FaulTM: Error Detection and Recovery Using Hardware Transactional Memory

Gulay Yalcin[1,2] Osman Unsal[1] Adrian Cristal[1]
[1]Barcelona Supercomputing Center
[2]Universitat Politecnica de Catalunya
Email: {gulay.yalcin, osman.unsal, adrian.cristal }@bsc.es

***Abstract*—Reliability is an essential concern for processor designers due to increasing transient and permanent fault rates. Executing instruction streams redundantly in chip multi processors (CMP) provides high reliability since it can detect both transient and permanent faults. Additionally, it also minimizes the Silent Data Corruption rate. However, comparing the results of the instruction streams, checkpointing the entire system and recovering from the detected errors might lead to substantial performance degradation. In this study we propose FaulTM, an error detection and recovery schema utilizing Hardware Transactional Memory (HTM) in order to reduce these performance degradations. We show how a minimally modified HTM that features lazy conflict detection and lazy data versioning can provide low-cost reliability in addition to HTM's intended purpose of supporting optimistic concurrency. Compared with lockstepping, FaulTM reduces the performance degradation by 2.5X for SPEC2006 benchmark.**

I. Introduction

Reliability is a first-class design constraint for processor designers, especially in mission-critical domain (e.g. automotive cruise-control systems or financial applications), since it is foreseen that technology trends will increase the transient and permanent fault rates in future processors [3], [2]. To tolerate hardware faults, a reliable system requires the inclusion of two key capabilities: 1) error detection and 2) error recovery.

Error detection is the process of discovering that an error has occurred. To satisfy the strict reliability requirements of mission-critical systems, various redundancy-based error detection solutions have been proposed [7], [14], [20], [22], [21], [26]. In particular, lockstepping is a popular hardware based error detection scheme and is widely implemented in systems requiring high-reliability such as the IBM S/390 G5 [20] or the HP NonStop servers [26]. Lockstepping executes an instruction stream redundantly in two synchronized and lockstepped processors and checks if both produce identical results. However, the comparison of execution results in order to detect divergent execution causes synchronization and comparison overheads in execution time especially if the inter-processor communication channel has a limited bandwidth.

Error recovery is the process of restoring the system's integrity after the occurrence of an error. Global checkpointing, a well-known recovery scheme, recovers detected errors by rolling back all processors to an earlier validated state [16], [23]. However, it will not be scalable as we move towards many-core systems, due to two main reasons: First, global checkpointing schemes should implement synchronization mechanisms (either at checkpoint creation or checkpoint validation) to guarantee that all structures (e.g. cores) rollback to the same state in case of an error. Second, when an error is detected in one core, all the processors (faulty or not) have to roll back since errors could have propagated to the error-free cores through shared variables. Thus, assuming that error rate will be higher for higher core counts, it is foreseen that in future many-core processors, global checkpointing may take even more time than the execution of the application itself [15]. Moreover, many error detection proposals depend upon external error recovery mechanisms, thus they require additional integration effort.

In addition to the performance degradation in the error-free execution, reliability schemes require supplementary hardware and design of this extra hardware dedicated only for reliability (e.g buffers to save checkpoints). These structures increase system implementation and test complexity. Thus, most of the academic reliability proposals have not been implemented in real hardware. To the best of our knowledge, lockstepping is one of the few redundancy-based error detection proposals with a real implementation. Although software based reliability schemes have been proposed in order to avoid new hardware design, these schemes are not capable of detecting permanent faults. Also, they require recompilation of the source code.

In this study, we have **two main goals** in designing a highly reliable hardware: (1) to minimize the implementation complexity by utilizing existing hardware with minor changes, (2) to reduce the performance degradation. For these purposes, we introduce FaulTM[1], an error detection and recovery proposal based on Hardware Transactional Memory (HTM) which provides an ideal base for reliability. FaulTM does not propose a completely proprietary design, which would be complex to implement and test, but instead builds on HTM. In this study, we are motivated by the fact that HTM will soon be implemented in mainstream processors and available from large system integrators [5], [9], [17]. In this paper, we explain how a generic HTM could be minimally modified so that it can also support reliability in addition to HTM's intended purpose of supporting optimistic concurrency. To the best of our knowledge, this is the first design which utilizes HTM to provide a highly reliable system.

