Clover: Compiler Directed Lightweight Soft Error Resilience

Qingrui Liu [1] Changhee Jung [1] Dongyoon Lee [1] Devesh Tiwari[2]

[1]Virginia Tech, USA [2]Oak Ridge National Laboratory, USA

{lqingrui, chjung, dongyoon}@vt.edu tiwari@ornl.gov

Abstract

This paper presents Clover, a compiler directed soft error detection and recovery scheme for lightweight soft error resilience. The compiler carefully generates soft error tolerant code based on idempotent processing without explicit checkpoint. During program execution, Clover relies on a small number of acoustic wave detectors deployed in the processor to identify soft errors by sensing the wave made by a particle strike. To cope with DUE (detected unrecoverable errors) caused by the sensing latency of error detection, Clover leverages a novel selective instruction duplication technique called tail-DMR (dual modular redundancy). Once a soft error is detected by either the sensor or the tail-DMR, Clover takes care of the error as in the case of exception handling. To recover from the error, Clover simply redirects program control to the beginning of the code region where the error is detected. The experiment results demonstrate that the average runtime overhead is only 26%, which is a 75% reduction compared to that of the state-of-the-art soft error resilience technique.

Categories and Subject Descriptors B.8.1 [*Performance and Reliability*]: Reliability, Testing, and Fault Tolerance; D.3.4 [*Programming Languages*]: Processors—Compilers

General Terms Reliability, Languages

Keywords Soft Error Resilience; Compilers; Tail-DMR Frontier; Idempotent Processing; Acoustic Wave Detectors

1. Introduction

Resilience against soft errors is one of the key research challenges for current and future computing systems. Soft errors have been the cause of a significant number of failures in real-world systems, ranging from embedded systems to large-scale high performance computing (HPC) systems [7, 13, 21, 22, 25, 31]. Unfortunately, due to technology scaling, electronic circuits are likely to be more susceptible to radiation-induced soft errors (also known as transient faults). Soft errors are typically caused by cosmic rays and alpha particles from packaging material. Soft errors may lead to application crash or even worse, silent data corruptions (SDC) which are not caught by the error detection logic but may cause the program to produce incorrect output. Another worst type of results are detected unrecoverable errors (DUE) that often directly impact the reliability of the computer systems. To achieve the resilience, it is essential to have both the detection and the correction of soft errors.

In the dark silicon era, soft errors are becoming increasingly important concern for computer system reliability. Ever-growing power density due to the limited supply voltage scaling is leading toward the advent of near-threshold computing that can improve energy efficiency by an order of magnitude, but at the expense of near-threshold voltage and lower frequency [33, 36]. However, the near-threshold voltage and the process variation make it harder to predict the response of the circuits to a particle strike, thus making them much more susceptible to soft errors. According to Shafique et al. [32], near-threshold voltage operation may cause up to 30x higher soft error rate than nominal voltage operation. Similar trends have been observed by other researchers as well [14, 16]. Consequently, soft error resilience is essential not just for guaranteeing program correctness, but also for realizing the full potential of near-threshold voltage computing to maximize energy efficiency, which is particularly important for energy constrained embedded systems.

These trends have motivated researchers to devise effective resilience mechanisms to mitigate the side effects of soft errors. Unfortunately, existing techniques often suffer from too high performance overhead [23, 28, 29] or require costly hardware support and resource consumption (e.g., occupying entire cores or leveraging special microarchitecture) [2, 4, 24, 26]. Despite increased hardware, performance and power costs, these techniques may not eliminate both the SDC and the DUE, or need for expensive checkpointing. To address these issues, this paper presents Clover, a compiler directed lightweight resilience scheme that can detect and correct soft errors without the need for checkpointing and high performance overhead.

Clover leverages recent advances on a sensor-based soft error detection technique. It detects a soft error by sensing the acoustic wave generated by a particle strike rather than the consequence (e.g., program crash), thereby causing no direct performance penalty [34, 35]. For soft error recovery, Clover combines this soft error detection technique with idempotent processing [8, 9, 11]. The compiler generates idempotent code regions, the re-execution of which does not change the output of the regions. Such side-effect-free re-execution enables Clover to correct errors occurred in a region by simply jumping back to its beginning without explicit checkpoint, provided they are detected within the same region. However, naively combining the sensor-based soft error detection and the idempotent processing does not automatically guarantee correct program execution.

Curse of DUE The crux of the problem is that the sensor-based soft error detection incurs a certain detection latency. It can be minimized at the expense of adding more sensors (i.e., chip area overhead). Therefore, in practice there would be non-negligible error detection latency. Unfortunately, this makes it possible for a

LCTES'15, June 18–19, 2015, Portland, OR, USA.
Copyright © 2015 ACM 978-1-4503-3257-6//...$15.00.
http://dx.doi.org/10.1145/2670529.2754959

soft error to be detected across idempotent regions, which leads to DUE. For example, an error occurring in one idempotent region ends up being detected in the next idempotent region. Thus, simply re-executing the idempotent region will not correct those errors that have occurred in the previous region(s) but were detected in this idempotent region whose inputs (live-in) may have been corrupted by the errors.

To overcome this challenge, Clover intelligently augments these techniques with the instruction level dual modular redundancy (DMR) where instructions are duplicated, verified and intertwined with the original instructions. In this approach (referred to as DMR hereafter), the compiler inserts checks to determine if the original instructions and their duplicated copies have the same computed values at certain synchronization points in the combined code for error detection [28]. In general, this approach can achieve zero detection latency since the checks instantly identify soft errors. However, such an advantage comes with the significant overheads in terms of performance and power, due to the increased instruction count.

Tail-DMR To achieve low overheads, Clover attempts to minimize the use of DMR by exploiting sensor-based soft error detection. The idea is that as long as the error is detected in the same idempotent region, its re-execution can correct the error. In light of this, this paper proposes tail-DMR where the compiler delineates a boundary in each region to break it into two parts: (head and tail). The first part (head) relies on soft error detection via sensors while the second part (tail) on DMR to detect errors, i.e., tail-DMR. This paper calls such a boundary *tail-DMR frontier*. In particular, the compiler determines the frontier so that the DMR-enabled part (i.e., tail) has to be as long as the worst-case sensor-based error detection latency. This ensures that all soft errors are detected in the same region, enabling re-execution of idempotent regions to guarantee correct execution. Since the detection latency is typically small as shown in Section 2, the length of the DMR-enabled part can also be small, and hence execution of the DMR-enabled part will incur only low overhead; Section 4 investigates the trade-off between the sensor area overhead and performance penalty caused by the DMR execution. Consequently, Clover can transparently provide soft error resilience without significant resource consumption and performance degradation. The following are the contributions of this work:

- This paper proposes a novel technique to detect and recover soft errors with low performance overhead. This is the first technique to exploit the advantages of idempotent processing, dual-modular redundancy and sensor-based support for detecting soft errors, toward achieving this goal. We show that Clover intelligently combines these techniques and offsets the drawback of each technique to provide a low-cost and low-overhead mechanism against soft errors. It neither requires microarchitecture modification nor occupies additional cores.
- This paper explores and quantifies the trade-offs in exploiting sensor-based support for soft error detection, instruction-level DMR, and idempotent processing. We show that these trade-offs yield a practical design point for Clover to be applied in real-world scenarios.
- Finally, the evaluation shows that Clover can detect and recover from soft errors without significant performance degradation. Clover incurs an average performance overhead of 26% for a range of Mediabench applications which is a 75% reduction compared to that of the state-of-the-art approach. Moreover, unlike prior work, Clover does not increase code size significantly, which is particularly important for embedded systems.

