Leverage Redundancy in Hardware Transactional Memory to Improve Cache Reliability

Zhichao Yan
University of Texas-Arlington
Arlington, Texas
zhichao.yan@mavs.uta.edu

Hong Jiang
University of Texas-Arlington
Arlington, Texas
hong.jiang@uta.edu

Witawas Srisa-an
University of Nebraska-Lincoln
Lincoln, Nebraska
witty@cse.unl.edu

Sharad Seth
University of Nebraska-Lincoln
Lincoln, Nebraska
seth@cse.unl.edu

Yujuan Tan
Chongqing University
Chongqing, China
tanyujuan@cqu.edu.cn

ABSTRACT

Soft error is a type of transient errors that occur due in part to reductions in capacitance and operating voltages in modern electronic components. Recently, the problem of soft errors has become more prevalent due to several design factors, including aggressive device scaling and newer energy-efficient designs, thus significantly threatening the reliability of computer systems. Since the occurrence of soft errors is non-deterministic, detecting them and recovering from them can be quite challenging. A common way to detect soft errors is to execute two identical program instances and then compare their results. Although this approach is effective, it is not efficient as both non-trivial computation and memory resources must be invested to support such redundant executions.

The introduction of hardware transactional memory (HTM) in modern chip multiprocessors (CMPs) provides an opportunity to leverage its redundant information to address the emerging reliability concerns including soft errors. In this work, we propose and implement a reliability-enhanced HTM system, called RE-HTM, which leverages redundancy to detect soft errors occurring in the L1 data cache and then recover from them. We then empirically evaluate RE-HTM and the results indicate that RE-HTM is more effective than the existing approach of running two redundant execution instances while incurring lower runtime overhead on protecting L1 data cache from soft errors.

CCS CONCEPTS

• **Computer systems organization → Parallel Architectures**; *Redundancy*; Reliability;

KEYWORDS

Soft errors, transactional memory, cache

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICPP 2018, August 13–16, 2018, Eugene, OR, USA
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6510-9/18/08…$15.00
https://doi.org/10.1145/3225058.3225093

ACM Reference Format:
Zhichao Yan, Hong Jiang, Witawas Srisa-an, Sharad Seth, and Yujuan Tan. 2018. Leverage Redundancy in Hardware Transactional Memory to Improve Cache Reliability. In *ICPP 2018: 47th International Conference on Parallel Processing, August 13–16, 2018, Eugene, OR, USA.* ACM, New York, NY, USA, Article 4, 10 pages. https://doi.org/10.1145/3225058.3225093

1 INTRODUCTION

Soft errors in computer systems can occur due to various sources including alpha particles of package decay, energetic neutrons and protons of cosmic rays, thermal neutrons, environmental random noise, and signal integrity problems. When a soft error occurs, it can lead to *silent data corruption* (*SDC*) that may affect various computer components including processors and cache memories [27]. As such, their occurrences can degrade the dependability and correctness of modern computer systems. For example, a message corrupted only by a single bit, caused Amazon to shut down all communications between its S3 servers in 2008 [1]. That particular service disruption lasted several hours, resulting in significant losses in terms of financial and productivity.

With respect to microprocessors, there are two major components that are orthogonally vulnerable to soft errors. The first component is the main processing unit plus its internal storage components such as registers and latches that are vulnerable to soft errors. The second ***component vulnerable to soft errors is L1 cache, which is the focus of this work***. Aggressive scaling of semiconductors creates opportunities for processor designers to enlarge the capacities and increase the levels of on-chip cache memories. Knowing that soft errors can occur in caches, researchers have traditionally been focusing on the simplest pattern of soft error, *single-event upset* (*SEU*) through the use of *error correction code* (*ECC*). This is especially true in the L1 cache, where ECC, due to its simplicity, is commonly used to meet the low-latency and low-power design requirements of L1 cache.

However, due to the power constraint in modern processors, power efficient design techniques such as *power gating*, *dynamic voltage and frequency scaling* (DVFS) [18, 20] are often adopted by designers to reduce the CPU's supply voltage. While these techniques can reduce energy consumption, they also make on-chip memory cells more prone to *multi-cell upset* (MCU) by energetic particles or environmental random noises. As we continue to scale

SRAM cells to finer technology nodes, the MCU rate also exponentially increases because more memory cells are likely to be affected by the strikes from energetic particles. Recently, a MCU error resulted in a corruption of several hundred bits along entire columns and rows [16]. To deal with MCU, complex ECC schemes are needed. However, as such schemes incur high space and time overheads they are often infeasible for applications in high performance and low-latency L1 data caches. Consequently, *modern L1 cache systems, despite of using ECC, can still suffer from soft errors, especially for more MCU type of soft errors.*

In this paper, we propose to leverage the inherent redundancy in HTM to protect data in the L1 data cache from the impact of soft errors. HTM is a new feature adopted by several chip manufacturers, such as Intel and IBM, in their commercial CMP productions [37, 39]. It usually provides some dedicated hardware resources to store intermediate results and only make these results visible after the transaction commits its work. If the available space redundancy during each transaction's execution is sufficient not only to detect occurrences of soft errors in *critical program execution regions* but also to recover from them by *aborting and re-executing the transaction*, we can enhance reliability of computer systems by protecting the vulnerable L1 data cache from any soft errors. *We choose to focus on L1 data cache because current Intel's HTM implementation have adopted the L1 data cache as its speculative buffer that can be further exploited to help detect and recover from soft errors. In addition, L1 cache, due to performance constraint, cannot combat multi-bit corruptions by simply employing complex ECC mechanisms, an approach commonly used in lower level cache and main memory* (more discussion is provided in Section 2).

In this study, we design a reliability-enhanced HTM system, called RE-HTM, wherein we add a function handler to the commit stage for each transaction to check whether soft errors have occurred during a transaction's lifetime. It maintains the difference ("delta") between the old and new versions of each transactional modification to protect the written data. The delta information can be used to check the equality of the old version and changes to the new version in the commit stage.

In summary, we make the following contributions in this work. (1) We introduce a novel RE-HTM scheme to leverage the redundancy existing in HTM to detect occurrences of soft errors. (2) We extend an HTM interface to integrate the function to detect and recover soft errors during the transaction's lifetime. (3) We prototype the RE-HTM system and explore various benefits and trade-offs.

The rest of the paper is organized as follows. Section 2 describes background and motivation for this work. Section 3 elaborates on the design of RE-HTM. We present the evaluation and analysis of RE-HTM in Section 4 and conclude the paper in Section 5.

2 BACKGROUND AND MOTIVATION

In this section, we provide background on soft errors and discuss the motivation for this work, which leverages specific interfaces provided by a modern HTM system to enhance reliability.

