Compiler-Directed Soft Error Detection and Recovery to Avoid DUE and SDC via Tail-DMR

QINGRUI LIU, CHANGHEE JUNG, and DONGYOON LEE, Virginia Tech
DEVESH TIWARI, Oak Ridge National Laboratory

This article 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 checkpoints. 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 DUEs (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) that provides a region-level error containment. Once a soft error is detected by either the sensors 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 experimental 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. In addition, this article evaluates an alternative technique called tail-wait, comparing it to Clover. According to the evaluation with the different processor configurations and the various error detection latencies, Clover turns out to be a superior technique, achieving 1.06 to 3.49× speedup over the tail-wait.

CCS Concepts: • **Computer systems organization** → **Reliability**; • *Redundancy;*

Additional Key Words and Phrases: Soft error resilience, compilers, tail-DMR frontier, idempotent processing, acoustic wave detectors

ACM Reference Format:
Qingrui Liu, Changhee Jung, Dongyoon Lee, and Devesh Tiwari. 2016. Compiler-directed soft error detection and recovery to avoid DUE and SDC via tail-DMR. ACM Trans. Embed. Comput. Syst. 16, 2, Article 32 (December 2016), 26 pages.
DOI: http://dx.doi.org/10.1145/2930667

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 [Luo et al. 2014; Li et al. 2010; Saggese et al. 2005; Constantinescu 2003; Haque and Pande 2010; Mukherjee et al. 2005]. Unfortunately, due to technology scaling, electronic circuits are likely to be more susceptible to radiation-induced soft errors (also known as transient faults). They are typically caused by cosmic rays and alpha particles from packaging material. Soft errors may

This work was partly supported by the National Science Foundation under the grant CCF-1527463, and by the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is managed by UT Battelle, LLC for the US DOE under the contract No. DE-AC05-00OR22725.
Authors' addresses: Q. Liu, C. Jung, and D. Lee, 2202 Kraft Drive, Blacksburg, VA 24060; email: lqingrui@vt.edu; D. Tiwari, 1 Bethel Valley Rd, Oak Ridge, TN 37831.

DOI: http://dx.doi.org/10.1145/2930667

lead to application crashes or, even worse, silent data corruptions (SDCs), which are not caught by the error detection logic but may cause the program to produce incorrect output. Another type of results are detected unrecoverable errors (DUEs) that often directly impact the reliability of the computer systems. To achieve soft error resilience, it is essential to have both the detection and the correction schemes for the soft errors.

In the dark silicon era, soft errors are becoming an 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 [Wang and Skadron 2013; Taylor 2012]. 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 the soft errors. According to Shafique et al. [2014], near-threshold voltage operation may cause up to a 30× higher soft error rate than nominal voltage operation. Similar trends have been observed by other researchers as well [Henkel et al. 2013; Kaul et al. 2012]. 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 the soft errors. Unfortunately, existing techniques often suffer from high performance overhead [Reis et al. 2005b, 2007; Lyons and Vanderkulk 1962] or require costly hardware support and resource consumption (e.g., occupying entire cores or leveraging special microarchitecture) [Carretero et al. 2009; Racunas et al. 2007; Austin 1999; Meixner et al. 2007]. Despite increased hardware, performance, and power costs, these techniques may not eliminate both the SDCs and the DUEs or the need for expensive checkpointing. To address these issues, this article presents Clover, a compiler-directed lightweight resilience scheme that can detect and correct the 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 [Upasani et al. 2014a, 2014b]. For soft error recovery, Clover combines this soft error detection technique with idempotent processing [de Kruijf et al. 2012; de Kruijf and Sankaralingam 2013; Feng et al. 2011]. The compiler partitions and transforms the entire program into different 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 the errors occurring in a region by simply jumping back to its beginning without explicit checkpoints, 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 nonnegligible error detection latency. Unfortunately, this makes it possible for a 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 but were detected in this idempotent region whose inputs (live-in) may have been corrupted by the errors. Worse, the idempotent regions generally have a small region size (see Section 4.1), leading to more DUEs as more errors cannot be detected within the region by the sensors.

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 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 [Reis et al. 2005b]. In general, this approach can achieve zero detection latency since the checks instantly identify the soft errors. However, such an advantage comes with significant overhead in terms of performance and power, due to the increased instruction count.

Tail-DMR. To achieve low overhead, 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 article 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 the sensors, while the second part (tail) relies on the DMR to detect the errors, that is, tail-DMR. This article 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 longer than the worst-case sensor-based error detection latency. This ensures that all the 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 tradeoff 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.

Following are the contributions of this work:

—This article proposes a novel technique to detect and recover the 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 the 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 the soft errors.
—This article explores and quantifies the tradeoffs in exploiting sensor-based support for soft error detection, instruction-level DMR, and idempotent processing. We show that these tradeoffs yield a practical design point for Clover to be applied in real-world scenarios.
—Clover can detect and recover from the 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.
—Clover's tail-DMR turns out to be superior to a new alternative to it called tail-wait that waits at the end of each region for the time of the worst-case error detection latency. For example, the tail-DMR achieves 1.06 to 3.49× speedup over the tail-wait.

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) [Upasani et al. 2014a, 2014b]. For example, Upasani et al. [2014a] deploy a set of acoustic wave detectors with cantilevers on silicon and propose techniques to precisely detect the particle strike without requiring redundant

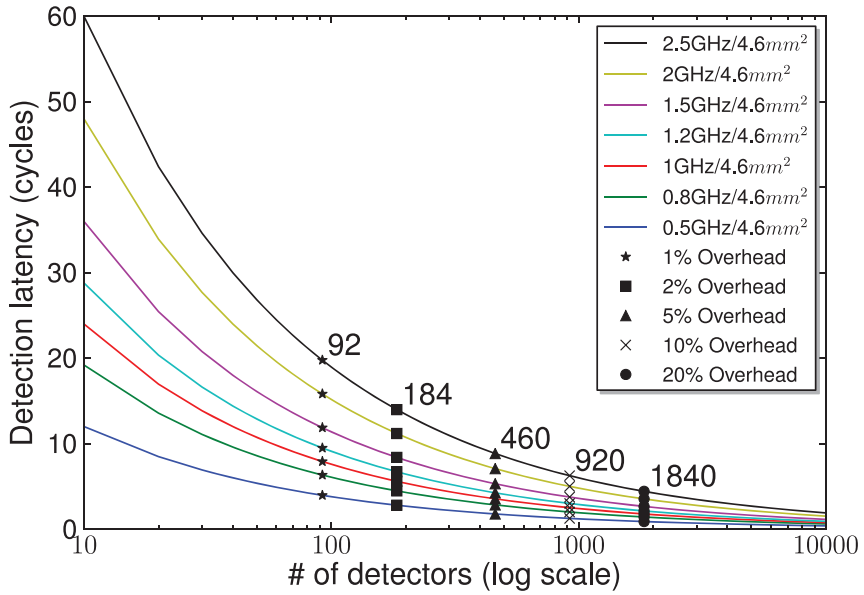

Fig. 1. Soft error detection latencies varying the number of sensors under the configurations of ARM cortex-A9 out-of-order processors where the core part takes a quarter of total die size.

microarchitectural structure. Clover relies on this kind of sensor-based soft error detection scheme.

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; that is, 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).

Error Detection Latency Exploration. To this end, this article 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. [2014a], the detection latency was calculated for a 25% core area ratio to the total die size.[1] Given a total die size ($4.6mm^2$) across different clock frequencies (0.5–2.5GHz), 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, that is, 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; for example, a detection latency of five cycles can be achieved only by increasing the die size by 1% with a 0.5GHz frequency.
—As expected, lower clock frequency translates to shorter error detection latency; that is, 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 it mainly targets

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

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

Based on the exploration, this article makes the assumption that the sensor-based soft error detection can achieve the worst-case detection latency of five cycles; that is, this is the default configuration of Clover. According to the recent work of Upasani et al. [2014b], it is possible to achieve much lower area overhead with a more careful placement of sensors on the chip. Section 4 later evaluates Clover across the different worst-case detection latencies varying the performance of the underlying processor, that is, the pipeline commit-width.

Idempotent Processing for Soft Error Recovery. An idempotent region can be freely re-executed to generate the same output. Therefore, soft error recovery can be achieved by simply jumping back to the beginning of the faulty region and re-executing the region.