HTM[2] simplifies parallel programming by executing transactions atomically meaning that all the instructions in a

[1]The first draft of our idea has been appeared in a previous event [27].

[2]The reader is encouraged to refer to [11] and [10] for explanation of HTM fundamentals.

978-3-9815370-0-0/DATE13/©2013 EDAA

transaction either commit as a whole, or abort and roll back their changes. Transactions record their tentative reads and writes in a read-set and write-set respectively. FaulTM executes applications in two redundant threads (i.e it creates a backup thread) and in special-purpose reliable transactions (From now on, we will call these reliable transactions as *reliTX* in order to avoid any confusion with regular TM transactions). FaulTM classifies any mismatch between the write-sets and register files of reliTX pairs (i.e. original reliTX and backup reliTX) as a hardware error (transient or permanent), and aborts both reliTXs, which are then restarted. In the case of a complete match, original reliTX commits the changes to the shared memory. Note that FaulTM utilizes an HTM which detects conflicts at the end of transactions (lazy conflict detection) and commits the validated version of the data at the end of the transactions (lazy data versioning).

The major contributions of this paper are as follows:

- We propose FaulTM, a reliability scheme based on existing HTM implementations with appropriate modifications for reliability.
- We detail the FaulTM design which has low error detection and recovery overhead.
- FaulTM is a self-contained reliability proposal in which error detection and recovery is integrated without requiring any external mechanism.

We integrated the FaulTM design in a substantially modified version of the M5 simulator [4] modeling an HTM [8]. Compared with lockstepping, FaulTM reduces the performance degradation by 2.5X for SPEC2006 benchmark. With the inclusion of register file comparison, FaulTM can recover 100% of the injected faults in our fault injection simulation.

II. Fundamentals of FaulTM

FaulTM tolerates transient and permanent faults. A transient fault is a bit flip due to radiation events or power supply noise. Permanent faults are irreversible physical changes in the semiconductor devices. Error is the manifestation of a fault. Fault is benign if it does not harm an application.

A. Basic Design of FaulTM

FaulTM provides high reliability based on 4 steps (Figure.1):

Creating reliTXs: At the beginning of the execution, FaulTM creates a backup thread which executes the identical instruction stream to the original thread. Then the original and backup threads are executed as two separate reliTXs.

Executing reliTXs: Original and backup reliTXs are executed atomically and in isolation. Each reliTX independently sends load request to shared memory or read-sets. In FaulTM, there are no conflicts between the original and the backup reliTXs, because the backup reliTX is only for validation of error-free execution and it does not modify shared memory.

Ending reliTXs: Since both threads have identical instructions in the absence of a fault, their read-/write-sets have to be identical as well. Original and backup reliTXs wait for each other (spin) to reach the commit stage. Then, the reliTX pair compare their write-sets and register files through the

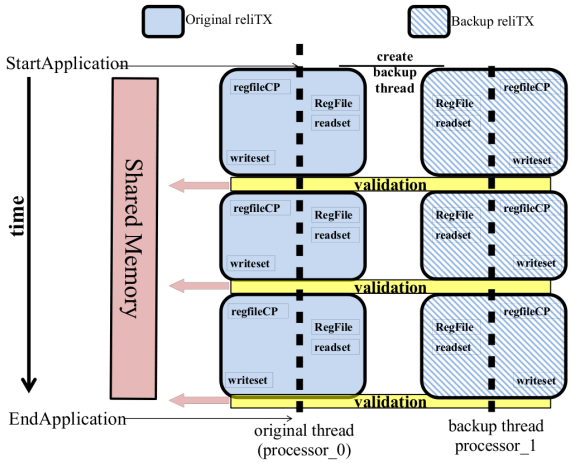


Fig. 1. Design of FaulTM for Sequential Codes

comparators in the backup reliTX. (From now on we call this operation as *validation*). If they match, the original reliTX commits its changes to memory, and the backup reliTX is cleared as if it aborts and it does not re-execute. Mismatch means an error due to a hardware fault in one of the reliTX that starts the recovery.