2.1 Background

Sources of Soft Errors. As the complementary metal oxide semiconductor (CMOS) feature sizes continue to shrink, the radiation from deep space, packaging materials, thermal neutrons, environmental random noise, and signal integrity problems have caused interference faults to occur at an increasing rate [24, 40]. Furthermore, the exponential increase in the number of transistors on a chip due to fabrication- technology scaling, has reduced nodal capacitances and required lower voltages. The benefits of such practice are improved performance while dissipating less energy. However, this also reduces the *critical charge* (Q_{crit}) required to excite a circuit node, making systems more susceptible to errors caused by the surrounded environmental factors. These environmental radiation-induced faults are referred to as *soft errors.*

When a soft error occurs, one or more bits in a silicon chip incorrectly flip causing the affected location to contain an erroneous result. With the current practice of integrating high-density, high-speed, low-capacitance, and low-voltage memory devices into microprocessor chips, memory's reliability, especially for the static random access memory (SRAM), commonly used to implement caches, can also suffer from such radiation-induced soft errors. As a result, ensuring reliability from soft errors becomes as a first-class design constraint for processor designers. Unfortunately, soft errors occur non-deterministically and therefore, detecting and recovering them can be very challenging.

A Study of Soft Errors in SRAM. Specifically, soft error rate (SER) for SRAM can be estimated by the Hazucha-Svensson model in Equation 1, where this SER is a linear function of the memory cell area size but an exponential function of the critical charge, where critical charge can be simplified as a linear function of the power supply voltage [11]. More specifically, the critical charge Q_{crit} can be defined as the sum of two quantities: a capacitance and a conduction component, which is shown in Equation 2. Usually, V_{dd} is the power supply voltage, C_n is the node capacitance, I_{dp} is the maximum drain conduction current of the PMOS and T_f is the flipping time of the cell. The conduction component, which is the product of I_{dp} and T_f, can be ignored in SPICE simulations to characterize the critical charge Q_{crit} [29]. As a result, SER of SRAM is an exponential function of its power supply voltage.

$$SER = Constant \times Flux \times Area \times e^{-\frac{Q_{crit}}{Q_{coll}}} \tag{1}$$

$$Q_{crit} = V_{dd} \times C_n + I_{dp} \times T_f \tag{2}$$

A recent report (shown in Table 1, one FIT represents one failure per 10^9 hours) [34] reveals that the SER for individual SRAM cells, used to construct caches, increases when the feature size is reduced from 65nm to 40nm [3]. This reversal of a long-term SER trend due to scaling is likely to continue into the future [13], making SRAM more vulnerable to soft errors. At the same time, there will be more and more MCU soft errors with the fabrication-process scaling. As such, soft errors are already a major reliability concern for both current and future high-performance CMP's cache memories.

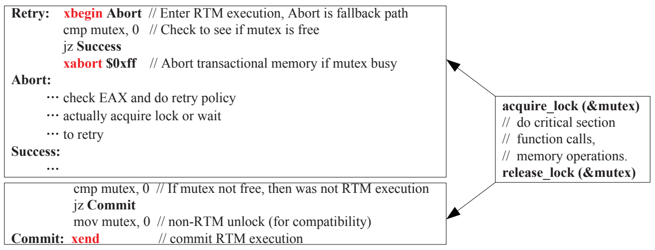
Error Correction Code (ECC). Advanced ECC techniques can protect SRAM from multiple-bit soft errors. However, the high overheads in both area and latency become a major obstacle to integrate these techniques into the L1 cache [22], because the industry generally favors simple and cheap design. In Table 2, we list the overheads of *Single Error Correct and Double Errors Detect* (SECDED)

Table 1: Soft error rates in different types of electronic component

Type	SER (FIT/bit)	SER Comment
SRAM	10^{-4}-10^{-2}	Reversal of a decline trend to an increase trend with design rule
DRAM	10^{-10}-10^{-5}	A downward trend due to fixed cell capacitance as design rule shrinks
Logic	10^{-5}-10^{-3}	Trend remaining flat as process technology shrinks

Table 2: Space overheads in different types of error correction code

Data Bits	Single Bit Correct - Double Bit Detect			Double Bit Correct - Three Bit Detect		
	ECC Bits	Total Word	Overhead	ECC Bits	Total Word	Overhead
16	6	22	27%	11	27	41%
32	7	39	18%	13	45	29%
64	8	72	11%	15	79	19%
128	9	137	7%	17	145	12%



```
Retry:   xbegin Abort  // Enter RTM execution, Abort is fallback path
         cmp mutex, 0  // Check to see if mutex is free
         jz Success
         xabort $0xff  // Abort transactional memory if mutex busy
Abort:
       ··· check EAX and do retry policy
       ··· actually acquire lock or wait
       ··· to retry
Success:
       ···
```

```
           cmp mutex, 0  // If mutex not free, then was not RTM execution
           jz Commit
           mov mutex, 0  // non-RTM unlock (for compatibility)
Commit:  xend            // commit RTM execution
```

```
acquire_lock (&mutex)
//  do critical section
//  function calls,
//  memory operations.
release_lock (&mutex)
```

Figure 1: Transactional memory interface in Intel's transactional synchronization extensions, where XBEGIN, XEND, and XABORT are new instructions used in the Restricted Transactional Memory (RTM) mode

and *Double errors Correct and Triple Errors Detect* (DECTED) under various data word length configurations. As such, L1 cache often uses simple ECC schemes, while L2 and L3 caches can possibly adopt more complex and powerful ECC schemes. In practice, however, cache designs usually sacrifice reliability for performance in addition to the constraints of power and area budget. For example, all three levels of cache in Intel's Haswell i7-4770k processor are protected by single-bit ECC scheme, and in the newer Intel's Skylake i7-6700K processor, both the L1 and L2 levels of cache are protected by single-bit ECC scheme while adopting multi-bit ECC scheme to protect its L3 cache. Single-bit ECC can only detect two bits error and correct one bit error at most. Thus, it can suffer from MCU soft errors. Interleaved ECC can improve the capability to detect and correct multi-bits errors but also consumes more power. However, interleaved ECC schemes are not capable of handling soft errors that corrupt several hundred bits along entire columns and rows [16].

In summary, *the high SER of SRAM in conjunction with the use of simple ECC schemes makes L1 cache especially vulnerable to soft errors, especially for more and more MCU soft errors in the future.*

Detection and Recovery from Soft Errors. Recent research focus has been on protecting the processor's internal structures from soft errors [6, 8, 26, 28, 32, 36]. However, the result of the aforementioned SER studies and a common practice in modern cache design indicate that more efforts should be spent on protecting data in the L1 data cache from soft errors. This subsection focuses on existing work on detecting and recovering from soft errors, which can be categorized into hardware- and software-based solutions, to address this need. Both categories define their replicated spheres and use redundant information to detect and recover the corrupted data [26].

Hardware-based solutions usually replicate inputs while collecting outputs to make comparisons to detect soft errors. Specific forward or backward retry policies are used to facilitate the recovery from soft errors [32]. Earlier systems utilized complex redundant hardware components such as dual modular redundancy (DMR) or triple modular redundancy (TMR) to meet system reliability requirements. Solutions based on simultaneous multithreaded (SMT) processors such as FaulTM [38], can achieve redundancy by executing two identical copies of the same program as independent threads. The outputs are then compared to detect transient faults [28, 36]. Generally approaches that run redundant threads or transactions, require double theprocessing resources.