Idempotent processing partitions and transforms the entire control flow graph (CFG) of a program into different idempotent regions. That is, the entire program is protected by Clover. Each region contains a single entry basic block and zero or more exiting basic blocks where the entry block dominates all other blocks. Then, all those regions are transformed or instrumented to be idempotent. A region of code is idempotent if and only if its inputs (e.g., the values that are live-in to the region) are not overwritten (i.e., no antidependence 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 they were at the region entry. Therefore, this makes the re-execution of the region unsafe, that is, 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. [2012] and de Kruijf and Sankaralingam [2013] leverage their region partition algorithm to place region boundaries to break the memory-level antidependence and utilize register renaming to eliminate the register antidependence (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 an explicit checkpoint but at the expense of increasing the register pressure. Once the 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. [2011] take a different approach to get around the antidependence without a significant increase of the register pressure. They first identify all the nonidempotent regions and selectively protect some of them by explicitly checkpointing at the region entry those inputs that are overwritten within the region; that is, 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. because of its simplicity (i.e., lack of explicit checkpoint and rollback), complete coverage (i.e., partition the entire program into different idempotent regions), and insignificant code size increase.

Fault Model. The fault model of Clover exactly follows that of idempotent processing. First, memory, caches, and register files are protected against the soft errors, for example, using error-correcting codes (ECC). Many commodity-embedded processors have already integrated ECC protection to these components [ARM 2015]. Second, execution

of the program is guaranteed to follow its static control flow paths; we assume a low-cost, low-latency solution such as Khudia and Mahlke [2013]. Third, instructions must write to the correct register destinations. Fourth, stores in the region have to be safely buffered as with branch misprediction until the region is verified. For this purpose, a gated store buffer is often leveraged [Liu et al. 2016a, 2016b; Liu and Jung 2016]. Finally, the address generation unit is protected for stores to write in the correct locations. Those five components are widely assumed in the literature on software-based error recovery [Reis et al. 2005b; de Kruijf et al. 2012; de Kruijf and Sankaralingam 2013], and there have been many solutions to realize them [Meixner et al. 2007; Reis et al. 2005a, 2005b; Khudia and Mahlke 2013]. 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, which is hard to detect, and correct the soft errors at low cost.

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 to eliminate DUEs 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 raised by either the sensors or tail-DMR, the exception handler of which simply redirects program control to the beginning of the region.

Achieving Complete Soft Error Recovery. Although the merits of the 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, the soft errors may still corrupt the architectural state of the processor core, and these schemes cannot recover such soft errors correctly. We also show in Section 4 that such a naive combination of both the schemes ends up leaving a considerable portion of dynamic instructions susceptible to the 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, that is, 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 DUE.

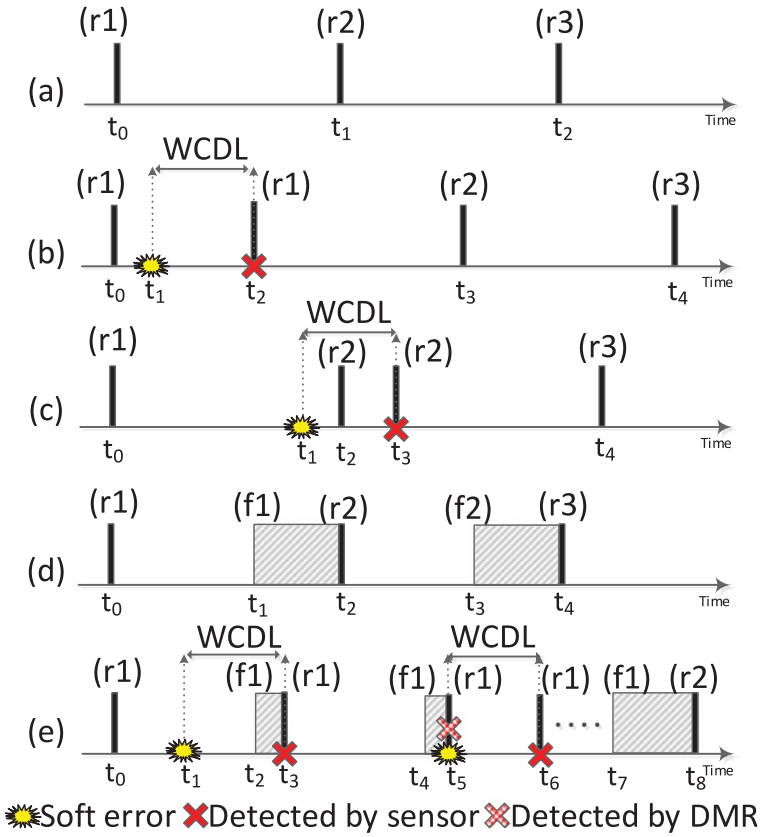

Fig. 2. Problem of idempotent processing in the presence of the sensor-based soft error detection scheme and its worst-case detection latency (WCDL), and our tail-DMR solution.

3.1. Tail-DMR

To overcome this challenge, this article proposes utilizing a reasonable amount of sensors (thereby reducing the chip area overhead) and selectively duplicating 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 article refers to this 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 that occurred in the head of the region, while the DMR identifies those that occurred in the tail. We call such a boundary the *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 (see Section 4.1).

3.1.1. Tail-DMR Frontier for a Region-Level Error Containment. To guarantee that each soft error that 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 the errors from escaping the region where they occur without being detected. As a result, along with such a region-level error containment, Clover's idempotence-based recovery scheme can always correctly recover from them by re-executing the region. Again, if the 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 3.1. *Given a tail-DMR frontier that makes the execution time of the DMR-enabled part longer than the time of WCDL, all the errors occurring in each region are detected in the same region.*

PROOF. We provide the proof by contradiction. Suppose the argument is false; that is, for an error occurring 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 occur 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 3.1 must be true. □

THEOREM 3.2. *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 3.1. □

Intuitively, Theorem 3.1 means that all the errors occurring in a region should be detected before the region finishes, while Theorem 3.2 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

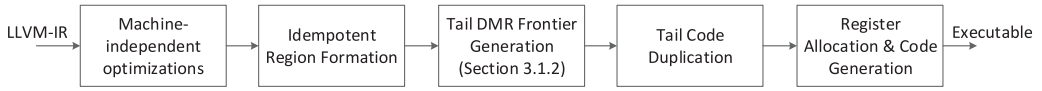

Fig. 3. Clover compiler workflow.

detect an error occurring in the region after it is finished. However, at this moment, the error must have already been corrected based on Theorem 3.2. 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, that is, the beginning of the current region where the error is detected. The takeaway is that Clover can eliminate DUEs.

3.1.2. Clover Compiler Overview. Clover performs detailed compiler analyses to protect an entire idempotent code region against the soft errors. Clover introduces additional compiler back-end passes to generate soft-error-tolerant code. Figure 3 shows the compilation workflow of Clover.

Once the compiler front end translates source code into LLVM intermediate representation (IR), Clover applies traditional compiler optimizations on the IR. Then, the optimized IR goes through idempotent region formation passes, which partition and transform the entire program into idempotent regions so that the regions become re-executable without any side effects. Note that Clover applies De Kruijf's region partition algorithm [de Kruijf et al. 2012] to partition and generate the idempotent regions. At the end of this stage, the LLVM IR is lowered to the machine-specific IR; that is, 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. Finally, the compiler performs register allocation and runs the rest of the back-end passes to emit an executable. This section focuses on elaborating the tail-DMR frontier generation pass.

3.1.3. Safe Approximation of WCDL with Instruction Counting. The tail-DMR frontier pass is designed to recognize those instructions that are vulnerable to DUEs in the tail of an idempotent region. As mentioned in Theorem 3.1, 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 cycles. However, static analysis cannot exactly model the execution time of the instructions.

To get around this problem, 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 the processor's pipeline,[2] which is called $Threshold_{WCDL}$; if the WCDL is five cycles and the commit-width is 2, the tail-DMR forms the DMR-enabled part with only 10 instructions. Thus,

[2]Such approximation is safe as the commit-width is the ideal IPC number for a processor. The number of instructions being committed in each cycle can never exceed the ideal IPC number. Further optimization such as best-case execution time (BCET) analysis can be applied to reduce the number of instructions to be duplicated. For example, SIMD instructions may need more than one cycle to be executed. However, this is out of the scope of this article, and we leave it as future work along with a profile-guided region multiversioning [Zhou et al. 2014] based on adaptive execution techniques [Jung et al. 2005; Lee et al. 2010; Jung et al. 2009].