2. Background

Sensor-Based Soft Error Detection Recently, researchers have proposed a new approach that detects the actual particle strike rather than its consequence (i.e., the program crash, hang or incorrect output) [34, 35]. For example, Upasani *et al.* [34] deploy a set of acoustic wave detectors with cantilevers on silicon and propose techniques to precisely detect the particle strike without requiring redundant micro-architectural structure.

Clover relies on this kind of sensor-based soft error detection scheme to correctly execute a program in the event of soft errors. The error detection latency determines how long the tail part of idempotent regions (tail-DMR) should be to guarantee that its execution time is greater than the detection latency. Thus, the length of the DMR-enabled part is subject to the error detection latency, i.e., a lower soft error detection latency allows a shorter DMR-enabled part (lower performance overhead) but at the expense of more sensors on the chip (higher area overhead).

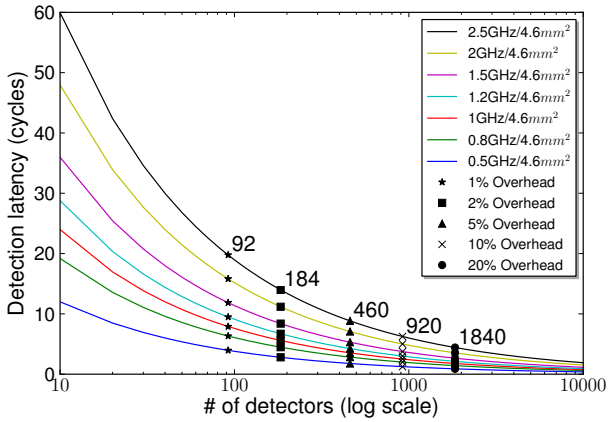

Figure 1. Soft error detection latencies varying the number of sensors under the configurations of ARM cortex-A9 out-of-order processors where the core part takes a quarter of total die size.

Error Detection Latency Exploration To this end, this paper investigates possible detection latencies on various processor configurations to find an appropriate detection latency with an acceptable area overhead. Figure 1 shows different detection latencies for ARM cortex-A9 out-of-order processors. Leveraging data presented by Upasani et al. [34], the detection latency was calculated for a 25% core area ratio to the total die size [1]. Given a total die size (4.6 mm^2) across different clock frequencies (0.5∼2.5 GHz), we vary the number of sensors to understand how the resulting detection latency changes. In the curves of Figure 1, we show several interesting points to represent how many sensors can be deployed within different area overhead budgets, i.e., 1% , 2%, 5%, 10% and 20%. We make two major observations:

- A short error detection latency can be achieved without increasing the area overhead significantly, e.g., detection latency of 5 cycles can be achieved only by increasing the die size by 1% with a 0.5 GHz frequency.
- As expected, lower clock frequency translates to shorter error detection latency, i.e., if NTV-like voltage scaling, which inevitably decreases the clock frequency, is used to improve energy efficiency, the resulting error detection latency will be much shorter. This exactly fits the philosophy of Clover, since

[1] The core part area excludes L1 and L2 caches.

it mainly targets NTV-enabled embedded systems that are particularly vulnerable to soft errors due to the aggressive voltage scaling with NTV operation.

Based on the exploration, this paper makes the assumption that the sensor-based soft error detection can achieve the worst case detection latency of 5 cycles, i.e., the default configuration of Clover. According to the recent work of Upasani *et al.* [35], it is possible to achieve much lower area overhead with a more careful placement of sensors on the chip.

Idempotent Processing for Soft Error Recovery An idempotent region is a part of program code that can be freely re-executed to generate the same output. Thus, soft error recovery can be achieved by simply jumping back to the beginning of the region. More precisely, a region of code is idempotent if and only if its inputs are not overwritten, i.e., no anti-dependence on the inputs, during the execution of the region. Thus, the inputs to the entry of the region will remain the same within the region, making idempotent regions harmless to be re-executed many times. If some inputs are overwritten within the region, their values do not remain the same as it were at the region entry. Therefore, this makes the re-execution of the region unsafe, i.e., ending up changing the expected output produced by the region. Consequently, it is a requirement for the idempotent execution that the inputs to the regions should never be overwritten during the execution of the region.

With that in mind, researchers propose different techniques for preserving the input as it is at the entry of the region. De Kruijf *et al.* [8, 9] places boundaries to break the memory-level anti-dependence, and leverages register renaming to eliminate the register anti-dependence (i.e., a new pseudo-register is allocated to break the dependence) on the inputs to the region. This enables the idempotence of the regions in an elegant manner without explicit checkpoint but at the expense of increasing the register pressure. Once soft errors are detected in the idempotent region, it can be simply re-executed to recover from the errors.

On the other hand, Feng *et al.* [11] take a different approach to get around the anti-dependence without significant increase of the register pressure. They first identify all the non-idempotent regions and selectively protect some of them by explicitly checkpointing at the region entry those inputs that are overwritten within the region, i.e., all the regions are not protectable. For every protected region, a recovery block is generated to restore the checkpointed values from memory on a fault. Thus, the resulting code size increase might not be acceptable for embedded systems. Finally, the actual recovery process requires a rollback runtime that consults the recovery block.

For complete soft error recovery, Clover extends the technique of De Kruijf *et al.* due to its simplicity (i.e., lack of explicit checkpoint and rollback) and insignificant code size increase.

Fault Model The fault model of Clover exactly follows that of idempotent processing. First, memory, caches, and register file are protected against soft errors, e.g., using error correcting codes (ECC). Many commodity embedded processors have already integrated ECC protection to these components [1]. Second, execution of program is guaranteed to follow its static control flow paths; we assume a low-cost, low-latency solution such as [18]. Third, stores in the region has to be safely buffered as with branch misprediction until the region is verified. Finally, the address generation unit is protected for stores to write in the correct locations. Those four components are widely assumed in the literature on software-based error recovery [8, 9, 28], and there have been many solutions to realize them [18, 24, 27, 28]. All other microarchitectural units remain unchanged and can be protected by Clover. The takeaway is that Clover can protect the processor core including random logic state, that is hard to detect and correct soft errors at low cost. For example, Clover neither occupies additional cores nor require special microarchitecture for error detection. Furthermore, Clover does not even need to duplicate architectural status such as register file (RF) and register renaming table (RAT) for error correction.

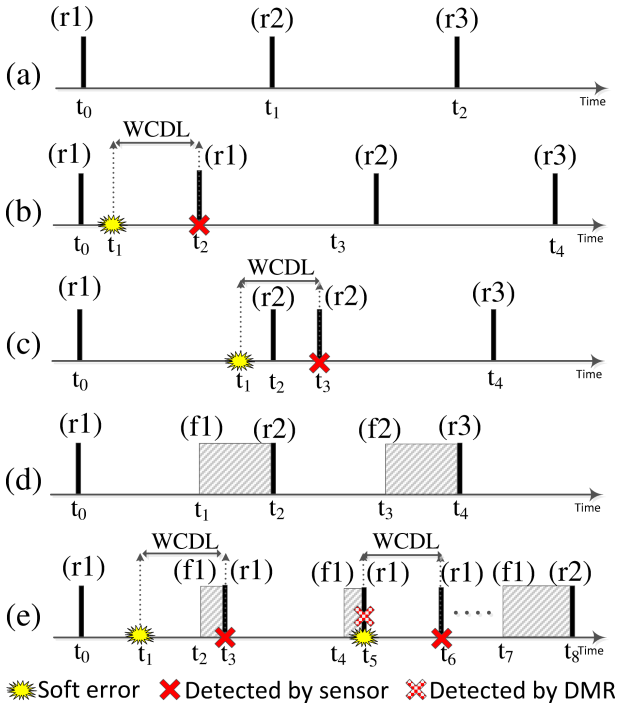

Figure 2. Problem of idempotent processing in the presence of the sensor-based soft error detection scheme and its worst-case detection latency (WCDL), and our tail-DMR solution.