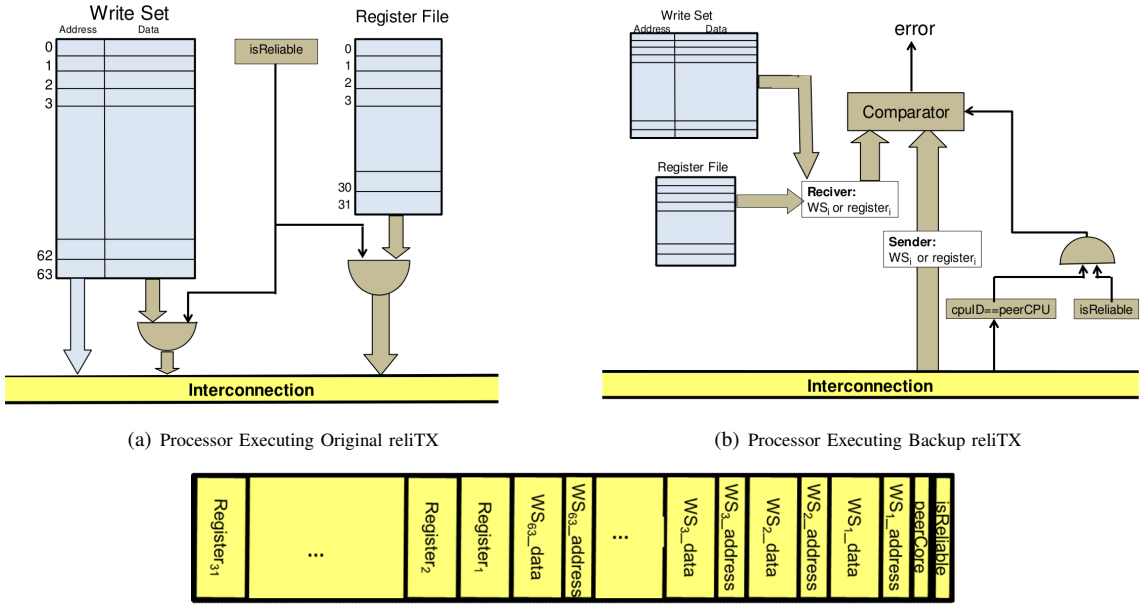
Error Recovery: If either write-sets or register files do not match at validation, both the original and backup reliTXs abort and they restart execution. If they match in the second execution, that means that there was a transient fault either in one of the cores or in the comparators in the first execution. Two successive mismatch signals between the same original and backup reliTXs signify that either one of the cores or the comparators has a permanent fault. FaulTM executes the reliTX in a third core to detect the source of the permanent fault by comparing the results with this third core's comparators as similar to TBFD [13].

B. Architecture of FaulTM

FaulTM extends a popular lazy-lazy HTM design [8] with minor modifications.

Splitting the reliTXs: In HTMs with lazy data versioning, local log buffers can emerge as a bottleneck for large transactions. Therefore, it is infeasible to execute the entire application in one reliTX. Fortunately, we can split reliTXs since they are for reliability not for parallelism. FaulTM starts validation processes when the write-set size reaches the transactional log size. Therefore, in run-time, the whole application is split into finer grained back-to-back reliTXs with constant-size log buffers. When the whole write-set is committed, a new reliTX starts by clearing the write-set. Splitting reliTXs allows us using a very small fully associative special transactional cache which holds the write-set and read-set of the whole reliTX.

Microarchitecture Extensions for FaulTM implementation: We need a simple controller having the list of processors in order to manage reliTXs. Besides coreIDs, we add several bits per core to this list such as *isReliable, isOriginal, peerCPU, wasFaulty and controlCore* bits, to account for mechanisms to deal with reliable execution. We may have additional TM applications running concurrently with our reliability-critical applications. Therefore, we use the `isReliable` bit to distinguish between the transactions and reliTXs. This bit is also sent



(a) Processor Executing Original reliTX

(b) Processor Executing Backup reliTX

(c) Data Sent to the Interconnect During Comparison

Fig. 2. The Validation Mechanism of FaulTM (Modified Structures are Shown in Grey)

together with the write-set at validation. If `isOriginal` bit is set, it indicates that the reliTX is the original one otherwise it is the backup reliTX. reliTX pairs have to be aware of each other to compare their results. The `peerCPU` bits point to the processor that has the peer reliTX of the reliTX that runs in the current processor. `wasFaulty` bit records if the core had a fault at the last time it executed a reliTX, hence it distinguishes if there might be a permanent fault. The `wasFaulty` bit is reset every time the core executes an error-free reliTX. When FaulTM detects two consecutive mismatches, it allocates a new core to recover from a permanent fault. The `controlCore` bit is set in the allocated core.