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 that categorizes the instructions of each region as presented later. Note that the unit of Clover's cost model is in terms of instruction as we approximate the time with the number of instructions discussed earlier.

Synchronization Instructions. Instructions in this category require immediate verification [Reis et al. 2005b]. Originally, store and control flow instructions fell into this category. They require equivalence verification to detect the soft errors; for example, for store instructions, the compiler inserts check instructions to compare the value of the original operands of the store instruction with their duplicated counterpart. If any mismatch is detected, tail-DMR raises an alarm to invoke the recovery process. 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 the following regions. That is, any live-out registers, which 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 these 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 instruction; thus, it does not include the cost of check instructions, which are 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 instructions: one for the original instruction and the other for the duplication instruction. In particular, the 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 DUEs 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 duplicated; that is, 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. These instructions are basically comparisons to check the equivalence of the values in the original instruction and the duplicated one; Section 3.1.4 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 instructions for the 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.

3.1.4. 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 as discussed earlier. If the count reaches a threshold that represents the time of WCDL (i.e., $Threshold_{WCDL}$), 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 definition register needs to be verified with checking instructions during the tail-DMR duplication.
—$\mathbb{TF}$: A set of instructions that belong to tail-DMR frontier.
—$\mathbb{PATHI}$: A set of visited instructions along one path during the reverse depth-first-search traversal.
—C_{path}: Accumulated cost of the 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 all the paths in a reverse depth-first-search (RDFS) order (lines 26–41). Each path is first initialized and keeps track of its own set of visited instructions ($\mathbb{PATHI}$), vulnerable registers ($\mathbb{VR}$), and path cost (C_{path}) during the RDFS traversal (lines 28–30).

For each visited instruction in the path, Clover updates $\mathbb{PATHI}$, $\mathbb{VR}$, and C_{path}, correspondingly (lines 32–34). 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, which will be used in computing the $\mathbb{VR}$ set. Then, to update the $\mathbb{VR}$, Clover leverages the heuristic shown in lines 1 to 11. If the visited instruction is a synchronization instruction, Clover adds all its use registers to the $\mathbb{VR}$ set since the value in those use registers may corrupt the architectural state if it is not verified. For example, if the operands of a store instruction are corrupted, the store may store values to some other memory location, which may break the idempotent property. In the next pass (i.e., the tail code duplication in Figure 3), necessary check instructions are therefore inserted into the original CDFG for verifying these 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 3.1, all the errors occurring before the tail-DMR frontier should be dealt with by the sensor-based soft error detection within the region; that is, 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 the 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.