3. Clover Approach

The goal of Clover is to provide a low-cost hardware/software cooperative technique for soft error resilience. Given a reasonable amount of sensors and the resulting detection latency, Clover exploits a novel selective instruction duplication technique called tail-DMR (dual modular redundancy), to eliminate DUE (detected unrecoverable errors) caused by the sensing latency of error detection. For soft error recovery, Clover leverages idempotent processing. Once an error is detected, Clover recovers from it by re-executing the region where it is detected. This error recovery process is performed as in the case of an exception, the handler of which simply redirects program control to the beginning of the region.

Achieving Complete Soft Error Recovery Although the merits of sensor-based soft error detection scheme and idempotence-based recovery scheme look complementary to each other, simply combining them together cannot always achieve correct soft error recovery. As we illustrate next, soft errors may still corrupt the architectural state of the processor core and these schemes can not recover such soft errors correctly. We also show in Section 4 that such a naive combination of both the schemes ends up leaving considerable portion of dynamic instructions susceptible to soft errors.

To illustrate this, Figure 2 describes the idempotence-based recovery scheme and highlights its limitation in the presence of the sensor-based soft error detection scheme and its worst-case detection latency (WCDL). Figure 2 (a) shows the original program execution timeline. Here, vertical bars indicate idempotent region boundaries during program execution, thus there are three regions

(i.e., `r1, r2, r3`) on each timeline. Figure 2 (b) represents an ideal case where the idempotent region can recover correctly from a soft error. At time `t1`, an energetic particle strikes the processor and corrupts the architectural state. After the time of WCDL, the detection scheme causes an exception for the system to initiate the recovery process. Due to the idempotence of the region, the system can recover from the soft error by simply jumping back to the most recent region boundary, i.e., the beginning of the current region (i.e., `r1`) where the error is detected. Note that the region `r1` is restarted at time `t2` on the timeline.

In contrast, Figure 2 (c) demonstrates how the WCDL can make an error go uncorrected even in the presence of the idempotence-based recovery scheme. Suppose an energetic particle strikes the processor at time `t1`. After as much time has passed as the WCDL (i.e., at `t3`), the detection scheme causes an exception for the soft error. However, the system jumps back to the most recent region boundary (i.e., `r2`) instead of `r1` due to the worst-case detection latency. Hence, the error escapes from the former region thereby corrupting the architectural states of the processor and possibly causing a program crash/hang/silent data corruption. This is referred to as the detected unrecoverable error (DUE).

3.1 Tail-DMR

To overcome this challenge, this paper proposes to utilize reasonable amount of sensors (thereby, reducing the chip area overhead) and to selectively duplicate those instructions that are under a risk of DUE in the tail of an idempotent region (thereby, reducing the runtime overhead while maintaining the correct error recovery in all cases). This paper calls such a selective instruction duplication as *tail-DMR*. For a given small number of sensors and the resulting detection latency, the compiler delineates a boundary in each region to break it into two parts, head and tail; the sensor-based detector identifies the errors occurred in the head of region while the DMR identifies those occurred in the tail. We call such a boundary *tail-DMR frontier*. Figure 2 (d) shows the program execution timeline after delineating the tail-DMR frontiers, where f1 and f2 represent the tail-DMR frontiers of r1 and r2, respectively. The shaded zones in the figure are protected by the tail-DMR for soft error detection. Figure 2 (e) describes how the proposed tail-DMR prevents the DUEs. While an error can take place outside the shaded zone at time `t1`, it can be detected still within the current region after WCDL (i.e., at time `t3`). Hence, the recovery scheme can safely redirect the program control to the beginning of the region (i.e., `r1`), thereby ensuring correct recovery from the error.

On the other hand, when an error occurs within the tail of a region (i.e., at time `t5` of Figure 2 (e)), the DMR immediately detects the error, and the re-execution of region `r1` can correctly recover from the error. After the time of WCDL (i.e., at `t6`), the error is detected once more by the sensor causing an exception. Thus, the program control is redirected to the beginning of the most recent region `r1` again. Note that this does not harm the program correctness due to the side-effect-free nature of the idempotent region. Moreover, since a soft error occurs once in a while, the overhead of such redundant recovery will not have a negative impact on the performance.

To guarantee that each soft error occurred in each region must be detected within the same region, Clover carefully determines the *tail-DMR frontier* so that the execution time of the DMR-enabled part (i.e., tail of the region) is longer than the length (time) of the WCDL. This is required to prevent errors from escaping the region, where they occur, without being detected. As a result, the idempotence-based recovery scheme can always correctly recover from them by re-executing the region. Again, if errors occurring in past regions remain uncorrected in the current region, re-executing it cannot achieve the recovery. However, we show that the design of Clover never allows such a case:

Theorem 1. *Given a tail-DMR frontier that makes the execution time of the DMR-enabled part longer than the time of WCDL, all the errors occurred in each region are detected in the same region.*

Proof. We provide the proof by contradiction. Suppose the argument is false, i.e., for an error occurred in the current region R_c, the error is not detected in the current region R_c . Since a region is divided into two parts by the tail-DMR, there are two possibilities for the assumption.

- The error took place in the tail of R_c. This directly contradicts the assumption, since the DMR detects all the errors that occurs in the tail of the region. That is, if the error occurs in the DMR-enabled part (i.e., tail) of R_c, the error must be detected by R_c. This is a contradiction to the assumption.
- The error took place in the head of R_c. According to the tail-DMR frontier, the DMR-enabled part of (i.e., tail) R_c takes longer than the time of WCDL. Therefore, the error is to be identified by the sensor-based detector before the tail of R_c finishes. That is, it is impossible for the error to escape from R_c. This is another contradiction to the assumption.

Therefore, Theorem 1 must be true. □

Theorem 2. *Given idempotent processing, all the errors that take place in each idempotent region are corrected before the region finishes.*

Proof. We omit the proof due to the page limitation. It can be trivially proved by induction using Theorem 1. □

Intuitively, Theorem 1 means that all the errors occurred in a region should be detected before the region finishes, while Theorem 2 provides a strong guarantee that when program control enters a new region, all the errors that previously occurred must have already been correctly recovered. Thus, no error can escape from the region where they take place without being detected and corrected. Clover exploits these theorems as a basis for the idempotent processing to successfully recover from all the errors occurring in each region by re-executing it.

In particular, if a region is so small that all its instructions should be protected by DMR (i.e., the tail-DMR frontier is set to the beginning of the region), the sensor may detect an error occurred in the region after it is finished. However, at this moment, the error must have already been corrected based on Theorem 2. Therefore, re-executing the most recent region (not the region where the error occurred) does not break the program correctness due to the side-effect-free nature of the idempotent region.

Consequently, the tail-DMR enables idempotent processing to correctly recover from all the errors detected in an arbitrary region by simply jumping back to the beginning of the most recent region boundary, i.e., the beginning of the current region where the error is detected. The takeaway is that Clover can eliminate detected unrecoverable errors (DUE).

