to the end of each basic block) it also performs error detection checks an the beginning of each basic block (before safe register preservation)[2]. If a CF error gets detected by the first checks, InCheck invokes the corresponding CF-specific diagnosis routine. These routines are differ-

[2]The reader is encouraged to examine [7] for a detailed description of the nZDC control-flow detection scheme.

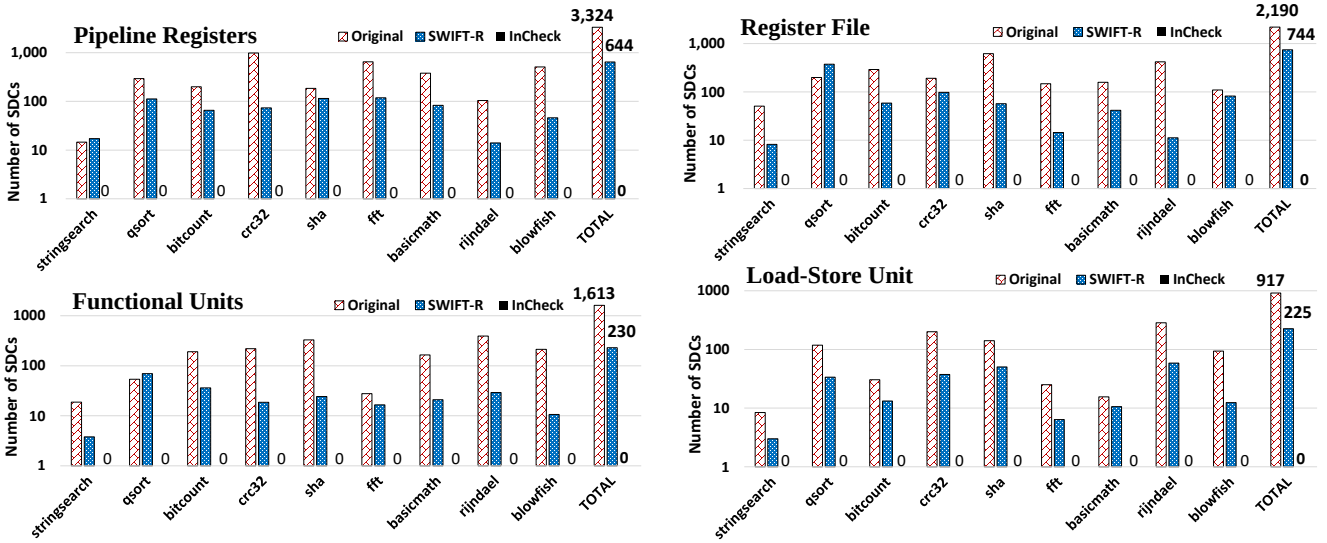

Figure 5: SDCs Distribution in Component-wise Fault Injection Experiments

ent from DF diagnosis routine (described in section 3.4), and their responsibility is to determine if the detected CF error is a wrong-direction or a wrong-target error. A control-flow error will be considered as wrong-direction error, if the last preserved PC is in the list of legitimate predecessors of the current basic block. If that is the case, error will be treated as recoverable and recovery takes place by restoring memory and register values to the initial state of the previously executed basic blocks. Otherwise, the error will be considered as wrong-target CF error and as well as errors which are detected by nZDC CF error detectors (positioned at the end of basic block) will be considered as unrecoverable – diagnosis routine terminates the execution of program. Fortunately, as [20] demonstrated, most of control-flow errors are wrong-direction errors, therefore are recoverable by InCheck error handling scheme.

4. EXPERIMENTAL METHODOLOGY

To quantify the effectiveness of InCheck, we implemented InCheck and SWIFT-R(as the the-state-of-the-art related work) techniques as late back-end passes in LLVM 3.7 infrastructure [6] for an ARMv8-a ISA(64-bit architecture). We compiled 9 programs from Mibench benchmark suite[10] with *-O3* compiler optimization flag. For each program three versions, Original, InCheck and SWIFT-R was produced. It should be noted that all experiments and results were performed on user functions (library functions were excluded).

We performed extensive fault injection experiments in all major sequential hardware components of a modern ARM cortex A-53 like simulated microprocessor. We performed our experiments on gem5[4], a cycle accurate simulator which the configuration shown in table 1. We performed single bit-flip fault injection experiments on major core components including, integer register file, decode and issue pipeline registers, functional units and load-store unit buffer registers. For each component 2000 faults were injected per version of program, which means 72,000 (4 * 2000 * 9) faults per each program version – overall 216,000 (72k * 3) faults. For each fault injection experiment, a target component and a (*bit*, *cycle*) are randomly selected before the simulation run. Once the simulator reaches the target fault injection *cycle*, simulation is paused and the selected *bit* is inverted. Then, simulation resumes with the faulty value until it gets terminated or the allowed simulation time (3x of normal execution time) gets over. The result of each simulation run is classified as one of the following:
1) Masked: Program terminates normally and the output is correct. **2) Failed/SDC:** Program terminates, but the output is incorrect. **3) Detected/Unrecoverable:** This outcome occur just in InCheck protected programs, and happens when an error is detected, but cannot be recovered from. **4) Others:** Program encounters a fatal error, such as segmentation fault or simulation time reaches its limit.