Adapting Conflict Detection for Error Detection In Figure 2, we demonstrate the paired processors running original and backup reliTX as well as the interconnect during error detection. Lazy-lazy TM systems compare the store addresses in both read-sets and write-sets for conflict detection. FaulTM compares store addresses as well as store data values in write-sets to detect if an error has manifested itself either in the address or in the data. Only reliTXs that set the `isReliable` bit send store data to their peer reliTX. Therefore, normal parallelism purposed transactions are not affected by this overhead. Note that reliability purposed comparison can be performed as in-order buffer comparison without requiring the associative address search of HTM.

Watchdog Mechanism: Assuming that an error in one reliTX leads to an incorrect execution path (i.e. an infinite loop), the reliTX may not reach the write-set limit or end of the application. FaulTM employs a watchdog mechanism which records the time since the last reliTX validation in the processor in order to enforce the reliTXs to validate and commit if the watchdog threshold has been exceeded.

Handling Input/Output Operations: In reliable systems, only error-free data can be communicated outside of the sphere which is called output commit problem. For example, the system can not print unvalidated data to the user screen. Also, input commit presents problem in reliability since input messages should be replayed after recovery. In FaulTM, we adopt the practical solution of both TM and checkpointing in which output values are deferred until validation (end of reliTXs in our case), and input values are logged to replay after recovery. Note that the size of the reliTXs are small enough for output delay.

Fatal Trap Exceptions: If one reliTX raises a hardware exception while its pair reliTX comes to commit, the recovery process starts and both reliTXs abort and restart execution.

C. Benefits of FaulTM

FaulTM presents five main benefits.

Less Comparison Overhead FaulTM reduces the comparison overhead compared to the previous redundancy-based fault-detection schemes [14], [7] due to two reasons. First, it compares the write-sets (instead of each store values) which have a fewer amount of entries than the total number of store instructions due to multiple stores to the same address. Second, register file comparison is done only at the commit stage of reliTXs (instead of after each instruction). Furthermore, comparison only at the commit point reduces the probability of detecting benign faults; because if a fault is masked within the reliTX, its effect is eliminated before the end of the reliTX. The comparison overhead of FaulTM can be reduced further by comparing hash-based signatures of write-sets and register files of reliTX pairs (We call this mechanism as FaulTM-sig in order to separate it from the base design). This mechanism is similar to Fingerprinting [21] in which a cumulative signature of the execution trace is generated after every instruction instead of after every reliTX.

Lightweight Checkpointing with TM FaulTM recovers from errors using the abort mechanism of TM which rolls back to the beginning of the reliTX in the erroneous core. We argue

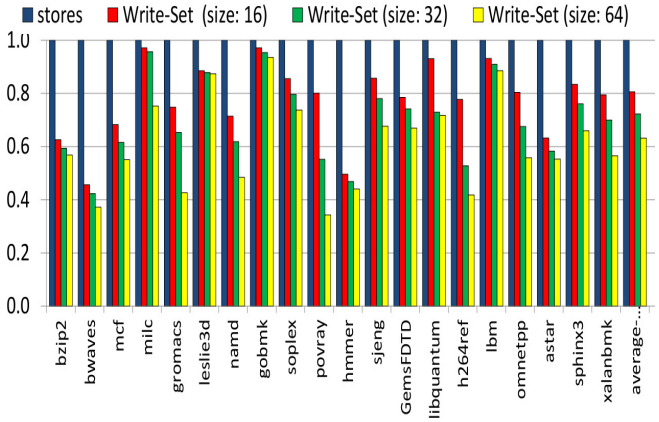


Fig. 3. Total entries in write-sets normalized according to the total number of store instructions in FaulTM.