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:    Return VR
11: end function
12:
13: function calculateCost(C, I)
14:    if I ∈ SYN then
15:        C ← C + C_syn
16:    else
17:        C ← C + C_dup
18:    end if
19:    if DefR ∈ VR then // DefR is the register defined by I
20:        C ← C + C_ck
21:        CK ← CK + I
22:    end if
23:    Return C
24: end function
25:
26: for each region boundary R ∈ R in CDFG do
27:    for each path P in reverse-DFS order from R do
28:        PATHI ← ∅
29:        VR ← ∅
30:        C_path ← 0
31:        for each Instruction I ∈ P do
32:            PATHI ← PATHI + I
33:            VR ← UpdateVulRegSet(VR, I, PATHI)
34:            C_path ← calculateCost(C_path, I)
35:            if C_path ≥ Threshold_WCDL ∨ I ∈ R then
36:                TF ← TF + I
37:                Terminate path P
38:            end if
39:        end for
40:    end for
41: end for
```

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 after the tail-DMR frontier. Line 7 in 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 definition register DefR. Then, the intersection of $\mathbb{KILL}$[DefR] and $\mathbb{PATHI}$ represents whether DefR is killed before the region ends. If the intersection is empty (i.e., the DefR is not killed, which means it is live-out across the end of the region), the DefR is added to $\mathbb{VR}$ for verification.

After the $\mathbb{VR}$ is updated, Clover calculates the path cost (i.e., C_{path}) based on the $\mathbb{VR}$ set and the instruction type of the current visiting instruction. Clover accumulates the

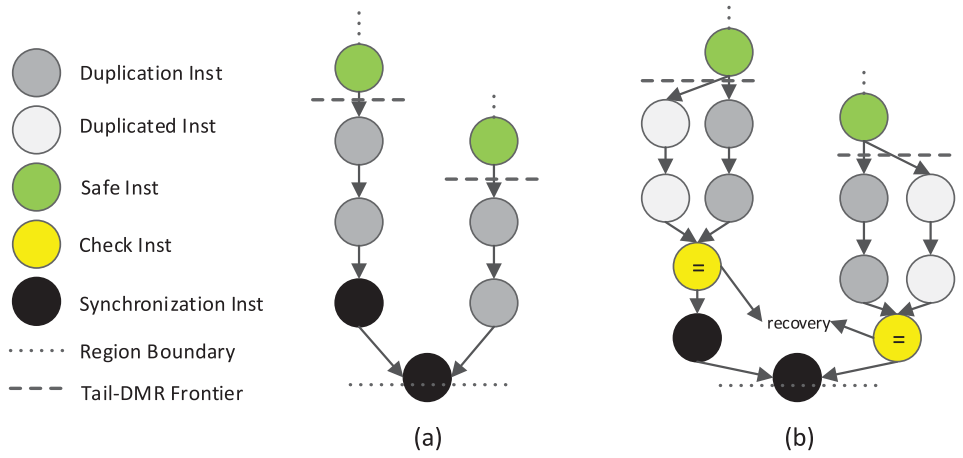

Fig. 4. Tail DMR-frontier and code duplication example: (a) an example control data flow graph with tail-DMR frontiers and (b) code duplication performed by Clover.

cost of visited instructions, which is determined differently depending on whether the instruction is a synchronization instruction or it is a duplication instruction (lines 14–18). In addition, if the register defined by the visiting instruction I belongs to the $\mathbb{VR}$ set, a check instruction cost (C_{ck}) is added to C_{path} (lines 19–22). Accordingly, the instruction I is added into the $\mathbb{CK}$ set to inform the next code duplication pass of where check instruction needs to be placed.

In lines 35 to 38, 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 36). Currently, the default values of the WCDL and the commit-width are 5 and 2, respectively; that is, the $Threshold_{WCDL} = 10$. In addition, Clover also terminates the path cost 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, and thus all the instructions of such a short region are protected by DMR.

Example. Figure 4 demonstrates an example of identifying the tail-DMR frontiers and performing code duplication to the program with tail-DMR frontiers. Figure 4(a) shows an example a control data flow graph. Clover recognizes the tail-DMR frontiers on a path-sensitive basis, which means Clover will iterate all the paths beyond the end of the region. Starting from the region boundary (one of the synchronization instructions), Clover traverses the paths in a reverse depth-first-search order. It accumulates the cost of each path based on Algorithm 1 and categorizes the instructions (Section 3.1.3) until it meets the threshold or other region boundaries. As we can see, the instructions between the tail-DMR frontiers and region boundary are categorized as duplication instructions except for the synchronization instructions. After that, Clover performs code duplication to the duplication instructions and inserts check instructions right before the synchronization instruction as needed (Figure 4(b)). In this way, Clover protects the tail part of the region with DMR.

3.1.5. Discussion and Limitation.

SDC. Even if an energetic particle strike is the major source of soft errors, they can also be induced by other sources, for example, 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).

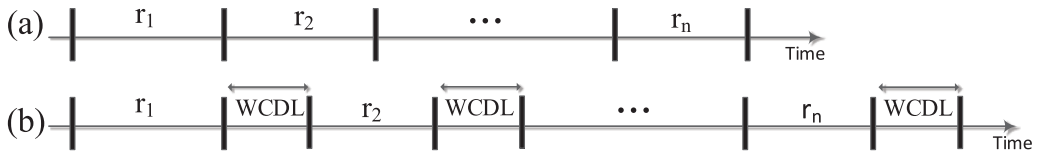

Fig. 5. Example of tail-wait: (a) the original program and (b) the tail-wait approach.

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

Multithreaded Applications. Although we do not evaluate Clover in the context of multithreaded applications, Clover is capable of handling multithreaded programs. Both DMR [Reis et al. 2005b] and idempotent processing [de Kruijf et al. 2012; de Kruijf and Sankaralingam 2013; Feng et al. 2011] have well addressed the problems with multithreaded programs. In addition, there are no conflicts if we extend both DMR and idempotent processing to support multithreaded applications. Therefore, Clover can be easily extended to support multithreaded programs.

Exception Handling. Soft error may lead to exception events such as divide by zero or segmentation fault. We advocate postponing the exception handling service by waiting for WCDL cycles until all potential errors have been detected as with prior work [Upasani et al. 2014a]. In this way, Clover can achieve the region-level error containment even in the presence of such exceptions.

3.2. Tail-Wait

Tail-wait is a straightforward alternative to Clover. In order to avoid DUEs in the tail region, the system can just wait for WCDL cycles at the end of each region to detect any potential errors as shown in Figure 5. In fact, it is possible for tail-wait to outperform Clover in some situations. This is because Clover's way of estimating the WCDL time is very conservative, that is, assuming the ideal processor performance to determine the tail-DMR frontier for the region-level error containment. For example, even if the SIMD instruction may take more than one cycle to execute, Clover calculates the execution time as one over the *commit_width* of the underlying processor. The rest of this section presents the analysis of the situations where Clover (tail-DMR) outperforms tail-wait in theory and vice versa.

First, Clover's performance overhead can be represented as additional cycles due to the instruction duplication, using Equation (1),

$$Overhead_{Clover} = \frac{WCDL \times commit\_width}{2 \times IPC}, \tag{1}$$

where IPC (Instructions Per Cycles) is the actual performance number delivered by the processor. The equation assumes that half of the $WCDL \times commit\_width$ instructions are those that are additionally inserted for Clover to duplicate the original instructions. Here, Clover's IPC is assumed to be the same as tail-wait's for ease of presentation. Note that this rather underestimates the performance of Clover as the DMR is known to increase the IPC significantly [Reis et al. 2005b]; that is, Clover's IPC is likely to be higher than tail-wait's.

As the tail-wait's performance overhead is just WCDL for each region, we can derive Equation (2) from Equation (1) to analyze the situation where Clover outperforms

tail-wait:

$$\frac{WCDL \times commit\_width}{2 \times IPC} \leq WCDL. \tag{2}$$

Therefore, we have

$$\frac{commit\_width}{2} \leq IPC. \tag{3}$$

From Equation (3), we can draw the conclusion that if the IPC number of Clover is greater than or equal to half of the commit-width of the underlying processor, Clover can always perform better than tail-wait. For example, when the commit-width is 2, Clover can outperform tail-wait provided Clover's IPC is greater than 1. In contrast, if the processor suffers from many long latency instructions within a short amount of time, such that the IPC drops down below 1, then tail-wait can be a better choice over Clover.

Note that if the region size is smaller than $WCDL \times commit\_width$, Clover can be superior over tail-wait, even though the IPC is less than half of the commit-width, violating the inequality in Equation (3). As an extreme example, when a region consists of only a few instructions, Clover duplicates them all since its execution time is too short to match the WCDL. On the contrary, after such a short region ends, tail-wait fully waits for WCDL cycles that would be longer than the execution time of the additional instructions inserted by Clover.

In Section 4.4, we empirically evaluate the tail-DMR compared to the tail-wait to support our analyses on their performance overhead varying WCDL and underlying processor configurations.

4. EVALUATION