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

Parameter	Value
CPU Model	ARM64 bit in-order processor
Pipeline	Two issue/4-stage
Number of FUs	2Int, 1Mul, 1Div, 1Float, 1Mem
L1 D/I-Cache	64KB (2-way) / 32KB (2-way)
TLB size	512 entries
Integer registerfile	32 registers (64-bit width)
Store buffer size	5 entries

5. EXPERIMENTAL RESULTS

5.1 InCheck: Effective & Safe Error Handling

Figure 5 depicts the absolute number (in logarithmic scale) of failures (SDCs) per hardware component. We did not use fault coverage metric, because it can be misleading [18]). Regardless of the target fault injection component, InCheck-protected programs never resulted in a failure! This implies that 1) No error could skip InCheck+nZDC's error detectors, 2) The diagnosis routine always distinguishes recoverable errors from unrecoverable ones accurately, and 3) If the detected error was recognized as recoverable, the recovery routine is always successful. InCheck is extremely effective as it protects functionally-related instructions of the program as well as error handling (preservation and checkpointing) operations. However, in comparison with the original programs, SWIFT-R transformation reduces the number of failures on total by 4.3x (5.2x, 2.9x, 8x and 4x for pipeline registers, register file, functional units and load-store unit, respectively). Our investigation from failed experiments reveals that unlike InCheck SWIFT-R provides safe recovery only from the faults which affect the computational instructions, and the rest of the faults either get masked by the

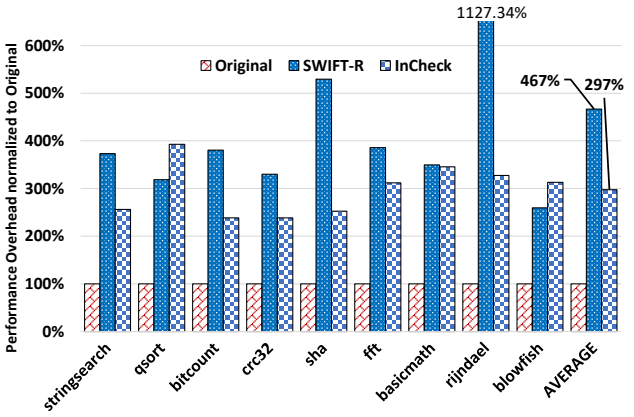

Figure 6: Execution Overhead of SWIFT-R & InCheck

programs or lead to failures or segmentation faults.
In-Check-protected programs can potentially recover from Soft Errors which lead to segmentation faults if their diagnosis routines initialize at the beginning of signal handler functions of applications. Since in gem5 system call emulation mode, the simulator terminates the program execution without forwarding segmentation fault signals to the application, We could not fully demonstrate the InCheck Recoverability.
InCheck Diagnosis routine declared around 96% of detected faults as recoverable. In less than 4% of times diagnosis routine provided safe-stop and prevented from failure by terminating the running program. If they were left unterminated, these unrecoverable faults could have either directly impacted the execution of a memory write operations or caused an unexpected jump in the program. Restarting can anyways be employed as recovery strategy in these scenarios.

5.2 InCheck: Efficient and Timely Recovery

Figure 6 shows the execution overheads of InCheck+nZDC and SWIFT-R protected programs normalized to Original Program. It can be seen that on an average, an InCheck version of a program can run 36% faster than its SWIFT-R equivalent. InCheck is faster because it pushes the uncommon diagnosis and recovery routines off the critical-path of execution. The performance overhead of frequent live register preservation is acceptable, because the corresponding memory preservation locations are usually presented in the cache and therefore will execute fast. Furthermore, the performance overhead of back-up loads (inserted right before program store instructions) are also not significant, because they do not cause any more memory miss – if the data is not in the cache, miss is inevitable. If not happen by back-up load it will be happen by store instruction itself.
To quantify the recovery-latency of InCheck, we counted the average number of extra instructions which were executed when an injected fault was detected and recovered. The InCheck recovery-latency in terms of dynamic instructions is, on an average, around 180 instructions which is unnoticeable in many cases.