In addition there are software based solutions that execute redundant threads in SMT processor. Software Anomaly Treatment (SWAT) [19], can monitor abnormal software behaviors, diagnose these behaviors to identify transient faults, and recover from them [33]. At the same time, these solutions can analyze the structure of a program to assess possible impacts of soft errors and then select test cases to detect and recover from soft errors [6, 8]. As such, the software-based solutions are more practical, as they do not need any hardware modification. However, they can be slow and require manual injection of monitoring code.

2.2 Motivation

Modern Transactional Memory Systems. Transactional memory (TM) can simplify concurrent programming by organizing a group of load and store instructions to execute in an atomic way. After more than two decades of development, TM finally enjoys mainstream adoption by the microprocessor industry. HTM systems provide a simple and clean interface to provide atomicity, consistency and isolation. They automatically isolate each transaction and provide programmers with the ability to compose complex code sections using the provided TM interfaces. As a result, the whole transaction behaves as an atomic operation, which can guarantee the serializability and recoverability. Such atomicity provides an ideal environment to integrate reliability enhanced feature as part of TM interfaces. This can be achieved without interfering with current capability to detect and recover from atomicity violations.

In Intel's Transactional Synchronization Extensions (TSX), which we modeled in our implementation, instructions such as `XBEGIN` and `XEND` are used to label a transactional region [14]. It organizes the intermediate results in its internal speculative buffer in the L1 cache before exposing them to the concurrent transactions. Instruction `XABORT` is used to abort a transaction and roll back the processor's state to that before the execution of `XBEGIN`. Note that TSX only provides transactional support at the L1 cache (i.e. transaction with its modifications overflow L1 cache will be aborted), which is sufficient for our work to protect vulnerable data in the L1 data cache, and our RE-HTM can be extended to protect L2 cache when L2 cache is included in HTM's speculative buffer in future.

In order to maintain transactional semantics (atomicity, consistency, and isolation), any memory modification in a transaction cannot be accepted until the transaction commits its work (e.g., `XEND` in TSX) [12, 30]. Thus, for each modified data in a transaction, there are at least two versions of this data: the original version without any transactional modifications and a new version resulting from a transactional modification plus some register state information. This redundant information is used to revert to the state before transaction starts its work on *abort* and commit the transactional computations on *commit*, respectively. A transactional conflict occurs when two or more concurrent memory accesses are issued to the same shared data, of which at least one access comes from a transaction and one of them is a write operation [10]. HTM systems can then detect any transactional conflicts and schedule the conflicted transactions in cache coherence protocol to guarantee correctness.

From this perspective, existing HTM systems have already integrated several hardware resources to help detect and resolve conflicts, and then make decisions to commit or abort transactions. In Intel's TSX, the conflict resolution mechanism has been further extended to provide programmable fallback paths so that developers can specify tasks to perform based on reported abort codes. *Throughout this paper, we borrow the same TM interface as TSX to integrate the capability to detect and recover corrupted data due to soft errors.* To illustrate how this fallback mechanism works in TSX, we present Figure 1, which shows how traditional lock-based code can be translated into a TM-based code in TSX.

In Figure 1, a program enters a transaction via `XBEGIN`. Its fallback path is located at the address of `Abort`. In this example, the abort code is set to "0xff" in a specified register (e.g., EAX) and the abort handler checks this code to perform retry or wait for other prerequisites before responding [14]. If no conflicts are detected, the transaction commits its work via `XEND`. Note that because the L1 cache is used as a speculative buffer, transactions overflow from L1 cache are aborted and re-executed through the controls of software handlers.

The availability of a fallback mechanism means that we can extend existing TM interfaces to check whether any soft errors had occurred during a transaction's lifetime. We can further optimize the HTM structures to maintain all necessary redundancy information to protect transactional data in the L1 data cache. This added functionality can be supported by existing TM interface without interfering with its ability to protect accesses by concurrent threads. As a result, we propose that modern TM systems can be used as a low-cost reliability solution to improve the on-chip memory reliability.

3 SYSTEM DESIGN AND IMPLEMENTATION

In this section, we describe our design and implementation of the proposed RE-HTM system. Specifically, we define the *replicated sphere* to protect the data in the vulnerable L1 data cache. We exploit existing redundancy in the version management module of HTM to protect the vulnerable data. The ultimate goal of the proposed system is improving system reliability at a fairly low cost.

3.1 Architectural Overview

RE-HTM is an enhancement to a typical CMP system with two level on-chip caches. The organization is based on a private L1 data cache per core and a shared inclusive L2 cache per chip. It supports ***lazy conflict detection*** and ***lazy version management*** transactional memory features as part of its hardware cache coherence protocol. Currently, RE-HTM only supports the ***inclusive cache*** because it needs to leverage the lazy version management feature. Specifically, it utilizes the L1 data cache to buffer the transactional writes at the cache line granularity during a transaction's execution. It then checks whether the memory access operations overlap with other executing transactions to detect any transactional conflict. The buffered content is either committed or aborted at the end of the transaction. When a transaction begins to access a dirty cache line, the system writes the dirty cache line back to the inclusive L2 cache. This is done to maintain an old copy of data in the L2 cache. Dirty data in the L2 cache is then written back to the main memory when it is evicted from the L2 cache. A transaction is aborted when its modifications are evicted from the L1 data cache. Our approach is designed to protect the vulnerable L1 cache. We also assume that the lower level memory such as L2 cache and main memory have already been protected by more complex ECC schemes. This is because these lower level memory systems can usually tolerate longer latency to achieve higher reliability [7, 15, 17].

We implemented the aforementioned system by extending the Intel's TSX interface in our simulated CMP system to support detection and correction of soft errors. We used the same instruction prefixes as TSX throughout this paper to make our design fully compatible with this interface. Specifically, we modified the `XEND` instruction to check for any occurrence of soft errors within a transaction's lifetime. Figure 2 compares the approaches used by the current state-of-the-art FaulTM and our proposed system. On the left side, FaulTM spawns a new thread running the redundant transaction to be used to check for transient faults. This requires that both threads are synchronized and a comparison is performed just prior to commit [38]. Our RE-HTM approach, on the other hand, simply detects any occurrence of soft errors in the commit stage without the need to spawn another redundant thread. During the execution, our approach injects redundant copy of data to protect (the selection process is explained in section 3.2).

When a transaction enters in its commit stage, any data inconsistency within a transaction indicates that soft error has occurred. When this is detected, the transaction is aborted. The system then re-executes the transaction to recover from an error and determine whether the error is caused by a transient or a permanent hardware error. The re-execution control can be done by simply extending the specific abort handler. While permanent hardware damage can also be checked by this process, we omit it in this work because it only focuses on its capability to detect and correct the corrupted data when any soft error occurs.

3.2 Replicated Sphere

In order to detect occurrences of soft errors within a transaction, the system needs to check contents of the protected variables in the replicated sphere and verify that there is no data corruption occurring within the transaction's lifetime.