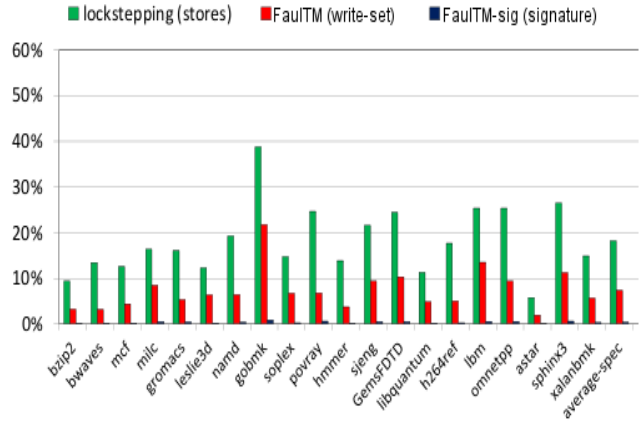


Fig. 4. The Error Detection Overhead of FaulTM vs Lockstepping in the Execution Time

that this is a simpler method than system-wide checkpointing [16], [23] that requires complex synchronization mechanisms to guarantee that all structures rollback to the same state. Moreover, system-wide checkpointing requires long recovery time due to its long checkpoint interval. FaulTM flushes the local log area and loads back the checkpointed register file for recovery only in the faulty core which has a negligible overhead. FaulTM does not require extra hardware resources, while reliTX sizes are very short compared to system-wide checkpointing.

Full-state Comparison Redundancy based reliability methods detect errors by comparing either only the results of store instructions [14], [18] or results of all instructions [7], [21], [24] by assuming that a fault is benign unless it propagates the architectural state. However, only full-state comparison (e.g. checking the register file) guarantees the error-free operation since the last validation. FaulTM provides full-state comparison at commit point with acceptable performance degradation.

Eliminating the requirement of separate input replication mechanisms Previous redundancy based methods [7], [14] require input replication mechanisms such as a load value queue, because a value can be changed by another thread in the system between the time it is read by the first thread and it is read by the second thread. In FaulTM, the conflict detection mechanism of TM guarantees that there would not be any modifications in the loaded values by other threads.

D. Overheads and Limitations

Core and Energy Overhead During the execution of a sequential application in a multi-core architecture, only one core is occupied and the others stay idle. FaulTM leverages one of the idle cores for the reliability purpose which supplies the capability of detecting both transient and permanent errors with 100% core overhead as previous redundancy based reliability schemes on CMPs [14], [7], [22], [21].

ReliTX Creation Overhead In a lazy-lazy HTM, creating a transaction means starting to write the values to the local log area instead of writing to the shared memory. Therefore, FaulTM does not have a reliTX creation overhead. However, the backup reliTX is obliged to copy the register file and TLBs from the original thread to be able to produce the same results. Fortunately, this copy operation does not need to be done when the transactions are back-to-back.

Spinning Overhead Whenever a reliTX reaches the end point, it spins, waiting for its pair to reach the same point. This overhead is higher in the previous schemes since they spin at every store instructions.

Comparison Overhead Validation of the execution is done by comparing the write-sets and register files of paired reliTXs by the core responsible for the backup reliTX.

Limitations and Potential Solutions First, in FaulTM, out-of-order memory operations may cause false positives. Although we detail FaulTM for in-order executions, a straightforward solution for out-of-order cores is storing only retired instructions to writesets. Second, there are several challenges in redundantly executing multithreaded applications (lock-based or transactional) such as handling instructions dedicated to maintaining synchronization (e.g. locks, barriers or create/join threads). Thus, they require a more detail design. We dedicate this first paper only for explaining the fundamentals of FaulTM in sequential applications. We leave the multi-threaded applications for the future work Third, the implementation in this paper does not support non-deterministic applications.