6. CONCLUSIONS

We presented InCheck, as an in-application soft Error detection, diagnosis and recovery scheme. Unlike existing techniques, InCheck protects the execution of error handling routines like checkpointing operations in-addition to the main program instructions. InCheck introduces verified register file preservation and single-memory-location checkpointing. It also performs diagnosis after error detection to provide safe recovery.

Acknowledgement

This work was partially supported by funding from National Science Foundation grants CCF 1055094 (CAREER).

References

[1] Shekhar Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *MICRO*, 2005.
[2] Jason Duell. The design and implementation of berkeley lab's linux checkpoint/restart. *Lawrence Berkeley National Laboratory*, 2005.
[3] Aupy et.al. On the combination of silent error detection and checkpointing. In *PRDC-19*. IEEE, 2013.
[4] Binkert et.al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 2011.
[5] Cheng et.al. CLEAR: Cross-Layer Exploration for Architecting Resilience-Combining Hardware and Software Techniques to Tolerate Soft Errors in Processor Cores. *arXiv preprint arXiv:1604.03062*, 2016.
[6] Chris et.al. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, 2004.
[7] Didehban et.al. nZDC: A compiler technique for near Zero silent Data Corruption. In *DAC-53*. ACM, 2016.
[8] Elnozahy et.al. Checkpointing for peta-scale systems: A look into the future of practical rollback-recovery. *TDSC*, 2004.
[9] Feng et.al. Shoestring: probabilistic soft error reliability on the cheap. In *ASPLOS*. ACM, 2010.
[10] Guthaus et.al. MiBench: A free, commercially representative embedded benchmark suite. In *WWC-4*. IEEE, 2001.
[11] Liu et.al. Clover: Compiler directed lightweight soft error resilience. In *LCTES*. ACM, 2015.
[12] Lu et.al. When is multi-version checkpointing needed? In *FTXS-3*. ACM, 2013.
[13] Mitropoulou et.al. DRIFT: Decoupled CompileR-Based Instruction-Level Fault-Tolerance. In *LCPC*. Springer, 2014.
[14] Oh et.al. ED4I: Error Detection by Diverse Data and Duplicated Instructions. *TC*, 2002.
[15] Quinn et.al. Software resilience and the effectiveness of software mitigation in microcontrollers. *TNS*, 2015.
[16] Reis et.al. SWIFT: Software implemented fault tolerance. In *CGO*. IEEE, 2005.
[17] Reis et.al. Automatic instruction-level software-only recovery. *MICRO*, 2007.
[18] Schirmeier et.al. Avoiding pitfalls in fault-injection based comparison of program ... In *DSN*, 2015.
[19] Schroeder et.al. Understanding failures in petascale computers. In *SciDAC*. IOP Publishing, 2007.
[20] Shrivastava et.al. Quantitative analysis of control flow checking mechanisms for soft errors. In *DAC-51*. IEEE, 2014.
[21] Shuguang et.al. Encore: low-cost, fine-grained transient fault recovery. In *MICRO-44*. ACM, 2011.
[22] Shye et.al. PLR: A software approach to transient fault tolerance for multicore architectures. *TDSC*, 2009.
[23] Sierawski et.al. Effects of scaling on muon-induced soft errors. In *International Reliability Physics Symposium*, 2011.
[24] Silberberg et.al. Neutron generated single-event upsets in the atmosphere. *TNS*, 1984.
[25] Taber et.al. Single event upset in avionics. *TNS*, 1993.
[26] Ulbrich et.al. Eliminating single points of failure in software-based redundancy. In *EDCC-9*. IEEE, 2012.
[27] Wang et.al. Compiler-managed software-based redundant multi-threading for transient fault detection. In *CGO*, 2007.
[28] Xu et.al. An instruction-level fine-grained recovery approach for soft errors. In *SAC-28*. ACM, 2013.
[29] Yim et.al. A fault-tolerant programmable voter for software-based n-modular redundancy. In *AeroConf*. IEEE, 2012.
[30] Yu et.al. Esoftcheck: Removal of non-vital checks for fault tolerance. In *CGO-7*. IEEE Computer Society, 2009.
[31] Yun et.al. DAFT: Decoupled Acyclic Fault Tolerance. *International Journal of Parallel Programming*, 2012.
[32] Zhang et.al. Fault recovery based on checkpointing for hard real-time embedded systems. In *DFT-18*. IEEE, 2003.
[33] Zhang et.al. Runtime asynchronous fault tolerance via speculation. In *CGO-10*. ACM, 2012.
[34] IRC. International Technology Roadmap For Semiconductors 2.0-Executive Summary. http://www.itrs2.net/itrs-reports.html, 2015. [accessed 19-November-2016].
[35] Spainhoweret.al. Ibm s/390 parallel enterprise server g5 fault tolerance: A historical perspective. *IBM J RES DEV*, 1999.