3.1.1 Clover Compiler Overview

Clover performs detailed compiler analyses to protect an entire idempotent code region against soft errors. Clover introduces additional compiler backend passes to generate soft error tolerant code. Figure 3 shows the compilation workflow of Clover. Once the compiler frontend translates source code into LLVM intermediate representation (IR), Clover applies traditional compiler optimizations on the IR. The optimized IR goes through idempotent region formation passes so that the region becomes re-executable without any

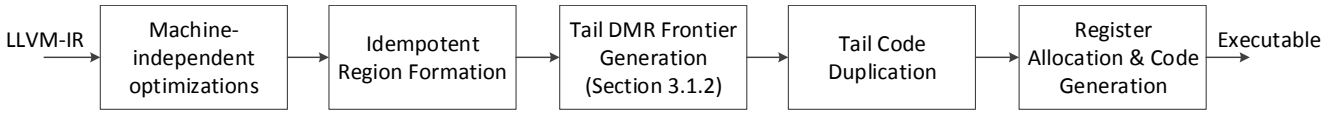

Figure 3. Clover Compiler Workflow

side effect. At the end of this stage, the LLVM IR is lowered to the machine-specific IR, i.e., instruction selection has already been done. Then, the compiler computes the tail-DMR frontier of an idempotent region, and performs the tail-DMR to selectively duplicate necessary instructions and insert compulsory checking instructions for complete error recovery of the region with no DUE (detected unrecoverable error). Finally, the compiler performs register allocation and runs the rest of backend passes to emit an executable. This section focuses on elaborating the tail-DMR frontier generation pass.

The tail-DMR frontier pass is designed to recognize those instructions that are vulnerable to DUE in the tail of an idempotent region. As mentioned in Theorem 1, it is essential that the frontier must be properly set for the execution time of the DMR-enabled part to be longer than the time of WCDL. To achieve this, Clover conservatively represents the time in terms of the number of instructions to be executed. The time of WCDL is conservatively approximated as the product of the WCDL and the commit-width of processor's pipeline, which is called $Threshold_{WCDL}$; if the WCDL is 5 cycles and the commit-width is 2, the tail-DMR forms the DMR-enabled part with only 10 instructions. Thus, Clover can match the time of WCDL by simply counting instructions starting from the end of an idempotent region. Note that the instruction counting should be applied to the resulting instructions of the DMR, not to the original instructions. That is, during the backward traversal of a region, Clover should increase the count not just for the original instruction but also for the duplication and the check instructions to be inserted for DMR, as if the region has already been transformed by the next tail code duplication pass shown in Figure 3. For this purpose, the tail-DMR frontier pass leverages a cost model of DMR which categorizes the instructions of each region as follows:

Synchronization Instructions Originally, store and control flow instructions fall into this category. They require equivalence verification to detect soft errors, e.g., for store instructions, the compiler inserts check instructions to ensure the correctness of the store operands. In particular, the tail-DMR considers a region boundary as a new synchronization point. This is necessary to prevent the errors occurring in the tail of a region from escaping to following regions. That is, any live-out registers, that are defined in the tail of the region, are required to have check instructions before the region boundary. We define a synchronization instruction set as the three types of instructions, and denote the set and the cost of the instructions as $\mathbb{SYN}$ and C_{syn}, respectively. The value of C_{syn} is one, thus it does not include the cost of check instructions which is modeled separately.

Duplication Instructions These instructions are supposed to be duplicated by the compiler thus generating one additional instruction in the region. Note that these instructions are only in the tail of the region. Unlike traditional DMR approaches, Clover duplicates all the instructions from the tail-DMR frontier to the end of the region. Again, all the synchronization instructions will not be duplicated. The cost of the duplication instructions is denoted as C_{dup}. In general, the cost is two; one for the original instruction and the other for the duplication instruction. In particular, C_{dup} of PHI instructions is zero, since the compiler eliminates them in the step of static-single-assignment (SSA) deconstruction for register allocation.

Safe Instructions These instructions are in the head of the region, preceding the tail-DMR frontier. In particular, they are not vulnerable to DUE as long as the tail-DMR is correctly applied. Every error occurring in these instructions will be detected within the region (i.e., before it finishes). Thus, safe instructions are never be duplicated, i.e., there is no cost associated with them.

Check Instructions These instructions are supposed to be inserted to verify the operands and the *live-out* value of the instructions or the region boundaries. This is basically equivalence checking of the values in the original instruction and the duplication; Section 3.1.2 illustrates the insertion of check instructions in more detail. The cost of check instructions is denoted as C_{ck}. We define a check instruction set as those instructions whose defined register needs to be verified at synchronization points, and denote the set and the cost of check instructions as $\mathbb{CK}$ and C_{ck}, respectively. Note that the C_{ck} depends on underlying architecture. For example, the C_{ck} would be two for ARMv7-A instruction set; one instruction for the compare instruction that updates the condition register, and the other one for the branch instruction that transfers the control flow based on the result of the condition register. The next section presents the detailed algorithm of the tail-DMR frontier computation leveraging the cost model.

3.1.2 Region-based Vulnerability Analysis: Computing the Tail-DMR Frontier on SSA

Clover iterates the instructions of an idempotent region backward from its end by traversing the control data flow graph (CDFG). For counting the instructions to match the time of WCDL for each path, Clover consults the cost model of each visited instruction to appropriately increase the count (i.e., path cost) depending on how the tail-DMR treats the instruction. If the count reaches a threshold that represents the time of WCDL, then Clover adds the last visited instruction to the tail-DMR frontier of the region. Before discussing the details, we define the following terms and notations that are used throughout this section.

- $\mathbb{VR}$: Vulnerable register set includes the registers whose value may corrupt the architectural state if it is not verified.
- $\mathbb{CK}$: Check instruction set denotes the instructions whose defined register needs to be verified during the tail-DMR.
- $\mathbb{TF}$: A set of instructions that belong to Tail-DMR frontier.
- $\mathbb{PATH}$: A set of visited instructions along one path during the reverse depth-first-search traversal.
- C_{path}: Accumulated path cost of visited instructions on the current path. If the cost reaches $Threshold_{WCDL}$, then the last-visited instruction is added to the tail-DMR frontier.
- $\mathbb{KILL}$: A function that maps each defined register to a set of instructions that *kill* the register.

Algorithm 1 describes how to compute the tail-DMR frontier. Starting from each region boundary in the CDFG, Clover traverses every path in a reverse depth-first-search (RDFS) order (line 25 $\sim$ 40). Each path is first initialized and keeps track of its own set of

Algorithm 1 Region-based Vulnerability Analysis

Inputs: **CDFG**, $\mathbb{SYN}$, $\mathbb{R}$ (i.e., a set of idempotent regions)
Outputs: $\mathbb{TF}$, $\mathbb{CK}$

```
1: function UpdateVulRegSet(𝕍ℝ, I, ℙ𝔸𝕋ℍ𝕀)
2:     // DefR is a register defined by I
3:     // 𝕂𝕀𝕃𝕃[DefR] is the set of instructions that kill DefR.
4:
5:     if I ∈ 𝕊𝕐ℕ then
6:         𝕍ℝ ← 𝕍ℝ + I's use registers
7:     else if 𝕂𝕀𝕃𝕃[DefR] ⋂ ℙ𝔸𝕋ℍ𝕀 = ∅ then
8:         𝕍ℝ ← 𝕍ℝ+DefR
9:     end if
10: end function
11:
12: function calculateCost(C, I)
13:     if I ∈ 𝕊𝕐ℕ then
14:         C ← C + C_syn
15:     else
16:         C ← C + C_dup
17:     end if
18:     if DefR ∈ 𝕍ℝ then // DefR is a register defined by I
19:         C ← C + C_ck
20:         ℂ𝕂 ← ℂ𝕂 + I
21:     end if
22:     Return C
23: end function
24:
25: for each region boundary R ∈ ℝ in CDFG do
26:     for each path P in reverse-DFS order from R do
27:         ℙ𝔸𝕋ℍ𝕀 ← ∅
28:         𝕍ℝ ← ∅
29:         C_path ← 0
30:         for each Instruction I ∈ P do
31:             ℙ𝔸𝕋ℍ𝕀 ← ℙ𝔸𝕋ℍ𝕀 + I
32:             𝕍ℝ ← UpdateVulRegSet(𝕍ℝ, I, ℙ𝔸𝕋ℍ𝕀)
33:             C_path ← calculateCost(C_path, I)
34:             if C_path ≥ Threshold_WCDL ∨ I ∈ ℝ then
35:                 𝕋𝔽 ← 𝕋𝔽 + I
36:                 Terminate path P
37:             end if
38:         end for
39:     end for
40: end for
```

visited instructions ($\mathbb{PATHI}$), vulnerable registers ($\mathbb{VR}$) and path cost (C_{path}) during the RDFS traversal (line 27 $\sim$ 29).