III. Evaluation

A. Simulation Setup

We use the M5 full-system simulator [4] with an implementation of a HTM system using lazy data versioning and lazy conflict detection [8]. We evaluate FaulTM using spec cpu2006 [12] benchmark suite with test data-set by executing either 2 billion instructions or until application termination. We evaluate FaulTM in the context of a CMP with 4 in-order Alpha 21264 cores [1] running at 1GHz with private L1D and L1I caches and a unified L2 cache. Each L1 cache is 64KB with four-way set associativity, and a two-cycle hit latency. The L2 cache is 2MB with eight-way set associativity, and 10 cycles of hit latency. All caches are write-back with a line size of 64B. Main memory latency is 100 cycles. We use fault injection to measure the reliability performance of FaulTM. We inject the faults to five different structures in a core; instruction opcodes, program counter, integer register file, special purposed register file and arithmetic logic unit. Note that, in-order cores do not have some complex structures required for out-of-order execution such as reorder buffer or issue queue. We inject 100 faults per structure in each application to a random location in each structure at a random time after warming up

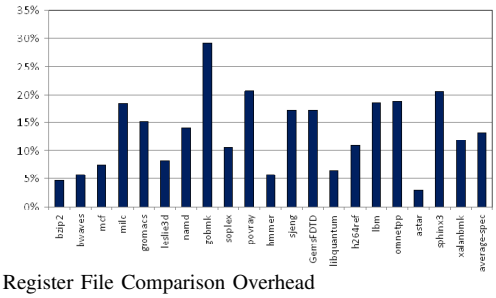


Fig. 5. Register File Comparison Overhead

200M instructions by performing one injection per simulation. While we flip the chosen bit for transient fault injection, we use stuck-at-0 and stuck-at-1 models for permanent faults. We simulate 10M cycles after fault injection to observe the effect of the injected fault.

B. Evaluation of FaulTM

We compare FaulTM against lockstepping in which after every store instruction, two threads synchronize and the results of the store are compared. Note that, validation of store values is common in several recent redundancy-based error detection techniques [7], [14], [18]. FaulTM compares the write-sets which have fewer amount of entries than total number of store instructions. In Figure 3 we present the normalized value of the total amount of entries in write-sets according to total number of store instructions in each application on different write-set sizes: 16, 32 and 64 entries. We find that, on average, total number of entries is 35% less than total number of store instructions when write-set size is 64.

In Figure 4, we compare the performance overhead of FaulTM (WS with 64 entries) with lockstepping including comparison and spin overheads. We assume that each comparison (e.g. comparison of a store value or an entry in either WS) can be accomplished in one cycle on an idealized bus where collisions are not modeled. This favors lockstepping since lockstepping is penalized more when the latency of the bus is higher. Spin overhead is, on average, 1 cycle for lockstepping and 4 cycles for FaulTM according to our simulation results. Compared with lockstepping, FaulTM reduces the performance degradation by 2.5X for SPEC2006 benchmarks. FaulTM-sig shows the performance degradation of FaulTM when hash-based signatures are compared instead of the entire write-sets. Note that, error coverage of signature comparison is not 100%. In Figure 5, we present the comparison overhead of the entire register file in FaulTM (13%) in order to guarantee the error-free execution since the last validation. Register value comparison after each instruction in lockstepping would cause extremely high overhead (not included in the evaluations here). Note that the register file can also be included in the signature value in FaulTM-sig.

Figure 6 presents the reliability performance of FaulTM. According to fault injection experiments, FaulTM provides 100% error coverage for both transient and permanent faults with the comparison of entire write-set and register file. In case of comparing signatures instead of comparing entire write-sets, 5 transient errors (among 155000 transient fault injection) are not detected in FaulTM-sig (not in the graph), thus, in extremely high reliability requiring systems, signature usage might not be appropriate. FaulTM, also, avoids detecting benign faults. We find that 4% of transient faults (not in the graph) are treated as error in Fingerprintig [21] while FaulTM does not try to recover these benign faults and it does not cause a false positive signal since they are masked before the end of reliTXs. `Watchdog` and `Exception` are explained in Section II-B.

For error recovery, reliable systems require additional checkpointing mechanism which presents checkpoint creation and recovery overhead. Comparatively, FaulTM has a negligible transaction creation and abort overhead. The average number of instructions in reliTXs are generally less than 10K instructions which is very low compared to system-wide checkpointing mechanisms (10M instructions) [16], [23] that reduces the re-execution overhead of the instructions from the beginning of the reliTX for the error recovery. Moreover, smaller checkpoint interval is essential to support I/O operations [21].

IV. Related Work

In the following sections, we cover previous work.