We implement the compiler passes of Clover on top of LLVM Compiler Infrastructure [Lattner and Adve 2004]. The idempotent region formation algorithm is also integrated in LLVM. We perform the experiments with 17 applications from Mediabench [Lee et al. 1997] and MIbench [Guthaus et al. 2001] benchmarks in different categories. All the applications were compiled with standard -O3 optimization. We conduct our simulations on Gem5 [Binkert et al. 2011] with system call emulation mode for a modern two-issue out-of-order 0.5GHz processor whose L1 (two-way/two-cycle) and L2 (eight-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 the region is, the better the 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, that is, combination of idempotent processing and full-DMR [de Kruijf et al. 2012]. Finally, we provide sensitivity analysis results to understand the tradeoff between the sensor area overhead and the resulting performance of Clover.

4.1. Region Characteristics

Figure 6(a) shows a cumulative distribution of dynamically executed idempotent regions of all the applications listed in Table I. The x-axis (in log scale) represents the number of instructions in regions. We highlight *unepic* and *adpcmdecode*. As shown in Figure 6(b), the majority of regions in *unepic* are composed of less than 10 instructions,

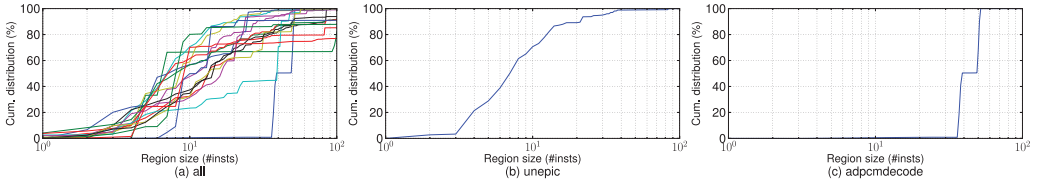

Fig. 6. The distribution of the original idempotent code regions (dynamic).

Table I. Dynamic Region Characteristic with Five-Cycle WCDL

Application	#Total Insts (10^3)	#Total Regs (10^3)	Aver. Leng.	#Vulner. Insts (10^3)	Vulner. Insts Ratio
adpcmdecode	6,557	149	44	1,486	22.66%
adpcmencode	8,346	223	37	1,231	14.75%
epic	59,759	1,186	50	8,370	14.00%
unepic	9,898	941	10	6,847	69.17%
jpegdecode	4,382	280	15	2,252	51.39%
jpegencode	17,335	1,205	14	9,335	53.85%
mesatexgen	204,820	8,497	24	68,748	33.56%
pegwitencrypt	35,616	2,600	13	18,586	52.18%
g721decode	512,016	14,239	35	94,027	18.36%
g721encode	268,789	7,766	34	51,155	19.03%
gsmdecode	68,406	2,270	30	19,625	28.68%
gsmencode	110,750	3,350	33	27,756	25.06%
mpeg2decode	165,491	5,792	28	49,946	30.18%
mpeg2encode	1,320,760	17,867	73	143,323	10.85%
sha	120,338	1,121	107	9,530	7.91%
susanedges	78,967	2,190	36	15,526	19.66%
susancorners	27,265	455	59	2,823	10.35%
geomean	58,368	1,822	31	13,736	23.53%

and they occupy a 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 6(c). That is, most of the regions are long enough to hide the performance penalty caused by the tail-DMR, and thus it will cause negligible performance overhead for *adpcmdecode*.

Table I further details the dynamic region characteristics of the benchmark applications. Columns 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., second column/third column). The geometric mean of the average region length is 31. Such fine-grained recovery is beneficial because it guarantees the continuity of program execution in the event of soft errors. The system just needs to roll back to the beginning of the faulty region and re-execute the region, whose average length is 31, making the recovery overhead negligible to the users. In contrast, restarting the program will disrupt the program continuity and affect the users. In the extreme case, the sensor could raise an alarm every 1.3 minutes [Upasani et al. 2014a]; the program can never finish if it takes more than 1.3 minutes. However, Clover just needs to pay the negligible recovery overhead of re-executing 31 instructions per fault.

Columns 5 and 6 represent the total number of vulnerable instructions and the ratio (i.e., fifth column/second column), respectively. Note that simply relying on the naive combination of the idempotent processing and sensor-based detection scheme leaves all the regions potentially vulnerable to DUEs. On average, a total of 23%

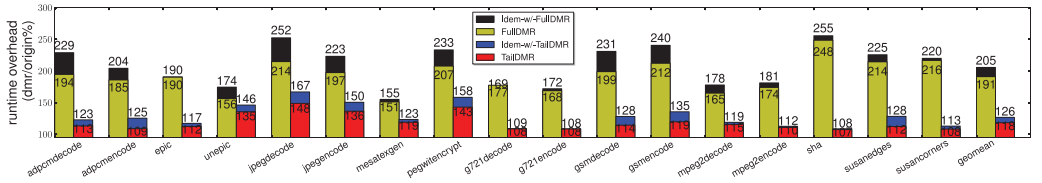

Fig. 7. Performance overhead of Clover vs. Full-DMR.

of dynamic instructions are vulnerable to the 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 is doubled after the DMR is performed; that is, 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 much lower performance overhead compared to full DMR.

4.2. Performance Overhead and Code Size

Figure 7 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 [de Kruijf et al. 2012], where full-DMR is combined with idempotent processing, while the second bar corresponds to the runtime of Clover. In the figure, each bar is broken into two parts; the bottom and the top represent the overhead 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*) represent the overhead due to idempotence-based error recovery in the presence of full-DMR and Clover's tail-DMR, respectively. As shown in Figure 7, the idempotence-based recovery is not that significant, that is, on average, 14% (*Idem-w/-FullDMR*) and 8% (*Idem-w/-TailDMR*). Thus, most of the overhead is caused by the error detection schemes, that is, on average, 91% (*FullDMR*) and 18%(*TailDMR*). Overall, the full-DMR 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. Figure 7 also confirms that the length of regions is critical to Clover's performance overhead (i.e., second bar). The general trend is that the higher ratio of vulnerable instructions shown in Table I translates to higher performance overhead.

Table II summarizes the code size increase of the full-DMR with idempotent processing versus Clover. The number of additional static instructions inserted into the original program is represented in column 2 (the full-DMR approach) and column 3 (Clover). Clover achieves on average a 46% static instruction reduction when compared to the full-DMR approach as shown in column 4. 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%. Clover achieves on average a 53% binary size reduction when compared to the full-DMR approach as shown in column 7.

4.3. Sensitivity Analysis

We investigate the factors that affect Clover in this section. As the overhead of Clover mostly comes from the tail-DMR, the fraction of the tail of each region protected by DMR will impact the resulting performance of Clover. The WCDL and pipeline width are two

Table II. Code Size Comparison: Full-DMR Versus Tail-DMR
Gsmencode and *Susancorners* share the same binaries with *Gsmdecode* and *Susanedges*, respectively. Therefore, they have the exact same data.

Application	#Full DMR Insts	#Tail DMR Insts	Insts Reduction Ratio	Full DMR Binary Size Increase Ratio	Tail DMR Binary Size Increase Ratio	Binary Size Reduction Ratio
adpcmdecode	408	146	64.21%	47.24%	1.34%	97.17%
adpcmencode	411	146	64.47%	47.24%	1.34%	97.17%
epic	9,314	5,443	41.56%	106.54%	58.96%	44.66%
unepic	7,179	5,210	27.42%	108.46%	69.70%	35.74%
jpegdecode	52,209	34,728	33.48%	124.79%	82.13%	34.19%
jpegencode	50,282	33,326	33.72%	130.77%	79.86%	38.93%
mesatexgen	183,345	104,805	42.83%	128.74%	68.18%	47.04%
pegwitencrypt	12,297	7,144	41.90%	96.63%	50.01%	48.24%
g721decode	2,524	1,393	44.80%	66.87%	34.81%	47.95%
g721encode	2,659	1,481	44.30%	68.02%	35.88%	47.26%
gsmdecode	10,687	5,490	48.62%	68.65%	31.20%	54.55%
gsmencode	10,687	5,490	48.62%	68.65%	31.20%	54.55%
mpeg2decode	15,884	9,580	39.68%	107.01%	58.29%	45.53%
mpeg2encode	23,680	11,528	51.31%	112.55%	49.24%	56.25%
sha	857	407	52.50%	61.02%	26.05%	57.30%
susanedges	7,187	2,522	64.90%	102.50%	33.03%	67.77%
susancorners	7,187	2,522	64.90%	102.50%	33.03%	67.77%
geomean	7,552	3,858	46.23%	86.60%	30.42%	53.02%

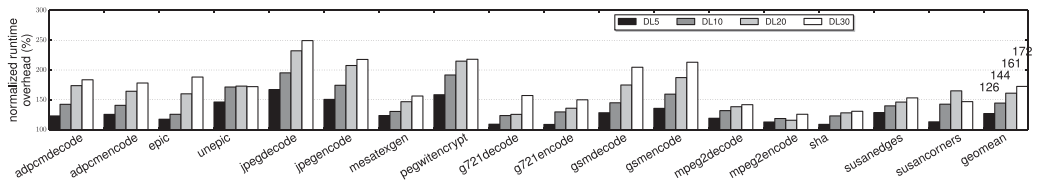

Fig. 8. Sensitivity to the WCDL where pipeline width is 2.

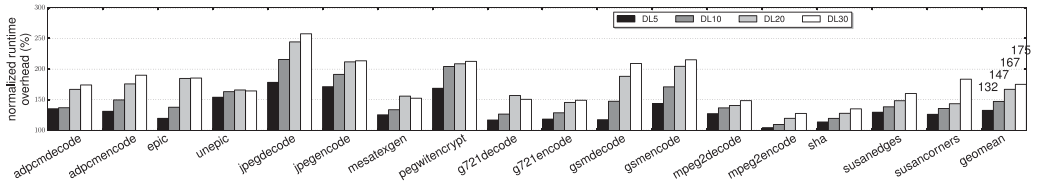

Fig. 9. Sensitivity to the WCDL where pipeline width is 3.

critical factors that determine the fraction of tail region with tail-DMR frontiers. As shown in Figures 8 through 10, we perform sensitivity analysis by varying the pipeline width and WCDL to see the impact of those two factors on the performance. Figures 8, 9, and 10 show the normalized runtime overhead with respect to the baseline without instrumentation with different pipeline width from 2, 3, to 4, respectively. For each configuration, we vary the WCDL from 5 to 10 to 20 to 30.

For all cases, the performance overhead of Clover generally increases (from 26%–34% to 69%–75%) as the worst-case detection latency increases (from 5 to 30). The results were expected because the number of instructions that need to be duplicated for tail-DMR becomes larger with longer WCDL. Note that the sensor-based detection

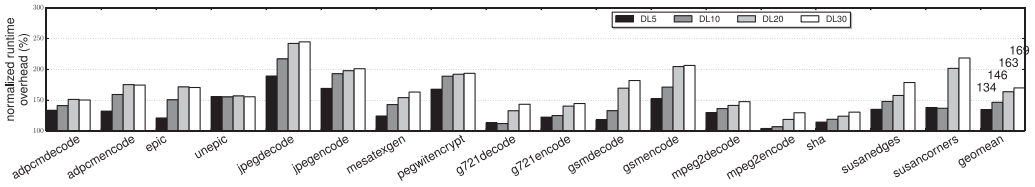

Fig. 10. Sensitivity to the WCDL where pipeline width is 4.

scheme can adjust its WCDL by varying the amount of sensors being deployed and their placement on the processor core. However, as illustrated in Section 2, a larger amount of sensors will result in higher area overhead. We observe that a few applications such as susancorners and mpegencode deviate from this general trend. The abnormal L1 instruction cache access (including hits and misses) increases in these applications can account for these aberrant cases. For example, we observe around a 20% increase of L1 instruction cache access cycles in susancorners when the detection latency is 20, which in turn increases the performance overhead, making it an exception. We suspect that DMR affects the code layout, triggering such abnormality.