For each visited instruction in the path, Clover updates $\mathbb{PATHI}$, $\mathbb{VR}$, and C_{path}, correspondingly (line 31 $\sim$ 33). To update $\mathbb{PATHI}$, Clover simply inserts the visited instruction I into $\mathbb{PATHI}$. Keeping track of visited instructions is beneficial for analyzing the liveness of a register in static-single-assignment (SSA) form. Then, to update the $\mathbb{VR}$, Clover leverages a heuristic shown in line 1 $\sim$ 10. If the visited instruction belongs to $\mathbb{SYN}$ set, Clover adds all their used registers to the $\mathbb{VR}$ set since they are vulnerable. In the next pass (i.e., the tail code duplication in Figure 3), necessary check instructions are therefore inserted to the original CDFG for verifying the registers.

In particular, Clover does not need to protect those registers that are live-in at the tail-DMR frontier. As discussed in the proof of Theorem 1, all the errors occurring before the tail-DMR frontier should be dealt with by the sensor-based soft error detection within the region, i.e., its re-execution can correctly recover from the errors. This allows Clover to safely assume that all the live-in registers at the tail-DMR frontier are resilient against soft errors. To this end, the next code duplication pass does not insert check instructions for such live-in registers even if they belong to the $\mathbb{VR}$ set.

Recall that Clover considers the region boundary as a synchronization point. That is, every live-out register at the end of the region must be verified by the region if the live-out register is defined within the tail-DMR frontier. The line 7 of Algorithm 1 shows how Clover can easily compute such a live-out register on the SSA form. Suppose $\mathbb{KILL}$[DefR] is the set of instructions that kill the register DefR. Then, the intersection of $\mathbb{KILL}$[DefR] and $\mathbb{PATHI}$ represents the liveness of DefR at the end of the region along the path. If the intersection is empty, i.e., the DefR is live-out, it is added to $\mathbb{VR}$ for verification.

To calculate the path cost (i.e., C_{path}), Clover just accumulates the cost of visited instructions which is determined differently depending on whether the instruction is in $\mathbb{SYN}$ set or it is a duplication instruction (line 13 $\sim$ 17). In addition, if the register defined by the instruction I belongs to $\mathbb{VR}$ set, a check instruction cost (C_{ck}) is added to C_{path} (line 18 $\sim$ 21). Accordingly, the instruction I is added into $\mathbb{CK}$ set to inform the next code duplication pass of where check instruction needs to be placed.

In line 34 $\sim$ 37, Clover terminates one path if C_{path} reaches the $Threshold_{WCDL}$, i.e., the time of WCDL is matched. Thus, Clover adds the last-visited instruction to the tail-DMR frontier (line 35). Currently, the default values of the WCDL and the commit-width are 5 and 2, respectively, i.e.., the $Threshold_{WCDL} = 10$. In addition, Clover also terminates the path code calculation process when another idempotent region boundary is encountered, i.e., the region is too short. In this case, Clover simply considers the boundary instruction as the frontier, thus all the instructions of such a short region are protected by DMR.

3.1.3 Discussion and Limitation

DUE To eliminate DUE (detected unrecoverable errors), soft errors have to be detected in the same idempotent region for its re-execution to successfully recover from them. This is guaranteed by Clover as shown in the proof of Theorem 1 and 2, i.e., all the idempotent regions are resilient against DUE. Consequently, Clover can achieve zero DUE as long as program is compiled by Clover. On the contrary, naively combining the sensor-based soft error detection and the idempotent processing cannot avoid DUE. Rather, all the regions are potentially vulnerable to DUE regardless of the length of the region.

SDC Even if an energetic particle strike is the major source of soft errors, they can also be induced by other sources, e.g., random noise such as inductive/capacitive crosstalk and power supply noise. Since these sources are not covered by the sensor-based soft error detection, Clover might generate silent data corruption (SDC).

Multiple Soft Errors Occurring in One Region They can be easily handled by Clover. As stated in Section 3.1, all the errors are guaranteed to be detected and corrected within the same region.

Tail-Wait One might think of waiting for the time of WCDL as an alternative to Tail-DMR. However, as the next evaluation section shows, many regions are very short, i.e., the waiting time is relatively long compared to the execution time of such short regions. The runtime overhead makes the waiting approach less attractive, but it would be worth exploring the approach due to the energy efficiency. We leave this as future work.

4. Evaluation

We implement the compiler passes of Clover on top of LLVM Compiler Infrastructure [19]. The idempotent region formation algo-

rithm is also integrated in LLVM. We perform the experiments with 17 applications from Mediabench [20] and MIbench [12] benchmarks in different categories. All the applications were compiled with standard -O3 optimization. We conduct our simulations on Gem5 [3] with system call emulation mode for a modern 2-issue out-of-order 0.5 GHz processor whose L1 (2-way/2-cycles) and L2 (8-way/20-cycles) LRU caches are 32KB and 2MB, respectively. The pipeline widths are all 2 including commit-width, and the ROB and physical integer RF have 128 and 256 entries, respectively.

We first analyze the length of idempotent regions, since it is a critical factor that affects the performance of Clover; in general, the longer region, the better performance. For example, in longer regions, the portion of the DMR-enabled part is relatively small, whereas in short regions, the majority of their instructions have to be duplicated by DMR. Then, we analyze the execution time overhead of Clover comparing it to the state-of-the-art technique, i.e., combination of idempotent processing and full-DMR [9]. Finally, we provide sensitivity analysis results to understand the trade-off between the sensor area overhead and the resulting performance of Clover.

4.1 Region Characteristics

Application	#Total insts (10^3)	#Total regs (10^3)	Aver. leng.	#Vulner. insts (10^3)	Vulner. insts ratio
adpcmdecode	6557	149	44	1486	22.66%
adpcmencode	8346	223	37	1231	14.75%
epic	59759	1186	50	8370	14.00%
unepic	9898	941	10	6847	69.17%
jpegdecode	4382	280	15	2252	51.39%
jpegencode	17335	1205	14	9335	53.85%
mesatexgen	204820	8497	24	68748	33.56%
pegwitencrypt	35616	2600	13	18586	52.18%
g721decode	512016	14239	35	94027	18.36%
g721encode	268789	7766	34	51155	19.03%
gsmdecode	68406	2270	30	19625	28.68%
gsmencode	110750	3350	33	27756	25.06%
mpeg2decode	165491	5792	28	49946	30.18%
mpeg2encode	1320760	17867	73	143323	10.85%
sha	120338	1121	107	9530	7.91%
susanedges	78967	2190	36	15526	19.66%
susancorners	27265	455	59	2823	10.35%
geomean	58368	1822	31	13736	23.53%

Table 1. Dynamic region characteristic with 5-Cycle-WCDL

Figure 4 (a) shows a cumulative distribution of dynamically executed idempotent regions of all the applications listed in Table 1. The x-axis (in log scale) represents the number of instructions in regions. We highlight *unepic* and *adpcmdecode*. As shown in Figure 4 (b), the majority of regions in *unepic* are comprised of less than 10 instructions, and they occupy considerable amount of the total execution time. The implication is that the tail-DMR will cause significant performance overhead for *unepic*. In contrast, *adpcmdecode* has many long regions as shown in Figure 4 (c). That is, most of the regions are long enough to hide the performance penalty caused by the tail-DMR, thus it will cause negligible performance overhead for *adpcmdecode*.