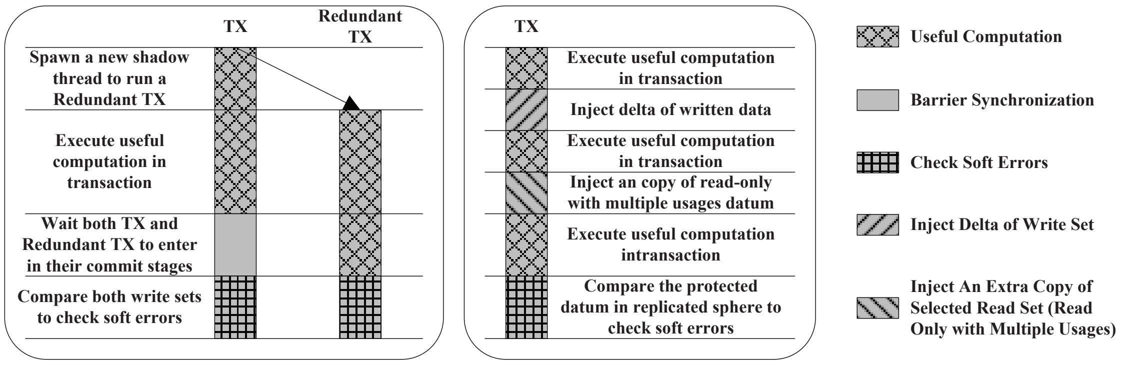

TX
Redundant TX
Spawn a new shadow thread to run a Redundant TX
Execute useful computation in transaction
Wait both TX and Redundant TX to enter in their commit stages
Compare both write sets to check soft errors
TX
Execute useful computation in transaction
Inject delta of written data
Execute useful computation in transaction
Inject an copy of read-only with multiple usages datum
Execute useful computation intransaction
Compare the protected datum in replicated sphere to check soft errors
Useful Computation
Barrier Synchronization
Check Soft Errors
Inject Delta of Write Set
Inject An Extra Copy of Selected Read Set (Read Only with Multiple Usages)

Figure 2: Execution mode comparison between FaulTM and the proposed reliability-enhanced HTM system

As shown in Figure 3, we have to protect all written data within a transaction to avoid any data corruption happens during the transaction lifetime. We do not protect any read-only memory content that is only read once (e.g., condition variables) within a transaction because it can be re-fetched from the lower level memory. However, when a read-only memory content is loaded multiple times, we need to maintain its correct value in the L1 data cache for subsequent accesses as some transactional read-only data may be silently evicted from the L1 cache. Therefore, our replicated sphere also includes each read-only variable in a transaction when it is loaded more than once.

Currently, HTM provides limited virtualizing support for transactions, which may induce the transactional abortions. As such, programmers need to be tactful to avoid such unnecessary transactional aborts. In this work, we disabled the hardware prefetcher from loading data from the lower-level memory into the L1 cache. Such prefetching can increase the number of aborts due to the resource contention. To achieve this goal, for the variables that are used by the previous transactions, we combine the commit operations of the previous transactions to write back the values to the lower level cache and invalidate the copy in the L1 cache. We also extended the lazy commit approach used by Intel's TSX to commit written data and apply it to all data (including previously loaded data).

Soft error detection requires that we collect the write-set content at each write operation and compare it with the latest content at the commit time. For read-only data, we need to verify that the two copies are the same. These different types of data can be analyzed by their dependence relationship within the transaction code. Therefore, the compiler can help to identify the data that we need to protect.

Figure 4 illustrates the process to detect occurrences of soft errors for transactional writes. As shown, our modified version-management module computes the difference between the new and the old versions of each transactional modification (*delta*) and then stores the new version in the L1 data cache. The old and delta values are stored in the L2 cache. In the commit stage, it compares these values to check whether a soft error has occurred. It is coupled with the lazy conflict detection and lazy version management of the hardware transactional memory. When a transaction finishes its work, it sends its read and write addresses to the cache controller

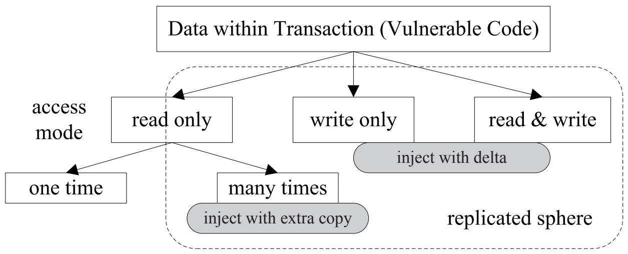


Figure 3: Replicated sphere defined in reliability enhanced HTM system

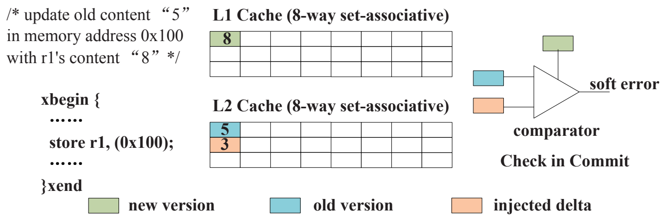

/* update old content "5" in memory address 0x100 with r1's content "8" */
xbegin {
……
store r1, (0x100);
……
}xend
L1 Cache (8-way set-associative)
8
L2 Cache (8-way set-associative)
5
3
soft error
comparator
Check in Commit
new version
old version
injected delta

Figure 4: Integrated management of delta value in version-management module

and perform conflict detection. If no transactional conflict is detected, the transaction commits its modifications in the L1 cache. We also modified the cache coherence protocol to help the L2 cache hold the old and delta values. It can be extended to permit the old and delta values to be swapped to the main memory by maintaining some tables to track the information. For each read-only memory content with multiple accesses within a transaction, the version-management module stores an extra copy in the L2 cache. Note that the inclusive L2 cache already has this capability. It can also check for equality when any data is evicted from the L1 cache or in the commit stage to guarantee that the data in L1 is not corrupted. Because our base system implements an inclusive L2 cache which contains all the contents reside in the L1 data cache, we simply added an extra compare operation in the commit stage or when some read set data is silently evicted from the L1 data cache. As a result, we can use the same comparator circuit with the delta input port to be set to "0".

When a transaction commits its work, the content of the write-set is written to the L2 cache or the main memory and becomes globally visible to the whole system. We have added a compare and check operation to this stage. It involves verifying the expected data values in the replicated sphere. An equal write-set entry in it is marked as ready to be committed. Subsequently, the system commits the transactional computation. Otherwise, it aborts and restarts the transaction to recover from soft error. This compare-and-check operation is implemented as an extended soft-error handler.

3.3 Optimizations

We only use (*delta*) to check for value equality, and therefore, we do not need to store its contents. Storing the delta value instead of the new data can save memory. Because our system uses a counting Bloom filter [5] to calculate the summary signature to represent the delta content of the write-set instead of storing the delta value for each entry in the L2 cache, using delta can also reduce the false positives of Bloom filter. As shown in Figure 5, we integrated a counting Bloom filter to store delta contents. It is worth noting that when multiple write operations are performed on the same variable within a transaction, our system removes the old delta before adding the new delta content to the counting Bloom filter. This operation can be determined by checking whether the written variable exist in the write set.