When the detection latency becomes large enough (e.g., 20), the performance overhead does not increase significantly as the detection latency keeps growing (from 20 to 30). The excessive number of small regions can account for such performance overhead saturation phenomena. As shown in Figure 6(a), most of the applications have more than 50% of total dynamic regions whose region size is fewer than 10 instructions. This implies that when the detection latency becomes large enough, most of the regions need to be fully protected by DMR because the sensor cannot detect the error within the region in the worst case. Therefore, the performance overhead becomes saturated when most of the region is fully protected by the DMR. However, Clover is still effective to reduce the performance overhead as large regions still dominate in the total program execution time. For example, we assume a program composed of 100 regions where 99 regions' size is 10 and only one region's size is 10,000. The program spends more than 90% of the time in the large region. Therefore, Clover can effectively reduce the performance overhead with tail-DMR compared to full DMR in those large regions.

On the other hand, we also observe that the performance overhead of Clover does not degrade significantly when the pipeline width increases (from 2 to 4) even though the number of vulnerable instructions increases as the pipeline width grows. Sometimes, the performance overhead even decreases as the pipeline width increases. Such performance variants were generally expected. This is because as the pipeline width increases, more instruction-level parallelism can be exposed to the processor, thus increasing the IPC as the duplicated instructions generally do not depend on the original instructions. Prior work (e.g., Swift [Reis et al. 2005b]) also observes the same phenomenon.

4.4. Comparison with Tail-Wait

Tail-wait is a straightforward way to guarantee all the errors to be detected within their regions. In this section, we compare the overhead of Clover with that of tail-wait. We implemented the tail-wait by inserting nops to the end of each region. The number of nops to be inserted is determined as the product of WCDL and pipeline width. We normalize the performance overhead of tail-wait to the same baseline similar to the previous section.

Figures 11 through 13 show that Clover is consistently better than tail-wait across different configurations. We confirm that the IPC numbers in Clover are always greater than half of the commit width (not shown in the graph), which supports our theoretical

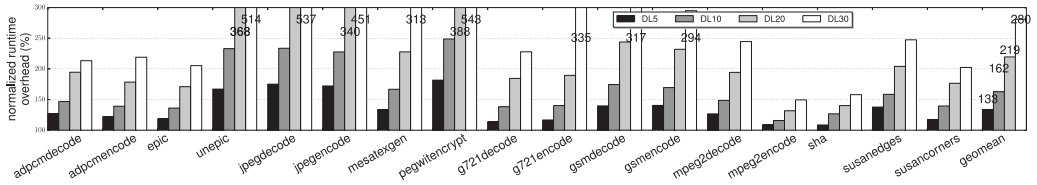

Fig. 11. Tail-wait across different WCDLs where pipeline width is 2.

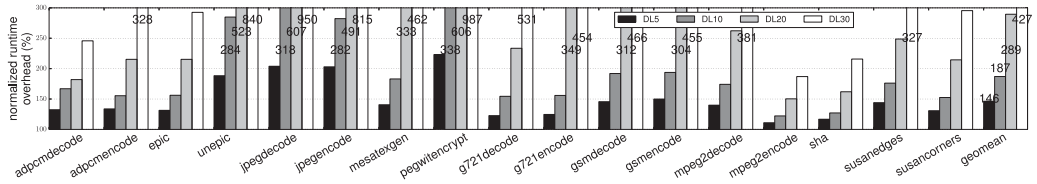

Fig. 12. Tail-wait across different WCDLs where pipeline width is 3.

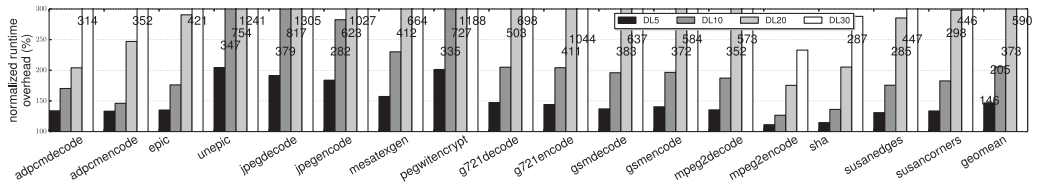

Fig. 13. Tail-wait across different WCDLs where pipeline width is 4.

performance analyses in Section 3.2. Interestingly, the performance overhead of tail-wait degrades much more significantly with longer WCDL and/or wider pipeline width, which shows the benefits of tail-DMR compared to the naive tail-wait strategy. The large number of small regions can account for such trend as the performance overhead of tail-wait is proportional to WCDL and pipeline width, while that of Clover will be saturated as discussed in Section 4.3. In all, Clover achieves 1.06 to 3.49× speedup over the tail-wait approach on average.

5. OTHER RELATED WORK

This section describes the prior works related to soft error detection, correction, and recovery. We also explain how our proposal, Clover, advances the state of the art and differs from previous approaches in this domain.

5.1. Soft Error Detection and Correction

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 [Reis et al. 2005b]. It checks the value of registers with their duplication counterpart at certain synchronization points, that is, memory and control flow instructions. Rotenberg [1999] takes advantage of simultaneous multithreading (SMT) to run a trailing thread that verifies the leading thread. 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 [Khudia and Mahlke 2014; Cong and Gururaj 2011; Hari et al. 2012b; Sastry Hari et al. 2013; Li et al. 2008; Hari et al.

2012a; Sahoo et al. 2008; Sastry Hari et al. 2009; Shafique et al. 2013; Khudia and Mahlke 2014; Li et al. 2013]. They all exploit some heuristics or statistics to identify the instructions that are critical to the program output and selectively protect these instructions with DMR. Hari et al. [2012b, 2013] also investigated approaches where the outcome of the soft errors for specific applications could be predicted. Hari et al. [2012a] also proposed novel program-level detectors to detect silent data corruptions and demonstrate that it reduces the overhead imposed by redundancy-based techniques. Li et al. [2008] proposed online methods to estimate the architecture vulnerability factor to the soft errors for various microarchitecture factors. On the other hand, SWAT uses program invariants to detect the hardware errors [Sahoo et al. 2008]. mSWAT, an extension of the SWAT framework, applies symptom-based detection and diagnosis for faults in multicore architectures [Sastry Hari et al. 2009]. Epipe [Li et al. 2013] identifies the vulnerable instructions that lead to SDC and selectively protects a subset of those vulnerable instructions considering the worst-case execution time (WCET) constraints in the fault-free execution. Similarly, Rehman et al. [2016, 2014a, 2014b] explore the varying vulnerability property of different applications along with different compilation versions to significantly reduce the application failures.

Khudia et al. leverage profile information to reduce the cost of duplication and checking [Khudia et al. 2012; Khudia and Mahlke 2014]. They propose profile-based instruction-level DMR to focus on those instructions that are likely to affect the important program output [Khudia et al. 2012]. Recently, Khudia and Mahlke [2014] exploited the value-locality of instruction results during the profile run to achieve selective DMR. Other techniques estimate the soft error vulnerability of instructions executed in the underlying processor [Rehman et al. 2011] and use statistical models to measure the probabilities of the error masking and the propagation in the processor for enabling selective protection [Shafique et al. 2013]. However, all those approaches achieve low performance overhead at the expense of reduced fault coverage. In contrast, Clover selectively protects some of the instructions (i.e., only those instructions vulnerable to DUEs) without sacrificing the fault coverage. Chen and Yang [2013] 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. 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 [Austin 1999] relies on a simple in-order core to verify the program execution, while Argus [Meixner et al. 2007] leverages invariant checking to ensure correctness. However, these approaches often introduce an excessive hardware complexity increase, which is not acceptable in embedded systems. ReStore [Wang and Patel 2006] advocates utilizing symptoms of the soft errors to detect them without significant overhead. Shoestring [Feng et al. 2010] enhances ReStore by selectively duplicating some vulnerable instructions with simple heuristics. However, both ReStore and Shoestring incur long detection latency, which may result in DUEs. Similar to the recent work of Khudia and Mahlke [2014] and Racunas et al. [2007] proposes to make use of the value-locality to detect the soft errors. However, the issues of false positives/negatives [Jung et al. 2014; Lee et al. 2014; Jung 2013] in the locality-based approaches prevent us from adopting their methods.

Several recent works have proposed reducing the ECC performance and power overhead and increasing the correction capabilities, such as Bamboo ECC [Kim et al. 2015], FreeFault [Kim and Erez 2015], and Virtualized ECC [Yoon and Erez 2010]. On the other hand, proposals such as Containment domains argue for a new programming construct for applications to express resilience requirements and tune error detection, state preservation, and recovery schemes [Chung et al. 2013]. More recently, Upasani

et al. [2012, 2013, 2014a, 2014b, 2016] proposed detecting the soft errors with a configurable amount of acoustic wave sensors. Their sensor-based detection scheme achieves low soft error detection latency with reasonable hardware overhead. According to the recent work of Upasani et al. [2014b], 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 article takes advantage of such sensor-based detection scheme and selectively duplicates only the instructions vulnerable to DUEs in the tail of an idempotent region for guaranteed soft error recovery.