Table 1 further details on the dynamic region characteristics of the benchmark applications. Column 2 and 3 show the dynamic instruction count and the number of idempotent regions executed, respectively. Column 4 presents the average region length, i.e., (2nd column/3rd column). The geometric mean of the average region length is 31, which makes it possible for the tail-DMR to be practically realized without causing significant overhead. Column 5 and 6 represent the total number of vulnerable instructions and the ratio (i.e., 2nd column/5th column), respectively. Note that simply relying on the naive combination of the idempotent processing and sensor-based detection scheme leave all the regions potentially vulnerable to DUE. On average, total 23% of dynamic instructions are vulnerable to soft errors. This indicates that only a small portion of instructions need to be protected by the tail-DMR. More precisely, the portion is almost cut in half in that the number of instructions are doubled after the DMR is performed, i.e., not all the vulnerable instructions are protected by Clover. This is because the tail-DMR inserts duplication and check instructions to the original region, which allows some vulnerable instructions to be placed beyond the tail-DMR frontier. Consequently, Clover can achieve very low performance overhead.

4.2 Performance Overhead and Code Size

Figure 5 represents the runtime overhead of different soft error resilience schemes, which is normalized to the baseline execution time with no resilience scheme. For each application, the first bar corresponds to the runtime of the state-of-the-art scheme [9] where full-DMR is combined with idempotent processing, while the second bar to the runtime of Clover. In the figure, each bar is broken into two parts; the bottom and the top represent the overheads of error detection and recovery, respectively. For example, the top parts of the first and the second bars (i.e., *Idem-W/-FullDMR* and *Idem-w/-TailDMR*) represents the overheads due to idempotence-based error recovery in the presence of full-DMR and Clover's tail-DMR, respectively. As shown in Figure 5, the idempotence-based recovery is not that significant i.e., on average 14% (*Idem-w/-FullDMR*) and 8% (*Idem-w/-TailDMR*. Thus, most of the overhead is caused by the error detection schemes, i.e., on average, 91% (*FullDMR*) and 18%(*TailDMR*). Overall, the full-DMR with with idempotent processing incurs 105% runtime overhead on average. In contrast, Clover incurs only 26% runtime overhead on average, which is a 75% reduction, at the expense of only 1% chip area overhead.

It is important to note that prior approaches [9, 10] do not duplicate load instructions since they assume a fault-tolerant load unit. However, this is not likely the case for embedded systems, and they will miss soft errors occurring from a load unit without the shielding. With that in mind, we duplicate load instructions in *FullDMR* and *TailDMR*. If Clover assumed such a fault-tolerant load unit, the overhead would be significantly reduced. Figure 5 also confirms that the length of regions is critical to Clover's performance overhead (i.e., 2nd bar). The general trend is that the higher ratio of vulnerable instructions shown in Table 1 translates to higher performance overhead.

Table 2 summarizes the code size increase of the full-DMR with idempotent processing versus Clover. The number of additional static instructions inserted to the original program is represeted in Column 2 (the full-DMR approach) and Column 3 (Clover). Column 4 shows the reduction Clover achieves as percentage compared to the full-DMR approach. On average, Clover achieves a static instruction reduction of 46%. With the importance of binary size in embedded systems in mind, we also show the ratio of the binary size increase to the original binary size for the full-DMR approach and Clover in Column 5 and Column 6, respectively. Overall, the average binary size increase of the full-DMR approach is 86%, whereas that of Clover is only 30%. Column 7 shows the reduction Clover achieves as percentage compared to the full-DMR approach. On average, Clover achieves a binary size reduction of 53%.

4.3 Sensitivity Analysis

The worst-case detection latency (WCDL) of the sensor-based soft error detection is another critical factor that affects the performance overhead of Clover. Note that the sensor-based detection scheme can adjust its WCDL by varying the amount of sensors being deployed and their placement on the processor core. As illustrated in section 2, larger amount of sensors will result in higher area overhead. To highlight the trade-off, we perform a sensitivity analysis of WCDL to the resulting performance overhead shown in Figure 6.

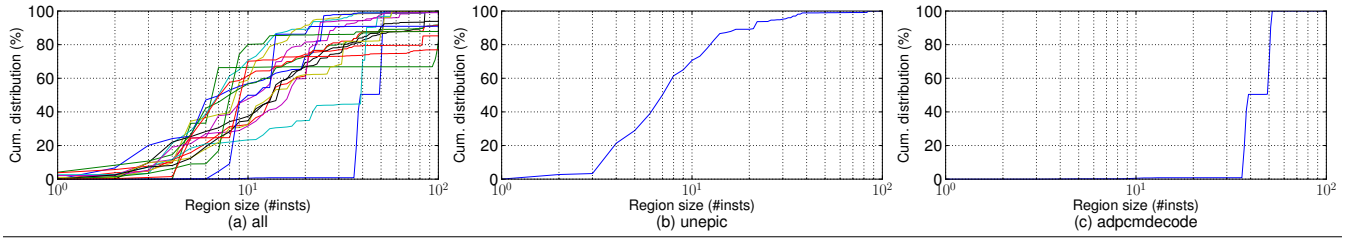

Figure 4. The distribution of the original idempotent code regions (dynamic)

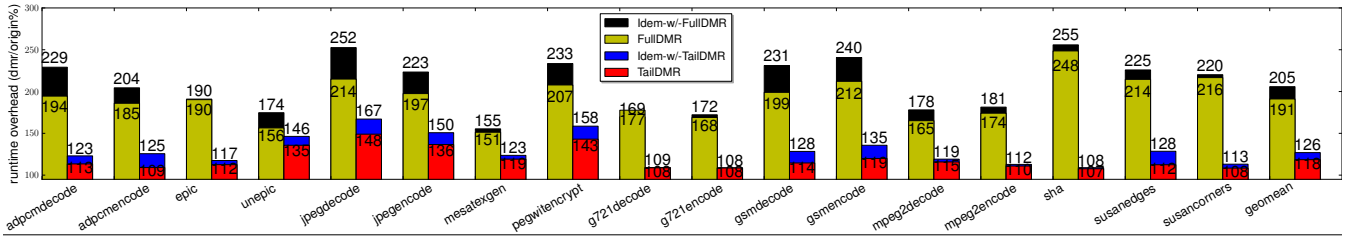

Figure 5. Performance overhead of Clover vs. Full-DMR

Application	#Full DMR insts	#Tail DMR insts	Insts reduction	Full DMR binary size increase ratio	Tail DMR binary size increase ratio	Binary size reduction
adpcmdecode	408	146	64.21%	47.24 %	1.34 %	97.17%
adpcmencode	411	146	64.47%	47.24 %	1.34 %	97.17%
epic	9314	5443	41.56%	106.54%	58.96 %	44.66%
unepic	7179	5210	27.42%	108.46%	69.70 %	35.74%
jpegdecode	52209	34728	33.48%	124.79%	82.13 %	34.19%
jpegencode	50282	33326	33.72%	130.77%	79.86 %	38.93%
mesatexgen	183345	104805	42.83%	128.74%	68.18 %	47.04%
pegwitencrypt	12297	7144	41.90%	96.63 %	50.01 %	48.24%
g721decode	2524	1393	44.80%	66.87 %	34.81 %	47.95%
g721encode	2659	1481	44.30%	68.02 %	35.88 %	47.26%
gsmdecode	10687	5490	48.62%	68.65 %	31.20 %	54.55%
gsmencode	10687	5490	48.62%	68.65 %	31.20 %	54.55%
mpeg2decode	15884	9580	39.68%	107.01%	58.29 %	45.53%
mpeg2encode	23680	11528	51.31%	112.55%	49.24 %	56.25%
sha	857	407	52.50%	61.02 %	26.05 %	57.30%
susanedges	7187	2522	64.90%	102.50%	33.03 %	67.77%
susancorners	7187	2522	64.90%	102.50%	33.03 %	67.77%
geomean	7552	3858	46.23%	86.60 %	30.42 %	53.02%

Table 2. Code size comparision: Full-DMR vs. Tail-DMR. *gsmencode* and *susancorners* share the same binaries with *gsmdecode* and *susanedges*, respectively. Therefore, they have the exact same data.