We choose a 4-bit counter to avoid possible overflow of the counting Bloom filter. With 4-bit, the overflow probability is less than $1.37 \times 10^{-15} \times m = 2.80 \times 10^{-12}$, where m is the vector size of the Bloom filter, chosen to be 2048 in our system. This signature-based design removes the extra space overheads at a cost of a counting Bloom filter. In the commit stage, the system performs an XOR operation of the new and old values of the counting Bloom filter to detect any occurrence of soft errors. In addition to reducing space in the L2 cache, this approach also reduces the write traffic to the L2 cache.

Because the L1 cache is used as a speculative buffer, a transactional write-set overflow triggers a transaction abortion. This type of abort is due to the limited support on virtualizing transactional memory. As such, developers need to design alternative execution paths to handle the various aborting cases. When a transaction degrades to software execution mode, it executes like existing software-based methods that maintain data redundancy in software and compare results to check for soft errors. Programmers need to specify the execution path in this case. To mitigate the effect of aborts due to overflowed transactions, we suggest that developers divide each large transaction into several smaller transactions, each of which can fit in the L1 data cache. It is the developer's responsibility to guarantee the functional equivalence between an original large transaction and the subsequent smaller transactions.

Another suggestion that can improve performance is to avoid having unsupported system calls and I/O operations in transactions. These instructions can induce transactional abortions. These kind of aborts come from the inherent problem of limited hardware support to virtualing transactional memory. For example, running a transaction several times may results in several aborts due to the OS related activities. The following work [31] has been implemented to alleviate this type of abort. However, since the adoption of HTM in CMP is still in its infancy, this type of abort must still be avoided by the programmers.

In the commit stage, each transaction may experience a time delay that can leave previously compared values (i.e., already checked for soft errors) unprotected until the transaction is fully committed. Fortunately, most transaction sizes are relatively small due to limited hardware resource for HTM. Small transactions mean smaller delay windows. Still we choose to tackle this issue by designing a greedy policy that commits an equal entry to the L2 cache without waiting for all the compare-and-check operation to finish. While the system still needs to maintain the old value to allow the system to roll back once an occurrence of soft error is detected, this optimization increases the performance of the common case; i.e., transactions not experiencing soft errors. This is done to reduce commit delays caused by comparing and checking all entries. We design a specific software handler to revert the old value in DRAM, when we find the occurrence of soft error during the process to greedily commit the equal entries. At the same time, we design a hardware component in the commit stage to accelerate the compare-and-check operation, which is shown in Figure 4.

3.4 Overheads

This approach only applies to systems with HTM support. Currently, there are vendors such as IBM and Intel, who include HTM features with their CMPs. RE-HTM is based on their limited HTM support and adds the counting bloom filter and comparators that constitute the additional hardware overheads. The hardware comparators' overhead is trivial. Implementing the 4-bits counting bloom filter with vector size of 2048 needs 1KB of SRAM. As a result, it adds about 3.1% overheads on SRAM comparing with 32KB L1 data cache, which is much less than the space overheads in different types of error correction code as shown in Table 2

4 SYSTEM EVALUATION AND ANALYSIS

In this section, we evaluate the overhead of our proposed RE-HTM system with the base system and compare it against the traditional redundant execution methods, such as FAULTM.

4.1 Experimental Environment

We use Simics [21] to simulate a 16-core CMP system with HTM functionality based on GEMS 2.1 [23]. These 16 cores are interconnected via an on-chip network with the mesh topology. We configure the 4 memory controllers integrated in the system to access the main memory. We model the CMP system's HTM feature similar to Intel's TSX extension. The L1 data cache adds extra bits to label the transactional read-set and write-set, working as the transactional buffer, while the L2 cache adopting the interleaved DECTED code to correct double errors and detect triple errors, the L2 cache contains all the datum stored in the L1 data cache. The detailed configuration parameters are listed in Table 3. Main memory also adopts the interleaved ECC scheme to guarantee the data's correctness from corruption.

There are two orthogonal components of protecting the L1 data cache from soft errors. The first component is to analyze the program and identify the vulnerable data. The second component is to propose some mechanism to protect the vulnerable data in the

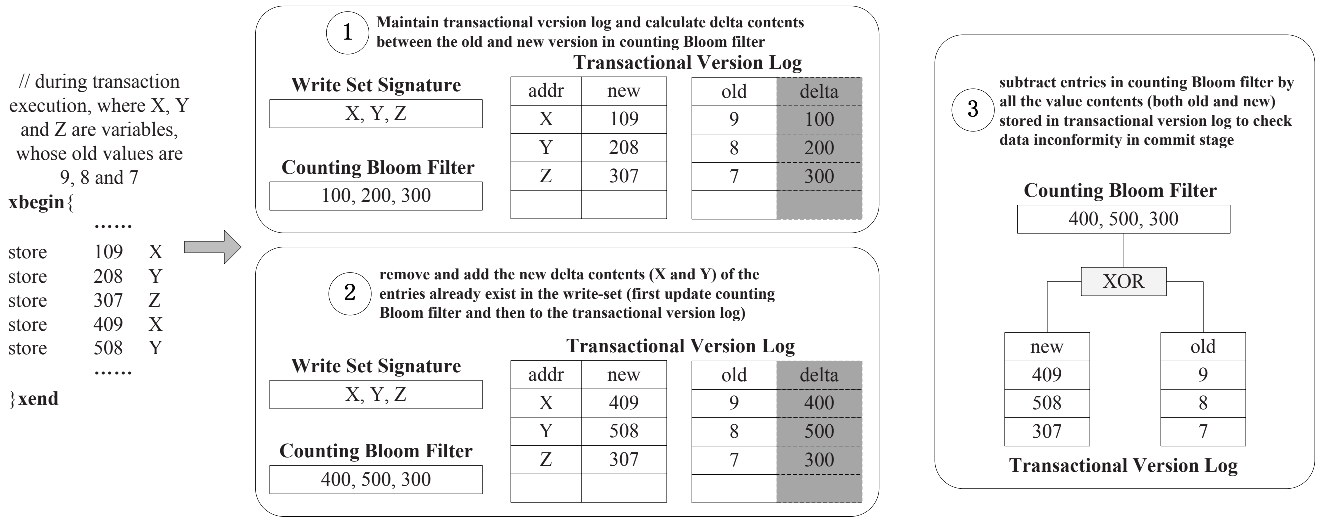

// during transaction execution, where X, Y and Z are variables, whose old values are 9, 8 and 7
xbegin{
……
store 109 X
store 208 Y
store 307 Z
store 409 X
store 508 Y
……
}xend
1 Maintain transactional version log and calculate delta contents between the old and new version in counting Bloom filter
Write Set Signature
X, Y, Z
Counting Bloom Filter
100, 200, 300
Transactional Version Log
addr new old delta
X 109 9 100
Y 208 8 200
Z 307 7 300
2 remove and add the new delta contents (X and Y) of the entries already exist in the write-set (first update counting Bloom filter and then to the transactional version log)
Write Set Signature
X, Y, Z
Counting Bloom Filter
400, 500, 300
Transactional Version Log
addr new old delta
X 409 9 400
Y 508 8 500
Z 307 7 300
3 subtract entries in counting Bloom filter by all the value contents (both old and new) stored in transactional version log to check data inconformity in commit stage
Counting Bloom Filter
400, 500, 300
XOR
new 409 508 307
old 9 8 7
Transactional Version Log

