Fault Tolerance for Multi-Threaded Applications by Leveraging Hardware Transactional Memory

Gulay Yalcin[†, §] Osman Sabri Unsal[†] Adrian Cristal[†]
[†] Barcelona Supercomputing Center, Spain
[§] Universitat Politecnica de Catalunya, Spain
gulay.yalcin@bsc.es, osman.unsal@bsc.es, adrian.cristal@bsc.es

ABSTRACT

Providing fault tolerance especially to mission critical applications in order to detect transient and permanent faults and to recover from them is one of the main necessity for processor designers. However, fault tolerance for multi-threaded applications presents high performance degradations due to comparing the results of the instruction streams, checkpointing the entire system and recovering from the detected errors to an agreed state. In this study, we present FaulTM-multi, a fault tolerance scheme for multi threaded applications running on transactional memory hardware which reduces these performance degradations. FaulTM-multi decreases the performance degradation of lockstepping, a conventional fault detection scheme, from 23% and 9% to 10% and 2% for lock-based parallel and TM applications respectively. Also, FaulTM-multi creates 28% less checkpoints compared to Rebound, the state of the art checkpointing scheme.

Categories and Subject Descriptors

B.8.1 [**Performance and Reliability**]: Reliability, Testing, and Fault-Tolerance; C.4 [**Performance of Systems**]: Fault tolerance

General Terms

Reliability

Keywords

HTM, Redundancy, Error Detection, Error Recovery

1. INTRODUCTION

Fault tolerance is a critical aspect of any computer system since chips in the field are expected to see increasing failure rates due to permanent and transient faults [4, 3]. Any fault tolerant scheme should include two key mechanisms; 1) error detection and 2) error recovery mechanisms. However, implementing these two mechanisms presents difficult challenges especially for parallel applications.

Many redundancy-based error detection schemes, which execute instruction streams redundantly and check if both produce identical results, have been proposed in order to satisfy the strict reliability requirements of mission-critical systems [28, 21, 12, 23, 30]. These schemes provide the highest reliability since they can detect both permanent faults and transient faults while minimizing the rate of Silent Data Corruptions (SDC). However, in these schemes, besides the synchronization and comparison overheads in the execution time, maintaining identical instruction streams to the redundant executions (called the input replication problem) is a challenge for parallel applications.

Besides error detection, error recovery is also an indispensable aspect of fault tolerance in which the detected errors are recovered by restoring the system's integrity to an earlier state (i.e. checkpointed state). However, as we move towards many-core systems, it is not straightforward to implement recovery schemes due to two reasons. First, processors should synchronize at the checkpoint creation time meaning that they should wait for each other. Second, when an error is detected in one core, all the communicating processors have to roll back to the validated state since errors could have propagated to the error-free cores through shared variables between threads.

Programming shared-memory parallel architectures with traditional lock-based thread synchronization presents several difficulties such as deadlock, convoying, or priority inversion. These difficulties have led researchers to investigate alternative programming models. Transactional Memory (TM) is one of the popular parallel programming paradigm which simplifies parallel programming by executing transactions atomically and in isolation (In Section 2 we explain the aspects of TM). Obviously, by the time TM applications become widespread, they will require fault tolerance, as well. However, providing fault tolerance for TM applications by using conventional fault tolerance schemes is not feasible since it requires additional hardware and performance overhead over TM itself.

In this study, our goal is providing low-cost, architectural fault tolerance design for multi-threaded applications for both traditional lock-based parallel applications and TM applications. To this end, we propose FaulTM-multi, a fault tolerance scheme for multi-threaded applications using Hardware Transactional Memory (HTM). FaulTM-multi leverages the lightweight checkpointing mechanism of TM that provides local checkpointing for fault tolerance while it avoids error propagation out of the core. Also, FaulTM-multi eliminates the requirement of separate input replication mechanism.

In our previous studies [30, 29], we present FaulTM, an architectural error detection and recovery proposal for sequential applications using HTM. However, the design for multi-threaded applications is not presented in FaulTM. In this study, we extend FaulTM for multi-threaded applications (lock-based and TM) by addressing the explained challenges.

The main contributions of this study are:

- We present FaulTM-multi, a fault tolerance scheme for multi-threaded applications running on Transactional Memory Hardware.
- We detail the FaulTM-multi design in such a way that it addresses the input incoherence problem and provides a lightweight checkpointing scheme for multi-threaded applications.
- We elaborate FaulTM-muti design for TM applications in order to supply fault tolerance to them by using already existing structures in the TM hardware.

According to our evaluation, FaulTM-multi reduces the fault detection overhead from 23% to 10% for lock-based parallel applications and from 9% to 2% for TM applications by creating 28% less checkpoints for lock-based applications. Note that, since FaulTM-multi utilizes already existing schemes on TM, it does not present a checkpoint creation overhead for TM applications.