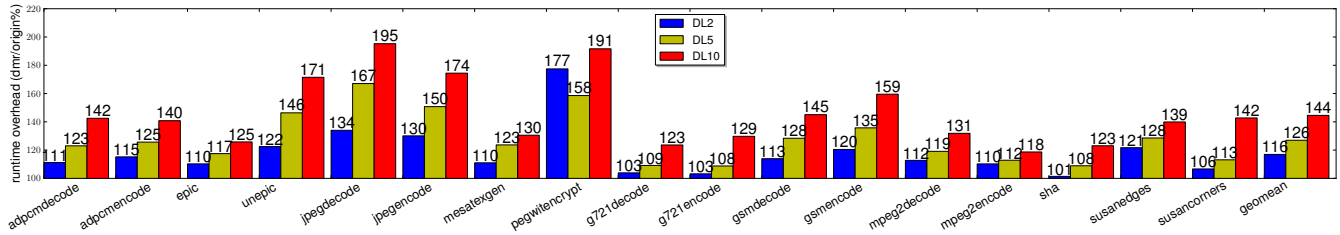

Figure 6. Impact of WCDL change on the performance overhead of Clover

Other than 5-cycles-WCDL (DL5) that is Clover's default configuration, we pick two other practical detection latencies taking into account the resulting chip area increase. Here, DL2 represents 2-cycles-WCDL which can be realized with 2% area overhead for a 0.5 GHz processor as shown Figure 1. Similarly, DL10 stands for 10-cycles-WCDL and can be realized with less than 1% area overhead for a 0.5 GHz processor. As expected, the performance overhead of Clover is proportional to the detection latency except for *pegwitencrypt*. This is mainly due to cache performance variance. Overall, lower detection latency implies higher area overhead as

well as higher hardware complexity, and vice versa. Consequently, Clover can provide a flexible and practical approach to adapt different WCDL settings, and it achieves lightweight soft error resilience with reasonable chip area overhead (i.e., DL5).

5. Other Related Work

Soft Error Detection Soft error detection relies on either hardware or software instruments to identify the errors. Software-based detection schemes often refer to N-modular redundancy execution. SWIFT is one of the state-of-the-art single-threaded software detection schemes [28]. It checks the value of registers with their duplication counterpart at certain synchronization points, i.e., memory and control flow instructions. Rotenberg takes the advantage of simultaneous multithreading (SMT) to run a trailing thread that verifies the leading thread [30]. Although these methods achieve a high fault coverage, they suffer from significant performance overhead or occupying one more processor core.

With that in mind, researchers explore the program characteristics to find opportunities for reducing the performance overhead while still maintaining an acceptable fault coverage [17][6]. They all exploit some heuristics to identify the instructions that are critical to the program output and selectively protect these instructions with DMR. Especially, Khudia and Mahlke leverage the value-locality of instruction results to improve the fault coverage [17]. However, these approaches achieve low performance overhead at the expense of reduced fault coverage. In contrast, this paper selectively protects some of the instructions (i.e., only those instructions vulnerable to DUE) without sacrificing the fault coverage. Chen and Yang propose a technique, that identifies the minimum set of instruction results being compared and checkpointed for the error resilience, to reduce the performance overhead while achieving full coverage [5]. However, the resulting runtime overhead reduction is not stated in their paper.

Hardware-based detection schemes introduce redundant hardware to verify the execution in the processor. DIVA [2] relies on a simple in-order core to verify the program execution while Argus [24] leverages invariant checking to ensure correctness. However, these approaches often introduce excessive hardware complexity increase which is not acceptable in embedded systems. ReStore [37] advocates to utilize symptoms of soft errors to detect them without significant overhead. Shoestring [10] enhances ReStore by selectively duplicating some vulnerable instructions with simple heuristic. However, both ReStore and Shoestring incur long detection latency which may result in DUE. Similar to the recent work of Khudia and Mahlke [17], Racunas *et el* [26] proposes to make use of the value-locality to detect soft errors. However, the high false positive rate and the mediocre fault coverage prevent us from adapting their methods. Fortunately, Upasani *et al.* propose to detect soft errors with configurable amount of acoustic wave sensors [34, 35]. Their sensor-based detection scheme achieves low soft error detection latency with reasonable hardware overhead. According to the recent work of Upasani *et al.* [35], it is possible to detect all the soft errors in an old ARM cortex-A5 core within one cycle by meticulously deploying only 17 sensors on the core. However, their technique requires the core size and the frequency to be extremely small, which is not realistic for modern processors. This paper takes advantage of such sensor-based detection scheme and selectively duplicates only the instruction vulnerable to DUE in the tail of an idempotent region for guaranteed soft error recovery.

Soft Error Recovery Checkpointing the whole program states (memory and registers) guarantees recovery from soft errors by allowing programs to roll back to the previous safe checkpoint[10, 34, 37]. However, full checkpointing often comes with significant performance loss and high power consumption. With that in mind, researchers propose techniques that can reduce the checkpointing overhead, but they require costly hardware support and resource consumption. For example, the recent work of Upasani *et al.* [34] keeps two copies of the register file and the register allocation table (RAT) to achieve low performance overhead. Jeyapaul *et al.* [15] explore multicore CMP architecture to recover from soft errors with an efficiently modified cache structure. However, they rely on only parity checking to sequential logic for detecting a soft error, i.e., combinational logic is still vulnerable to soft errors thus they may generate SDC. Flushing the pipeline to recover from a soft error [26, 35] is another alternative. This approach is expected to be very efficient in term of runtime overhead. However, this approach is often based on the assumption that detection can be done before the faulty instruction is committed, i.e., the error detection latency should be zero. Such low detection latency inevitably requires high performance/hardware overhead as stated in Section 5. In particular, Clover avoids such high overhead by integrating idempotent processing that recovers from soft errors by simply re-executing the region in which they occur. That is, even if soft errors have already corrupted architectural states, Clover can recover from the errors, and the detection latency does not need to be zero. However, idempotent processing requires soft errors to be detected within the same region as stated in Section 3. Clover overcomes such a challenge with a novel tail-DMR technique in the presence of sensor-based soft error detection.

6. Conclusion

This work presents Clover, a compiler directed soft error detection and recovery scheme. This is a fundamentally new approach to achieving lightweight soft error resilience with zero DUE (detected unrecoverable error). Clover is a low-cost hardware/software cooperative scheme. On the hardware side, Clover relies on a small number of acoustic wave detectors deployed in the processor to identify soft errors by sensing the wave made by a particle strike [34, 35]. On the software side, Clover leverages a novel selective instruction duplication, called tail-DMR (dual modular redundancy), to cope with DUE caused by the sensing latency of error detection. In addition, Clover generates soft error tolerant code based on idempotent processing for soft error recovery. Once a soft error is detected, Clover recovers from it by re-executing the idempotent region where it is detected. This error recovery process is performed as in the case of an exception, the handler of which simply redirects program control to the beginning of the region. The experiment results demonstrate that the runtime overhead of Clover is only 26%, which is a 75% reduction compared to that of the state-of-the-art soft error resilience technique.