Error Detection: To protect processor logic from transient faults, some studies utilize Simultaneous Multi-threading by executing two identical threads in the same core and comparing their results [18], [19], [24]. However, they are not suitable to detect permanent faults. In recent work, researchers leveraged chip multiprocessing (CMP) for error detection by pairing cores for redundant execution and checking their results [7], [14], [20], [26]. In lockstepping [20], [26], a classical reliability technique that is used by systems integrators, two synchronized and lockstepped processors run two identical instruction streams by receiving the same input and, comparing the output of the processors for every store instruction. Lockstepping requires tightly coupled processor pairs driven by the same clock signal which becomes an increasing burden as device scaling continues. CRT [14] validates only the memory values assuming that a fault is benign unless it propagates to the memory. However, full-state comparison is essential to guarantee the error-free execution since the last validation. For this issue, CRTR [7] compares the results of register instructions together with memory values which makes the comparison overhead very high. Fingerprinting [21] strives to compare the result of all instructions with a very low comparison overhead by producing the signature of execution history. However, the fault coverage of Fingerprinting is not 100% while it treats benign faults and errors equally.

FaulTM provides full state comparison (register file and store values) with a low overhead. It also avoids the detection of benign faults, thus minimizes false positives. In FaulTM, detection and recovery is integrated, whereas in most other techniques, recovery requires external mechanisms.

In order to avoid the redundancy overhead, researchers propose symptom-based error detection schemes [13], [25], [28], which monitor error symptoms (e.g. fatal traps, miss-predictions) for error detection. However, their error coverage is limited which causes higher SDC rate [6]. Shoestring [6] partially replicates the instruction stream to reduce the SDC rate. FaulTM is orthogonal to Shoestring that it can reduce the comparison overhead of the replicated part if the replication is done in the hardware.

Error Recovery: ReVive [16] and SafetyNet [23] are well-known global checkpointing mechanisms that create system-

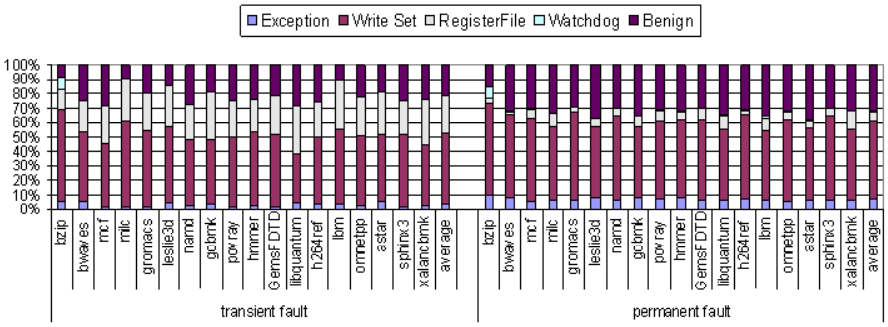


Fig. 6. Error Recovery Performance of FaulTM

wide checkpoints periodically. ReVive is only feasible in coarse granularity due to its large checkpoint size. However, I/O operations can only be supported in small checkpointing intervals [21]. Also, large checkpoints suffer from long recovery times. System-wide checkpointing schemes present several difficulties. First, they typically implement relatively complex synchronization mechanisms to guarantee that all structures (e.g. cores) rollback to the same state in case of an error. For instance, in SafetyNet, late synchronization causes several unvalidated checkpoints to be saved in the system which leads to an area overhead. Second, global checkpointing rollbacks all processors during recovery which causes a loss of error-free operations. FaulTM recovers from errors by leveraging the abort mechanism of lazy-lazy TM which keeps the invalidated data in the local log area of each core and writes only error-free data to shared memory. Therefore, FaulTM ensures that error does not propagate to shared memory and recovery is done internally in the core. Also, it eliminates the requirement of an additional synchronization mechanism to agree on a consistent recovery point.

V. Conclusions and Future Work

We introduce FaulTM, an error detection and recovery approach leveraging a lazy-lazy hardware transactional memory (HTM) system for both transient and permanent faults. FaulTM provides an efficient error recovery mechanism by utilizing the local checkpointing mechanism of TM. Also, it reduces the comparison overhead significantly by comparing the redundant execution streams at the end of the transactions instead of after every store instruction while avoiding error propagation to the whole system by utilizing the isolation property of transactions. Leveraging other HTM designs (i.e. HTMs with eager data versioning and/or eager conflict detection) for reliability presents other opportunities and challenges which will be tackled in future work.