In Section 2, we present the background of TM and FaulTM. In Section 3, we detail the FaulTM-multi for parallel applications. In Section 4, we extend the FaulTM-multi for TM applications. In Section 6 we evaluate our proposal and we conclude in Section 8.

2. BACKGROUND

In a computer system, a fault can be either transient or permanent. A transient fault (also known as Soft Error) is a bit flip due to some radiation event or power supply noise. Since the data bit stored in a device is corrupted until new data is written to that device, these errors are temporal (transient). Irreversible physical changes in the semiconductor devices are called permanent faults. Error is the manifestation of a fault. In this section, we present the background for Transactional Memory and FaulTM.

2.1 Transactional Memory

Transactional Memory (TM) is a promising technique which aims to simplify parallel programming by executing transactions (sequences of instructions) atomically and in isolation. Atomicity means that all the instructions in a transaction either commit as a whole, or abort and roll back their changes. When a transaction commits, its tentative updates are made permanent. Transactions record their tentative reads and writes in a read-set and write-set respectively. All TM proposals implement three key mechanisms: data versioning, conflict detection and conflict resolution. Data versioning manages all the writes inside transactions until transactions successfully commit or abort. Conflict detection tracks addresses of transactional reads and writes to identify concurrent accesses that violate consistency. Conflict resolution aborts one or more transactions to resolve conflicts. In this study, we use lazy conflict detection/resolution and lazy data versioning (lazy-lazy) HTM, meaning that conflict detection is done only at the end of the transactions and the validated version of the data is written (published) to the shared memory at the end of the transaction.

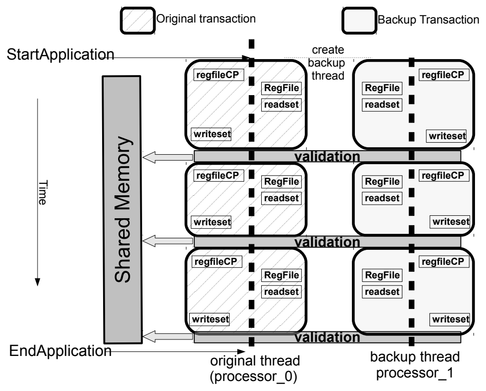


Figure 1: Architectural view of FaulTM. FaulTM executes applications in two redundant threads and in two *rel-tx* (i.e. original and backup). It classifies any mismatch between rel-tx pairs as an error and aborts both rel-txs. In the case of a complete match, one of the rel-txs commits the changes to the shared memory.

In this study, we are motivated by the fact that HTM will soon be implemented in mainstream processors. Although the first processor utilizing HTM, the Rock from SUN, was cancelled; the 48-core Vega2 chip from Azul systems which uses HTM [9] to accelerate Java is widely used. Recently, AMD detailed their Advanced Synchronization Facility (ASF) HTM proposal for X86 [8], IBM announced that BlueGene/Q, IBM's next generation supercomputer supports lazy-lazy TM [7] and Intel released details of Intel Transactional Synchronization Extensions (TSX) for Haswell, Intel's future multicore processor [17].

2.2 FaulTM

In our previous studies [30, 29], we present FaulTM, an architectural error detection and recovery proposal for sequential applications using Hardware Transactional Memory (HTM). In this study, we extend FaulTM design for multi-threaded applications.

Two key HTM characteristics are notably suitable for fault detection and recovery. First, HTM systems already have well-defined comparison mechanisms of read-/write-sets in order to detect if there is any conflict between transactions. While comparison of addresses is sufficient for conflict detection, some systems also send data along with addresses. FaulTM adapts these already existing conflict detection mechanisms for error detection. Second, HTM systems provide mechanisms to abort transactions in case of a conflict, thus they discard or undo all the tentative memory updates and restart the execution from the beginning of the transaction. Thus, a transaction's start can be viewed as a locally checkpointed stable state. FaulTM uses the already existing TM abort mechanism for error recovery.