Figure 5: Store delta contents in counting Bloom filter

Table 3: Configuration of The Simulated CMP System

Processor Core	2.0 GHz in-order, single issue
L1 Cache	32 KB 8-way, 64-byte line, write-back, 2-cycle latency
L2 Cache	8 MB 8-way, inclusive, write-back, 20-cycle latency
Main Memory	4 GB, 4 banks, 300-cycle latency
L2 Directory	Bit vector of sharers, 6-cycle latency
Interconnect	Mesh, 2-cycle wire latency, 1-cycle route latency

code snippets. In this work, we provide support for the second component by leveraging HTM to protect L1 data cache from soft errors. Program analysis techniques can help programmers to locate data with specific features and then to protect such data in transactions to guarantee correctness. Developing effective approaches to identify code locations vulnerable to soft errors still needs more investigation, such as exploiting the program vulnerability factor [9, 35], and we will explore it as our future work. In this work, however, we rely on the developer to identify the vulnerable data in a program. To do so, the input program is transformed into several snippets and the vulnerable snippets, as specified by the developers, are protected by HTM as transactions.

Note that the importance of various data sometimes depends on their access pattern. For example, a branch condition that is only used once can be safely loaded from L2 cache or main memory, and thus, needs no protection. This type of data can be located within a program by using existing static and dynamic program analyses to identify their resiliency [6, 8]. On the other hand, variables that are used to control conditional branches, frequently read, or shared by multiple threads are instances of such critical and vulnerable data that need protection. Similar to FaulTM, our system expects programmers or compilers to generate code that protect vulnerable data by including them as part of transactions.

Because our focus is not on developing approaches to identify these critical data, we, instead, choose the STAMP benchmark suite [25] to conduct our evaluation to learn the overheads and benefits of RE-HTM system. The STAMP benchmark suite already provides locations of one type of critical data—variables shared by multiple threads. In doing so, we can assume that all identified transactions represent code regions vulnerable to soft errors since data in these transactions are shared by multiple threads. In Table 4, we summarize the key characteristics of the selected applications from the STAMP benchmark suite. The transactional read-set and write-set are maintained at the cache line granularity. Note that we excluded the Bayes application in our evaluation because it has exhibited unpredictable behavior and high variability in its execution time [4]. We choose the baseline TM system to be the one with lazy version management and lazy conflict management, which behaves like Intel's TSX implementation.

We find that the STAMP benchmark suite already has several applications that spend more than 95% of the overall execution time in the transactional code. This means that our proposed system is being tested under rigorous workload (i.e., a large number of transactions and data to be protected from the soft errors). For applications that spend less time in transactional codes, we can use them to evaluate the overheads of protecting only a small portion of the code.

To inject soft errors, we use Simics to dynamically change the L1 cache content. We then record the changed locations to ensure repeatability. A prior study has shown a convincing evidence that 99.78% transient faults occurring in selected programs can be pruned to exploit their equivalences [8]. Based on this important observation, we simplify our soft error injection model by randomly injecting soft errors to 5% of the transactions in these applications, which represent on average 0.0015% to 4.95% of the overall execution time in our selected applications.

4.2 Evaluation Results and Analysis

As part of our overhead analysis, we decompose the overall execution time into the following non-overlapped components: times due to non-transactional work (NoTrans), transactional work (Trans), barrier synchronization (Barrier), conflict resolution (Conflict), and detection of soft errors (Check). The "Conflict" time only exists in

Table 4: Workload Characteristics of Benchmarks

	Input Parameters	Granularity	Contention	Read-set		Write-set	
				avg	max	avg	max
intruder	-a10 -l4 -n2038 -s1	fine	high	5.5	30	3.4	23
kmeans	-m40 -n40 -t0.05 -i random-n2048-d16-c16.txt	fine	low	3.8	7	1.7	2
ssca2	-s13 -i1.0 -u1.0 -l3 -p3	fine	low	2.0	3	2.0	2
vacation	-n4 -q60 -u90 -r16384 -t4096	middle	low	56.9	99	8.2	18
labyrinth	-i random-x32-y32-z3-n64.txt	coarse	high	91.1	311	90.7	252
genome	-g256 -s16 -n16384	coarse	high	24.1	234	5.9	151
yada	-a20 -i 633.2	coarse	high	37.3	421	30.2	370

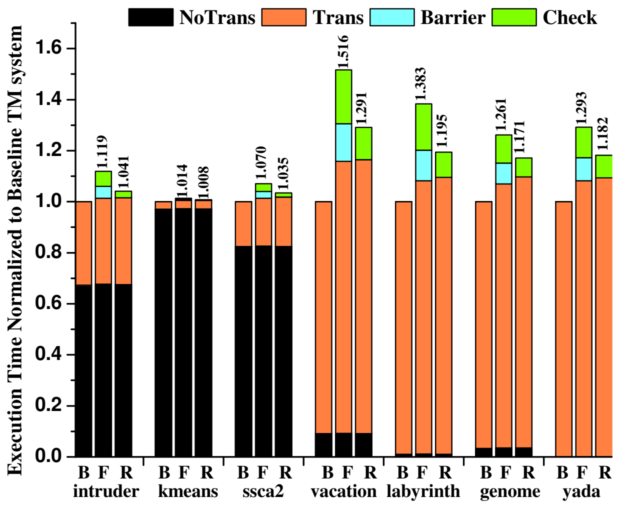


Figure 6: Distribution of execution times under the single-threaded configuration. B, F and R denote the baseline TM system, the FaulTM system, and the proposed signature-based RE-HTM system, respectively.

multi-threaded configurations, while the "Check" time only exists in FaulTM and the RE-HTM system.

We compare the overheads of our RE-HTM with those of FaulTM on protecting the L1 cache because it is the most relevant work to our approach. While FaulTM has some potential capability to protect some errors in the processor datapath. However, it lacks the evaluation on the coverage of datapath. More specifically, without the lockstep technique [2], we believe neither FaulTM nor our approach can really protect the core data path. The reason is that, besides the modifications in the L1 cache, FaulTM can only compare the register files that cannot protect the latches within the core (lacking the synchronization mechanism to compare the results). As a result, a practical approach is to separating different protection mechanisms on different components. As in our work, we only use RE-HTM to protect the L1 cache from soft errors.

As illustrated in Figure 6, we measured the execution times of seven applications in the single-threaded configuration on three different systems: baseline HTM (B), FaulTM (F) and counting Bloom filter signature-based (R) RE-HTM systems. In this mode, we find that "R" outperforms "F" and only incurs an average overhead of 12.8% when compared to the baseline system; "F", on the other hand, incurs an average overhead of 22.5%. Note that FaulTM needs to spawn a redundant reliable thread to execute the replicated transaction.