7. Acknowledgments

The authors would like to thank the anonymous referees for their valuable comments. This work was in part supported by the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is managed by UT Battelle, LLC for the U.S. DOE (under the contract No. DE-AC05-00OR22725).

References

[1] ARM. Cortex-a57 technique reference manual. URL `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0488g/index.html`.

[2] T. M. Austin. Diva: A reliable substrate for deep submicron microarchitecture design. In *Microarchitecture, 1999. MICRO-32. Proceedings. 32nd Annual International Symposium on*, pages 196–207. IEEE, 1999.

[3] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen,

K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, 2011.

[4] J. Carretero, P. Chaparro, X. Vera, J. Abella, and A. Gonzlez. End-to-end register data-flow continuous self-test. In S. W. Keckler and L. A. Barroso, editors, *ISCA*, pages 105–115. ACM, 2009.

[5] H. Chen and C. Yang. Boosting efficiency of fault detection and recovery throughapplication-specific comparison and checkpointing. In *Proceedings of the 14th ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, LCTES '13, pages 13–20. ACM, 2013.

[6] J. Cong and K. Gururaj. Assuring application-level correctness against soft errors. In *Proceedings of the International Conference on Computer-Aided Design*, pages 150–157. IEEE Press, 2011.

[7] C. Constantinescu. Trends and challenges in vlsi circuit reliability. *Microarchitecture, 2003. MICRO-36. Proceedings. 36nd Annual International Symposium on*, 23(4), July 2003.

[8] M. de Kruijf and K. Sankaralingam. Idempotent code generation: Implementation, analysis, and evaluation. In *CGO*, pages 1–12. IEEE Computer Society, 2013. ISBN 978-1-4673-5524-7.

[9] M. A. de Kruijf, K. Sankaralingam, and S. Jha. Static analysis and compiler design for idempotent processing. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 475–486, 2012. ISBN 978-1-4503-1205-9.

[10] S. Feng, S. Gupta, A. Ansari, and S. Mahlke. Shoestring: probabilistic soft error reliability on the cheap. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 385–396. ACM, 2010.

[11] S. Feng, S. Gupta, A. Ansari, S. A. Mahlke, and D. I. August. Encore: low-cost, fine-grained transient fault recovery. In *MICRO'11*, pages 398–409, 2011.

[12] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pages 3–14. IEEE, 2001.

[13] I. S. Haque and V. S. Pande. Hard data on soft errors: A large-scale assessment of real-world error rates in gpgpu. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID '10, pages 691–696, 2010. ISBN 978-0-7695-4039-9.

[14] J. Henkel, L. Bauer, N. Dutt, P. Gupta, S. Nassif, M. Shafique, M. Tahoori, and N. Wehn. Reliable on-chip systems in the nano-era: Lessons learnt and future trends. In *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, pages 99:1–99:10, 2013.

[15] R. Jeyapaul, A. Risheekesan, A. Shrivastava, and K. Lee. Unsync-cmp: Multicore cmp architecture for energy efficient soft error reliability. *Transactions on Parallel and Distributed Systems*, 25(1):254–263, January 2014.

[16] H. Kaul, M. Anders, S. Hsu, A. Agarwal, R. Krishnamurthy, and S. Borkar. Near-threshold voltage (ntv) design-ppportunities and challenges. In *DAC*, pages 1153–1158. ACM. ISBN 978-1-4503-1199-1.

[17] D. Khudia and S. Mahlke. Harnessing soft computations for low-budget fault tolerance. In *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pages 319–330, Dec 2014.

[18] D. S. Khudia and S. Mahlke. Low cost control flow protection using abstract control signatures. In *Proceedings of the 14th ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, LCTES '13, New York, NY, USA, 2013.

[19] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[20] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 330–335. IEEE Computer Society, 1997.

[21] X. Li, M. C. Huang, K. Shen, and L. Chu. A realistic evaluation of memory hardware errors and software system susceptibility. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, 2010.

[22] Y. Luo, S. Govindan, B. Sharma, M. Santaniello, J. Meza, A. Kansal, J. Liu, B. Khessib, K. Vaid, and O. Mutlu. Characterizing application memory error vulnerability to optimize datacenter cost via heterogeneous-reliability memory. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014*, pages 467–478. IEEE, 2014.

[23] R. E. Lyons and W. Vanderkulk. The use of triple-modular redundancy to improve computer reliability. *IBM Journal of Research and Development*, 6(2):200–209, 1962.

[24] A. Meixner, M. E. Bauer, and D. J. Sorin. Argus: Low-cost, comprehensive error detection in simple cores. In *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on*, pages 210–222. IEEE, 2007.

[25] S. S. Mukherjee, J. Emer, and S. K. Reinhardt. The soft error problem: An architectural perspective. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, HPCA '05, pages 243–247, 2005. ISBN 0-7695-2275-0.

[26] P. Racunas, K. Constantinides, S. Manne, and S. S. Mukherjee. Perturbation-based fault screening. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 169–180. IEEE, 2007.

[27] G. A. Reis, J. Chang, N. Vachharajani, S. S. Mukherjee, R. Rangan, and D. August. Design and evaluation of hybrid fault-detection systems. In *Computer Architecture, 2005. ISCA'05. Proceedings. 32nd International Symposium on*, pages 148–159. IEEE, 2005.

[28] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. Swift: Software implemented fault tolerance. In *Proceedings of the international symposium on Code generation and optimization*, pages 243–254. IEEE Computer Society, 2005.

[29] G. A. Reis, J. Chang, and D. I. August. Automatic instruction-level software-only recovery. *IEEE micro*, 27(1):36–47, 2007.

[30] E. Rotenberg. Ar-smt: A microarchitectural approach to fault tolerance in microprocessors. In *Fault-Tolerant Computing, 1999. Digest of Papers. Twenty-Ninth Annual International Symposium on*, pages 84–91. IEEE, 1999.

[31] G. P. Saggese, N. J. Wang, Z. Kalbarczyk, S. J. Patel, and R. K. Iyer. An experimental study of soft errors in microprocessors. *IEEE Micro*, 25(6):30–39, 2005.

[32] M. Shafique, S. Garg, J. Henkel, and D. Marculescu. The eda challenges in the dark silicon era: Temperature, reliability, and variability perspectives. In *Proceedings of the The 51st Annual Design Automation Conference on Design Automation Conference*, DAC '14, pages 185:1–185:6, 2014. ISBN 978-1-4503-2730-5.

[33] M. B. Taylor. Is dark silicon useful?: Harnessing the four horsemen of the coming dark silicon apocalypse. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 1131–1136, 2012. ISBN 978-1-4503-1199-1.

[34] G. Upasani, X. Vera, and A. Gonzalez. Avoiding core's due & sdc via acoustic wave detectors and tailored error containment and recovery. In *ISCA*, pages 37–48, 2014.

[35] G. Upasani, X. Vera, and A. Gonzalez. Framework for economical error recovery in embedded cores. In *On-Line Testing Symposium (IOLTS), 2014 IEEE 20th International*, pages 146–153. IEEE, 2014.

[36] L. Wang and K. Skadron. Implications of the power wall: Dim cores and reconfigurable logic. *IEEE Micro*, pages 40–48, 2013.

[37] N. J. Wang and S. J. Patel. Restore: Symptom-based soft error detection in microprocessors. *Dependable and Secure Computing, IEEE Transactions on*, 3(3):188–201, 2006.