5.2. Soft Error Recovery

Checkpointing the whole program state (memory and registers) guarantees recovery from the soft errors by allowing programs to roll back to the previous safe checkpoint [Wang and Patel 2006; Feng et al. 2010; Upasani et al. 2014a]. 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. [2014a] keeps two copies of the register file and the register allocation table (RAT) to achieve low performance overhead. Jeyapaul et al. [2014] 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; that is, combinational logic is still vulnerable to the soft errors and thus they may generate SDC. Flushing the pipeline to recover from a soft error [Racunas et al. 2007; Upasani et al. 2014b] is another alternative. This approach is expected to be very efficient in terms of runtime overhead. However, this approach is often based on the assumption that detection can be done before the faulty instruction is committed; that is, the error detection latency should be zero. Such low detection latency inevitably requires high performance/hardware overhead as stated in Section 5.1. In particular, Clover avoids such high overhead by integrating idempotent processing that recovers from the soft errors by simply re-executing the region in which they occur. That is, even if the 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 the 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 detectors.

The preliminary work of this article has been presented in Liu et al. [2015].

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 no DUEs. It can also achieve almost zero SDC as long as the soft errors come from the energetic particle strike, which is the major source of soft errors. 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 the soft errors by sensing the wave made by the particle strike. On the software side, Clover leverages a novel selective instruction duplication technique called tail-DMR that offers a region-level error containment to cope with DUEs caused by the sensing latency of the 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 raised by either the

sensor or the tail-DMR, the exception handler of which simply redirects program control to the beginning of the region. In this way, Clover can achieve soft error resilience with low runtime and negligible area overheads. The experimental 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 (i.e., idempotent processing + full DMR). Finally, this article evaluates the proposed tail-DMR technique compared to a new alternative to it called tail-wait. Evaluating the techniques with the different processor configurations and the various error detection latencies confirms that the tail-DMR is a superior technique, achieving a 1.06 to 3.49× speedup over the tail-wait.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous referees for their valuable comments.

REFERENCES

ARM. 2015. Cortex-A57 Technique Reference Manual. Retrieved from http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0488g/index.html.

Todd M. Austin. 1999. DIVA: A reliable substrate for deep submicron microarchitecture design. In *Proceedings of the 32nd Annual International Symposium on Microarchitecture (MICRO-32'99)*. IEEE, 196–207.

Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 simulator. *SIGARCH Computer Architecture News* 39, 2 (2011), 1–7.

Javier Carretero, Pedro Chaparro, Xavier Vera, Jaume Abella, and Antonio González. 2009. End-to-end register data-flow continuous self-test. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA'09)*. ACM, New York, NY, USA, 105–115.

Hao Chen and Chengmo Yang. 2013. Boosting efficiency of fault detection and recovery through application-specific comparison and checkpointing. In *Proceedings of the 14th ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES'13)*. ACM, 13–20.

Jinsuk Chung, Ikhwan Lee, Michael Sullivan, Jee Ho Ryoo, Dong Wan Kim, Doe Hyun Yoon, Larry Kaplan, and Mattan Erez. 2013. Containment domains: A scalable, efficient and flexible resilience scheme for exascale systems. *Scientific Programming* 21, 3–4 (2013), 197–212.

Jason Cong and Karthik Gururaj. 2011. Assuring application-level correctness against soft errors. In *Proceedings of the International Conference on Computer-Aided Design*. IEEE Press, 150–157.

Cristian Constantinescu. 2003. Trends and challenges in VLSI circuit reliability. *Proceedings of the 36nd Annual International Symposium on Microarchitecture, 2003 (MICRO-36'03)* 23, 4 (July 2003).

Marc de Kruijf and Karthikeyan Sankaralingam. 2013. Idempotent code generation: Implementation, analysis, and evaluation. In *CGO*. IEEE Computer Society, 1–12.

Marc A. de Kruijf, Karthikeyan Sankaralingam, and Somesh Jha. 2012. Static analysis and compiler design for idempotent processing. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'12)*. 475–486.

Shuguang Feng, Shantanu Gupta, Amin Ansari, and Scott Mahlke. 2010. Shoestring: Probabilistic soft error reliability on the cheap. *ACM SIGARCH Computer Architecture News* 38 (2010), 385–396.

Shuguang Feng, Shantanu Gupta, Amin Ansari, Scott A. Mahlke, and David I. August. 2011. Encore: Low-cost, fine-grained transient fault recovery. In *MICRO'11*. 398–409.

Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of thee 2001 IEEE International Workshop on Workload Characterization (WWC-4'01)*. IEEE, 3–14.

Imran S. Haque and Vijay S. Pande. 2010. 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)*. 691–696.

Siva Kumar Sastry Hari, Sarita V. Adve, and Helia Naeimi. 2012a. Low-cost program-level detectors for reducing silent data corruptions. In *Proceedings of the 2012 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'12)*. IEEE, 1–12.

Siva Kumar Sastry Hari, Sarita V. Adve, Helia Naeimi, and Pradeep Ramachandran. 2012b. Relyzer: Exploiting application-level fault equivalence to analyze application resiliency to transient faults. *ACM SIGPLAN Notices* 47 (2012), 123–134.

Jörg Henkel, Lars Bauer, Nikil Dutt, Puneet Gupta, Sani Nassif, Muhammad Shafique, Mehdi Tahoori, and Norbert Wehn. 2013. Reliable on-chip systems in the nano-era: Lessons learnt and future trends. In *Proceedings of the 50th Annual Design Automation Conference (DAC'13)*. 99:1–99:10.

Reiley Jeyapaul, Abhishek Risheekesan, Aviral Shrivastava, and Kyoungwoo Lee. 2014. UnSync-CMP: Multicore CMP architecture for energy efficient soft error reliability. *Transactions on Parallel and Distributed Systems* 25, 1 (January 2014), 254–263.

Changhee Jung. 2013. *Effective Techniques for Understanding and Improving Data Structure Usage*. Ph.D. Dissertation, Georgia Institute of Technology, Atlanta, GA.

Changhee Jung, Sangho Lee, Easwaran Raman, and Santosh Pande. 2014. Automated memory leak detection for production use. In *Proceedings of the 36th International Conference on Software Engineering*.

Changhee Jung, Daeseob Lim, Jaejin Lee, and SangYong Han. 2005. Adaptive execution techniques for SMT multiprocessor architectures. In *Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 236–246.

Chang Hee Jung, Dae Seob Lim, Jae Jin Lee, and Sang Yong Han. 2009. Adaptive execution method for multithreaded processor-based parallel system. US Patent No. 7,526,637.

H. Kaul, M. Anders, S. Hsu, A. Agarwal, R. Krishnamurthy, and S. Borkar. 2012. Near-threshold voltage (NTV) design opportunities and challenges. In *Proceedings of the 2012 49th ACM/EDAC/IEEE Design Automation Conference (DAC'12)*. 1149–1154.

D. S. Khudia and S. Mahlke. 2014. Harnessing soft computations for low-budget fault tolerance. In *Proceedings of the 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'14)*. 319–330.

Daya Shanker Khudia and Scott Mahlke. 2013. 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)*.

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

Dong Wan Kim and Mattan Erez. 2015. Balancing reliability, cost, and performance tradeoffs with FreeFault. In *Proceedings of the 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA'15)*. IEEE, 439–450.

Jungrae Kim, Michael Sullivan, and Mattan Erez. 2015. Bamboo ECC: Strong, safe, and flexible codes for reliable computer memory. In *Proceedings of the 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA'15)*. IEEE, 101–112.

Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization, 2004 (CGO'04)*. IEEE, 75–86.

Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. 1997. MediaBench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE Computer Society, 330–335.