For those applications spending most of their time in transactions, "R" spends more time on "Check" because these applications usually have bigger read-set and write-set, requiring more time to check protected data. Furthermore, "R" spends more time on "Trans" (0.003% to 0.16%) to maintain the necessary redundancy information, and "F" also spends more time on "Trans" because it needs to wait for its replicated reliable transaction to finish, whereas, our approach calculates the longer one as its "Trans" part.

Some applications have barriers to synchronize transactions when entering the commit stage. "F", by running two threads, spends non-trivial amount of time on "Barrier" and "Check". For those applications spending less time in transactions, the overheads of "R" are impacted by the size of the transactional read-set and write-set.

To observe our RE-HTM system's behaviors under multithreaded settings, we evaluate these systems under 2-, 4-, 8- and 16-thread configurations, respectively. Here we choose the applications "vacation", "labyrinth", "gnome" and "yada", which spend most of their time in transactional regions. Due to their high densities of transactions, these applications would likely exhibit different behaviors than those in the single-threaded setting as more transactions are executed in parallel. Table 5 reports the speedups of "B", "F" and "R" under the specified multi-threaded configurations. We find that "R" can achieve higher speedups than "F" across various configurations because it doesn't require the system to spawn an identical thread to check for soft errors, thus freeing extra threads to execute useful computation.

In Figure 7, we also decompose the overall execution times into the five components as previously described. Because "F" needs an extra thread to replicate each transaction, the recorded results begin with 2-thread configuration. From this part, we find that "R" performs much better than "F" under the same multi-threaded configurations. It even outperforms "F" when it employs fewer threads than "F".

As the number of threads increases, we observe that "Trans" and "Check" decrease accordingly. This is because the transactional regions are executed in parallel so more transactions can be executed in a given time. At the same time, we observe that "Barrier" and "Conflict" also increase. This is because concurrent transactions incur more overheads on barrier synchronizations and conflict resolutions. This is especially true for "R", in which soft error detection is done by having both the transaction (TX) and reliable threads enter the commit stage. In some execution instances, TX might try

Table 5: Execution Time Speedups under Multi-threaded Configurations

	1-thread			2-thread			4-thread			8-thread			16-thread		
	B	F	R	B	F	R	B	F	R	B	F	R	B	F	R
vacation	1.00	N/A	0.77	1.66	0.66	1.36	2.40	1.13	1.96	3.01	1.78	2.56	3.45	2.40	3.01
labyrinth	1.00	N/A	0.84	1.07	0.72	0.90	1.13	0.86	0.99	1.29	0.96	1.15	1.44	1.14	1.26
genome	1.00	N/A	0.85	1.68	0.79	1.45	2.80	1.39	2.42	3.97	2.28	3.42	4.90	3.28	4.31
yada	1.00	N/A	0.85	1.41	0.77	1.23	2.31	1.31	2.00	3.00	1.99	2.43	3.54	2.39	3.00

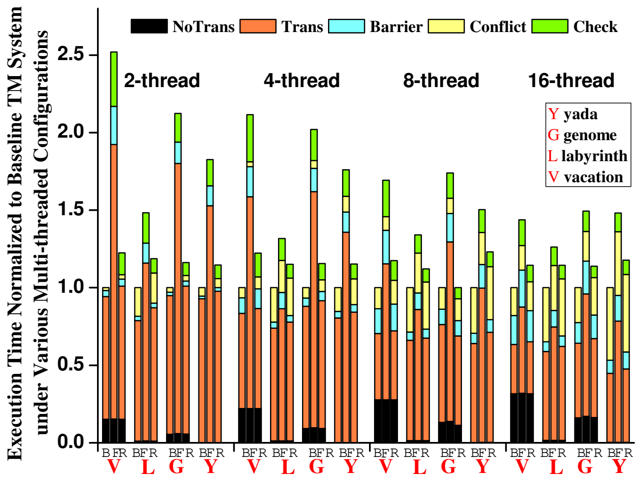


Figure 7: Execution times of selected applications under various multi-threaded configurations

to enter the commit stage first. At this point, it must wait for the Redundant TX to also enter the commit stage. In doing so, other threads can conflict with the waiting TX. In this scenario, both TX and Redundant TX must be aborted. We find this scenario to occur frequently enough to result in significant increase in both "Barrier" and "Conflict" overheads for "F".

In summary, we find that employing more threads does not result in significant speedups in most applications, which is dependent on the characteristics of the benchmark suite. However, our design incurs much less overhead than "F" while employing a smaller number of threads. This indicates that our RE-HTM is more execution and space efficient.

5 CONCLUSION AND FUTURE WORK

In this work, we propose a reliability-enhanced HTM system that exploits this feature to protect L1 data cache from soft errors. Our solution protects vulnerable data by enclosing them in transactions. We define the replicated sphere to maintain redundancy information and extend existing RTM interface to integrate the functionality of detecting soft errors and recovering from them. We compare the performance of our RE-HTM system to that of FaulTM, a state-of-the-art HTM based soft errors detection system and find that our approach is as effective as FaulTM in terms of soft errors detection and recovery but incur much lower overhead. Our approach consistently achieves higher speedups than FaulTM under the same number of executing threads because our method does not employ threads to replicate execution.

As for future work, we plan to evaluate the effectiveness of our system in detecting permanent and intermittent errors. We will evaluate our designs using more real-world applications in addition to these transactional memory applications. We will apply program analysis techniques to help automatically locate code sections that have potential to be more vulnerable to soft errors, for example, we can give higher priority to protect the code sections who has a higher probability suffering from the silent data corruption. At the same time, the framework of this RE-HTM system is easy to extend to detect any dynamical abnormal behaviors during the runtime. We plan to exploit redundancy information existing in HTM system's checkpoint and transactional log to address other reliability concerns including soft errors occurring in processing units. Lastly, we plan to use this framework to detecting the concurrency errors.

ACKNOWLEDGMENT

We would like to thank the anonymous reviewers for their valuable feedbacks and constructive suggestions. This research is partially supported by National Natural Science Foundation of China (NSFC) under Grant No.61402061, Chongqing Basic and Frontier Research Project of China under Grant No.cstc2016jcyjA0274, research grant from NetApp and the U.S. National Science Foundation (NSF) under Grant Nos. CCF-1704504 and CCF-1629625. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding agencies.