References

[1] *Alpha 21264 Microprocessor Hardware Reference Manual*. Compaq Computer Corparation, 1999.

[2] R. Anglada and A. Rubio. An Approach to Crosstalk Effect Analysis and Avoidance Techniques in Digital CMOS VLSI circuits . *International Journal of Electronics*, 6(5):9–17, 1988.

[3] R. Baumann. Soft Errors in Advanced Computer Systems. *IEEE Design and Test of Computers*, 22(3):258–266, 2005.

[4] N. L. Binkert et al. The M5 Simulator: Modeling Networked Systems. *IEEE Micro*, 26:52–60, 2006.

[5] J. Chung et al. ASF: AMD64 Extension for Lock-Free Data Structures and Transactional Memory. *Micro*, 0:39–50, 2010.

[6] S. Feng et al. Shoestring: Probabilistic Soft Error Reliability on the Cheap. In *Proceeding of ASPLOS*, pages 385–396, 2010.

[7] M. Gomaa et al. Transient-fault Recovery for Chip Multiprocessors. In *Proceedings of ISCA*, pages 98–109, 2003.

[8] L. Hammond et al. Transactional Memory Coherence and Consistency. In *Proceedings of ISCA*, pages 102–113, 2004.

[9] R. Haring et al. The IBM Blue Gene/Q Compute Chip . In *IEEE Micro*, pages 1–1, 2011.

[10] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory*.

[11] T. Harris et al. Transactional memory: An overview. *IEEE Micro*, 27(3):8–29, 2007.

[12] J. L. Henning. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Computer Architecture News*, 34:1–17, 2006.

[13] M. Li et al. Understanding the Propagation of Hard Errors to Software and Implications for Resilient System Design. In *Proceedings of ASPLOS*, 2008.

[14] S. S. Mukherjee et al. Detailed Design and Evaluation of Redundant Multithreading Alternatives. In *Proceedings of ISCA*, pages 99–110, 2002.

[15] R. A. Oldfield et al. Modeling the Impact of Checkpoints on Next-Generation Systems. In *Proceedings of MSST*, pages 30–43, 2007.

[16] M. Prvulovic et al. ReVive: Cost-Effective Architectural Support for Rollback Recovery in Shared-Memory Multiprocessors. In *Proceedings of ISCA*, pages 111–122, 2002.

[17] J. Reinders. Transactional Synchronization in Haswell, February 2012.

[18] S. K. Reinhardt et al. Transient Fault Detection via Simultaneous Multithreading. *SIGARCH Computer Architecture News*, 28(2):25–36, 2000.

[19] E. Rotenberg. AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors. In *Proceedings of FTCS*, page 84, 1999.

[20] T. J. Slegel et al. IBM's S/390 G5 Microprocessor Design. *IEEE Micro*, 19:12–23, 1999.

[21] J. C. Smolens et al. Fingerprinting: Bounding Soft-error Detection Latency and Bandwidth. In *ASPLOS*, pages 224–234, 2004.

[22] J. C. Smolens et al. Reunion: Complexity-effective Multicore Redundancy. In *Proceedings of MICRO*, pages 223–234, 2006.

[23] D. J. Sorin et al. SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery. In *Proceedings of ISCA*, pages 123–134, 2002.

[24] T. N. Vijaykumar et al. Transient-Fault Recovery Using Simultaneous Multithreading. In *Proceedings of ISCA*, pages 87–98, 2002.

[25] N. J. Wang et al. ReStore: Symptom-Based Soft Error Detection in Microprocessors. *IEEE TDSC*, 3:188–201, 2006.

[26] A. Wood et al. Data integrity in HP NonStop servers. In *Workshop on SELSE*, 2006.

[27] G. Yalcin et al. FaulTM: Fault-Tolerance Using Hardware Transactional Memory. In *Workshop on PESPMA*, 2010.

[28] G. Yalcin et al. SymptomTM: Symptom Based Error Detection and Recovery Leveraging Hardware Transctional Memory. In *Proceedings of PACT*, 2011.