Jaejin Lee, Jung-Ho Park, Honggyu Kim, Changhee Jung, Daeseob Lim, and SangYong Han. 2010. Adaptive execution techniques of parallel programs for multiprocessors. *Journal of Parallel and Distributed Computing* 70, 5 (May 2010), 467–480.

Sangho Lee, Changhee Jung, and Santosh Pande. 2014. Detecting memory leaks through introspective dynamic behavior modelling using machine learning. In *Proceedings of the 36th International Conference on Software Engineering*.

Jianli Li, Jingling Xue, Xinwei Xie, Qing Wan, Qingping Tan, and Lanfang Tan. 2013. Epipe: A low-cost fault-tolerance technique considering WCET constraints. *Journal of System Architecture* 59, 10 (November 2013), 1383–1393. DOI:http://dx.doi.org/10.1016/j.sysarc.2013.06.003

Xiaodong Li, Sarita V. Adve, Pradip Bose, Jude Rivers, and others. 2008. Online estimation of architectural vulnerability factor for soft errors. In *Proceedings of the 35th International Symposium on Computer Architecture, 2008 (ISCA'08)*. IEEE, 341–352.

Xin Li, Michael C. Huang, Kai Shen, and Lingkun Chu. 2010. 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)*.

Qingrui Liu and Changhee Jung. 2016. Lightweight hardware support for transparent consistency-aware checkpointing in intermittent energy-harvesting systems. In *Proceedings of the IEEE Non-Volatile Memory Systems and Applications Symposium (NVMSA'16)*.

Qingrui Liu, Changhee Jung, Dongyoon Lee, and Devesh Tiwari. 2015. Clover: Compiler directed lightweight soft error resilience. In *Proceedings of the 16th ACM SIGPLAN/SIGBED Conference on Languages,*

Compilers and Tools for Embedded Systems 2015 CD-ROM (LCTES'15). ACM, New York, NY, Article 2, 10 pages. DOI: http://dx.doi.org/10.1145/2670529.2754959

Qingrui Liu, Changhee Jung, Dongyoon Lee, and Devesh Tiwari. 2016a. Compiler-directed lightweight checkpointing for fine-grained guaranteed soft error recovery. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'16)*.

Qingrui Liu, Changhee Jung, Dongyoon Lee, and Devesh Tiwari. 2016b. Low-cost soft error resilience with unified data verification and fine-grained recovery. In *Proceedings of the 49th International Symposium on Microarchitecture (MICRO'16)*.

Yixin Luo, Sriram Govindan, Bikash Sharma, Mark Santaniello, Justin Meza, Aman Kansal, Jie Liu, Badriddine Khessib, Kushagra Vaid, and Onur Mutlu. 2014. Characterizing application memory error vulnerability to optimize datacenter cost via heterogeneous-reliability memory. In *Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'14)*. IEEE, 467–478.

Robert E. Lyons and Wouter Vanderkulk. 1962. The use of triple-modular redundancy to improve computer reliability. *IBM Journal of Research and Development* 6, 2 (1962), 200–209.

Albert Meixner, Michael E. Bauer, and Daniel J. Sorin. 2007. Argus: Low-cost, comprehensive error detection in simple cores. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture, 2007 (MICRO'07)*. IEEE, 210–222.

Shubhendu S. Mukherjee, Joel Emer, and Steven K. Reinhardt. 2005. The soft error problem: An architectural perspective. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA'05)*. 243–247.

Paul Racunas, Kypros Constantinides, Srilatha Manne, and Shubhendu S. Mukherjee. 2007. Perturbation-based fault screening. In *Proceedings of the IEEE 13th International Symposium on High Performance Computer Architecture, 2007 (HPCA'07)*. IEEE, 169–180.

S. Rehman, Kuan-Hsun Chen, F. Kriebel, A. Toma, M. Shafique, Jian-Jia Chen, and J. Henkel. 2016. Cross-layer software dependability on unreliable hardware. *IEEE Transactions on Computers* 65, 1 (January 2016), 80–94. DOI: http://dx.doi.org/10.1109/TC.2015.2417554

S. Rehman, F. Kriebel, M. Shafique, and J. Henkel. 2014a. Reliability-driven software transformations for unreliable hardware. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* 33, 11 (November 2014), 1597–1610. DOI: http://dx.doi.org/10.1109/TCAD.2014.2341894

Semeen Rehman, Florian Kriebel, Duo Sun, Muhammad Shafique, and Jörg Henkel. 2014b. dTune: Leveraging reliable code generation for adaptive dependability tuning under process variation and aging-induced effects. In *Proceedings of the 51st Annual Design Automation Conference (DAC'14)*. ACM, New York, NY, Article 84, 6 pages. DOI: http://dx.doi.org/10.1145/2593069.2593127

Semeen Rehman, Muhammad Shafique, Florian Kriebel, and Jrg Henkel. 2011. Reliable software for unreliable hardware: Embedded code generation aiming at reliability. In *CODES+ISSS*, Robert P. Dick and Jan Madsen (Eds.). ACM, 237–246.

George A. Reis, Jonathan Chang, and David I. August. 2007. Automatic instruction-level software-only recovery. *IEEE Micro* 27, 1 (2007), 36–47.

George A. Reis, Jonathan Chang, Neil Vachharajani, Shubhendu S. Mukherjee, R. Rangan, and D. I. August. 2005a. Design and evaluation of hybrid fault-detection systems. In *Proceedings of the 32nd International Symposium on Computer Architecture, 2005 (ISCA'05)*. IEEE, 148–159.

George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I. August. 2005b. SWIFT: Software implemented fault tolerance. In *Proceedings of the International Symposium on Code Generation and Optimization*. IEEE Computer Society, 243–254.

Eric Rotenberg. 1999. AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. In *Proceedings of the 29th Annual International Symposium on Fault-Tolerant Computing, 1999. Digest of Papers.* IEEE, 84–91.

Giacinto Paolo Saggese, Nicholas J. Wang, Zbigniew Kalbarczyk, Sanjay J. Patel, and Ravishankar K. Iyer. 2005. An experimental study of soft errors in microprocessors. *IEEE Micro* 25, 6 (2005), 30–39.

Swamp Kumar Sahoo, Man-Lap Li, Pradeep Ramachandran, Sarita V. Adve, Vikram S. Adve, and Yuanyuan Zhou. 2008. Using likely program invariants to detect hardware errors. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks With FTCS and DCC, 2008 (DSN'08)*. IEEE, 70–79.

Siva Kumar Sastry Hari, Sarita V. Adve, Helia Naeimi, and Prasadh Ramachandran. 2013. Relyzer: Application resiliency analyzer for transient faults. *IEEE Micro* 33, 3 (2013), 58–66.

Siva Kumar Sastry Hari, Man-Lap Li, Pradeep Ramachandran, Byn Choi, and Sarita V. Adve. 2009. mSWAT: Low-cost hardware fault detection and diagnosis for multicore systems. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 122–132.

Muhammad Shafique, Siddharth Garg, Jörg Henkel, and Diana Marculescu. 2014. The EDA challenges in the dark silicon era: Temperature, reliability, and variability perspectives. In *Proceedings of the 51st Annual Design Automation Conference on Design Automation Conference (DAC'14)*. 185:1–185:6.

Muhammad Shafique, Semeen Rehman, Pau Vilimelis Aceituno, and Jörg Henkel. 2013. Exploiting program-level masking and error propagation for constrained reliability optimization. In *Proceedings of the 50th Annual Design Automation Conference (DAC'13)*. ACM, New York, NY, Article 17, 9 pages. DOI: http://dx.doi.org/10.1145/2463209.2488755

Michael B. Taylor. 2012. 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)*. 1131–1136.

Gaurang Upasani, Xavier Vera, and Antonio Gonzalez. 2012. Setting an error detection infrastructure with low cost acoustic wave detectors. In *ISCA*. 333–343.

Gaurang Upasani, Xavier Vera, and Antonio Gonzalez. 2013. Reducing DUE-FIT of caches by exploiting acoustic wave detectors for error recovery. In *IOLTS*. 85–91.

Gaurang Upasani, Xavier Vera, and Antonio Gonzalez. 2014a. Avoiding core's DUE & SDC via acoustic wave detectors and tailored error containment and recovery. In *ISCA*. 37–48.

Gaurang Upasani, Xavier Vera, and Antonio Gonzalez. 2014b. Framework for economical error recovery in embedded cores. In *Proceedings of the 2014 IEEE 20th International On-Line Testing Symposium (IOLTS'14)*. IEEE, 146–153.

Gaurang Upasani, Xavier Vera, and Antonio Gonzalez. 2016. A case for acoustic wave detectors for soft-errors. *IEEE Transactions on Computing* 65, 1 (2016), 5–18.

Liang Wang and Kevin Skadron. 2013. Implications of the power wall: Dim cores and reconfigurable logic. *IEEE Micro* 33, 5 (2013), 40–48.

Nicholas J. Wang and Sanjay J. Patel. 2006. ReStore: Symptom-based soft error detection in microprocessors. *IEEE Transactions on Dependable and Secure Computing* 3, 3 (2006), 188–201.

Doe Hyun Yoon and Mattan Erez. 2010. Virtualized and flexible ECC for main memory. *ACM SIGARCH Computer Architecture News* 38 (2010), 397–408.

Mingzhou Zhou, Xipeng Shen, Yaoqing Gao, and Graham Yiu. 2014. Space-efficient multi-versioning for input-adaptive feedback-driven program optimizations. In *Proceedings of the 29th International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'14)*. ACM, 763–776. DOI: http://dx.doi.org/10.1145/2660193.2660229

Received September 2015; revised January 2016; accepted April 2016