REFERENCES

[1] [n. d.]. Amazon S3 Availability Event: July 20, 2008. Retrieved May 30 2018 from http://status.aws.amazon.com/s3-20080720.html
[2] M. Baleani, A. Ferrari, L. Mangeruca, A. Sangiovanni-Vincentelli, Maurizio Peri, and Saverio Pezzini. 2003. Fault-tolerant Platforms for Automotive Safety-critical Applications. In *Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*. 170–177.
[3] A. Dixit and Alan Wood. 2011. The Impact of New Technology on Soft Error Rates. In *Proceedings of the International Reliability Physics Symposium (IRPS)*. 5B.4.1–5B.4.7.
[4] Aleksandar Dragojevic and Rachid Guerraoui. 2010. Predicting the Scalability of an STM A Pragmatic Approach. In *Proceedings of the Workshop on Transactional Computing (TRANSACT)*.
[5] Li Fan, Pei Cao, and et.al. 2000. Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. *IEEE/ACM Transactions on Network* 8 (2000), 281–293. Issue 3.
[6] Shuguang Feng, Shantanu Gupta, Amin Ansari, and Scott Mahlke. 2010. Shoestring: Probabilistic Soft Error Reliability on the Cheap. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 385–396.
[7] S. L. Gong, M. Rhu, J. Kim, J. Chung, and M. Erez. 2015. CLEAN-ECC: High reliability ECC for adaptive granularity memory system. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.
[8] Siva Kumar Sastry Hari, Sarita V. Adve, Helia Naeimi, and Pradeep Ramachandran. 2012. Relyzer: Exploiting Application-level Fault Equivalence to Analyze Application Resiliency to Transient Faults. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating*

Systems (ASPLOS). 123–134.

[9] Siva Kumar Sastry Hari, Timothy Tsai, Mark Stephenson, Steve Keckler, and Joel Emer. 2015. SASSIFI: Evaluating Resilience of GPU Applications. In *The 11th Workshop on Silicon Errors in Logic - System Effects (SELSE)*.

[10] T. Harris, J. Larus, and R. Rajwar. 2010. *Transactional Memory (2nd edition)*. Morgan & Claypool.

[11] P. Hazucha and C. Svensson. 2000. Impact of CMOS Technology Scaling on The Atmospheric Neutron Soft Error Rate. *IEEE Transactions on Nuclear Science* 47, 6 (2000), 2586–2594.

[12] M. Herlihy and J. Moss. 1993. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 289–300.

[13] E. Ibe, H. Taniguchi, Y. Yahagi, K. Shimbo, and T. Toba. 2010. Impact of Scaling on Neutron-Induced Soft Error in SRAMs From a 250 nm to a 22 nm Design Rule. *IEEE Transactions on Electron Devices* 57 (2010), 1527–1538.

[14] Intel. 2012. Intel Architecture Instruction Set Extensions Programming Reference.

[15] D. W. Kim and M. Erez. 2016. RelaxFault Memory Repair. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*.

[16] Jangwoo Kim, Nikos Hardavellas, Ken Mai, Babak Falsafi, and James Hoe. 2007. Multi-bit Error Tolerant Caches Using Two-Dimensional Error Coding. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 197–209.

[17] J. Kim, M. Sullivan, and M. Erez. 2015. Bamboo ECC: Strong, safe, and flexible codes for reliable computer memory. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*.

[18] Wonyoung Kim, M.S. Gupta, Gu-Yeon Wei, and D. Brooks. 2008. System Level Analysis of Fast, Per-Core DVFS Using On-Chip Switching Regulators. In *Proceedings of the 14th IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 123–134.

[19] Man-Lap Li, Pradeep Ramachandran, Swarup Kumar Sahoo, Sarita V. Adve, Vikram S. Adve, and Yuanyuan Zhou. 2008. Understanding the Propagation of Hard Errors to Software and Implications for Resilient System Design. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 265–276.

[20] P. Macken, M. Degrauwe, M. Van Paemel, and H. Oguey. [n. d.]. (A Voltage Reduction Technique for Digital Systems. In *Proceedings of the 37th IEEE International Conference on Solid-State Circuits (ISSCC)*.

[21] P. Magnusson, M. Christensson, and et.al. 2002. Simics: A Full System Simulation Platform. *Computer* 35 (2002), 50–58.

[22] Mehrtash Manoochehri, Murali Annavaram, and Michel Dubois. 2011. CPPC: Correctable Parity Protected Cache. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA)*. 223–234.

[23] M. Martin, D. Sorin, and et.al. 2005. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *SIGARCH Computer Architecture News* 33 (2005), 92–99. Issue 4.

[24] T.C. May and Murray H. Woods. 1979. Alpha-Particle-Induced Soft Errors in Dynamic Memories. *IEEE Transactions on Electron Devices* 26, 1 (1979), 2–9.

[25] C. Minh, J. Chung, and et.al. 2008. STAMP: Stanford Transactional Applications for Multi-Processing. In *Proceedings of the International Symposium on Workload Characterization (IISWC)*. 35–46.

[26] Shubu Mukherjee. 2008. *Architecture Design for Soft Errors*. Morgan Kaufmann Publishers Inc.

[27] S.S. Mukherjee, J. Emer, and S.K. Reinhardt. 2005. The Soft Error Problem: An Architectural Perspective. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*. 243–247.

[28] S.S. Mukherjee, M. Kontz, and S.K. Reinhardt. 2002. Detailed Design and Evaluation of Redundant Multithreading Alternatives. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 99–110.

[29] R. Naseer, Y. Boulghassoul, J. Draper, Sandeepan DasGupta, and A. Witulski. 2007. Critical Charge Characterization for Soft Error Rate Modeling in 90nm SRAM. In *Proceedings of the 40th IEEE International Symposium on Circuits and Systems (ISCAS)*. 1879–1882.

[30] Ravi Rajwar and Martin Dixon. 2012. Intel Transactional Synchronization Extensions. In *Intel Developer Forum*.

[31] Ravi Rajwar, Maurice Herlihy, and Konrad Lai. 2005. Virtualizing Transactional Memory. In *Proceedings of the 32Nd Annual International Symposium on Computer Architecture (ISCA)*. 494–505.

[32] Steven K. Reinhardt and Shubhendu S. Mukherjee. 2000. Transient Fault Detection via Simultaneous Multithreading. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 25–36.

[33] 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 (MICRO)*. 122–132.

[34] C. Slayman. 2011. Soft Error Trends and Mitigation Techniques in Memory Devices. In *Proceedings of the Annual Reliability and Maintainability Symposium (RAMS)*. 1–5.

[35] Vilas Sridharan and David R. Kaeli. 2010. Using Hardware Vulnerability Factors to Enhance AVF Analysis. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA)*. 461–472.

[36] T. N. Vijaykumar, Irith Pomeranz, and Karl Cheng. 2002. Transient-Fault Recovery Using Simultaneous Multithreading. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 87–98.

[37] Amy Wang, Matthew Gaudet, Peng Wu, José Nelson Amaral, Martin Ohmacht, Christopher Barton, Raul Silvera, and Maged Michael. 2012. Evaluation of Blue Gene/Q Hardware Support for Transactional Memories. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 127–136.

[38] Gulay Yalcin, Osman Unsal, and Adrian Cristal. 2013. FaulTM: Error Detection and Recovery Using Hardware Transactional Memory. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*. 220–225.

[39] Richard M. Yoo, Christopher J. Hughes, Konrad Lai, and Ravi Rajwar. 2013. Performance Evaluation of Intel Transactional Synchronization Extensions for High-performance Computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*. 19:1–19:11.

[40] J.F. Zielger and H. Puchner. 2004. *SER–history, Trends and Challenges: A Guide for Designing with Memory ICs*. Cypress Inc.