In the FaulTM approach [30, 29] as it is presented in Fig-

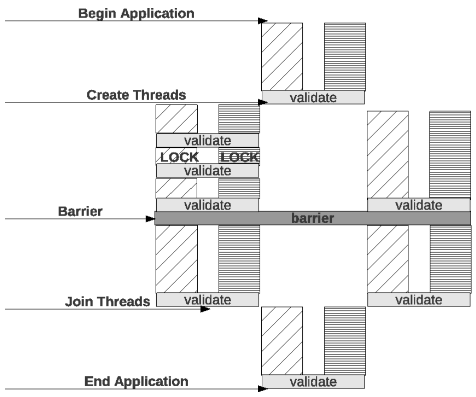


Figure 2: FaulTM-multi for traditional, lock-based parallel applications

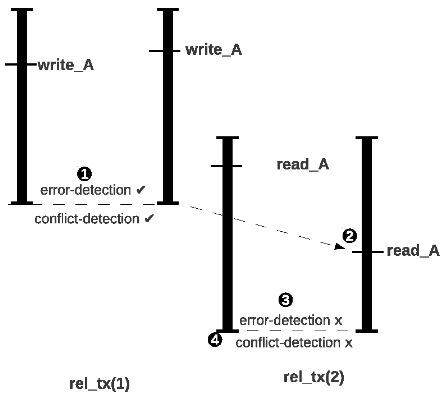


Figure 3: Addressing input replication in FaulTM-multi

ure 1, it 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 special-purpose reliable transactions as *rel-tx* in order to avoid any confusion with regular TM transactions. The FaulTM approach classifies any mismatch between the write-sets and register files of rel-tx pairs as a hardware error (transient or permanent), and aborts both rel-txs which are then restarted. In the case of a complete match, one of the rel-txs commits the changes to the shared memory and the other aborts and does not restart. The comparison of write-sets and register files of the original and backup rel-tx is called as *validation* since it is for validating that the execution of the transaction was error free. Since data in cache and memory structures are protected by ECC which is generated during the execution of store instructions, "*reading/writing from/to memory*" and "*conflict resolution*" are not vulnerable to hardware faults in FaulTM.

In fault tolerant systems, only validated, fault-free data can be communicated outside of the sphere which is called output commit problem. For instance, only validated data should be printed on the user screen. Also, since input messages should be replayed after recovery, input commit presents problem in fault tolerance as well. FaulTM [29] adopts the standard solution in which output values are delayed until validation (end of rel-txs in FaulTM case), and input values are logged to replay after recovery. Note that the size of the rel-txs are small enough for output delay.

3. NON-TM PARALLEL APPLICATIONS

There are two main challenges in redundantly executing multi-threaded applications: (1) handling instructions dedicated to maintaining synchronization such as locks, barriers or create/join threads and (2) maintaining identical instruction streams of redundant threads.

3.1 Synchronization Instructions

In FaulTM-multi, until rel-tx commits its write-set to the shared memory, the rest of the system can not be aware of the instructions in rel-tx. However, the thread management (e.g. create/join thread), coherency (allocate/release lock) and synchronization (e.g. barriers) instructions should be committed to the system to avoid deadlocks. Also, before the execution of these instructions, all the older instructions on the thread should be validated from the reliability point of view and committed. In this way, it is confirmed that the synchronization instruction is not in an erroneous path in the execution. When FaulTM-multi (Figure 2) encounters one of these special instructions (e.g. test-and-set, compare-and-swap) in a rel-tx, before executing the instruction, it first validates and commits the current rel-tx pair and starts a new rel-tx pair. As soon as the special synchronization instruction is executed, FaulTM-multi starts the commit process in order to make the system aware of the instruction. FaulTM-multi executes a special instruction in a rel-tx pair to keep it fault tolerant. Also, in this way, even if one of the rel-txs aborts due to a detected error after acquiring the lock, it will not try to reacquire it that it already held, which could have led to a deadlock. Note that lock release operations are accomplished by simple store instruction which can not be detected by the hardware so that it does not force rel-tx to commit. However, delaying lock release instructions until the end of rel-tx does not present any forward progress issues. Lock allocation/release is accomplished by only the original rel-tx since the backup rel-tx does not write anything to the shared memory. However, backup rel-tx is allowed to operate on the data locked by its original pair.

In Figure 2 we demonstrate how FaulTM-multi handles special instructions dedicated to maintain synchronization. In the figure, the application starts with a single thread and FaulTM-multi creates a backup thread in order to make it fault tolerant. Also these two threads are executed in rel-tx pairs. When the application attempts to create a thread to execute some parallel operations, FaulTM-multi forces the rel-tx pair in the system to start validate/commit process. After the thread creation, there are 4 threads in the system (i.e. 2 originals, 2 backups) running in rel-txs. Later on, the first rel-tx pair attempts to acquire a lock. Before executing the lock acquire instruction, the rel-tx pair validates and the original rel-tx commits its changes to the shared memory. After that, they execute the lock acquire operation reliably and only the original rel-tx issues the instruction to shared memory. After the lock is acquired, original and backup rel-tx pair operates on the data and writes the changes to their write-sets. After some point, when the write-set is full according to the FaulTM design, the rel-tx pair compares their write-sets for validation and original rel-tx commits its write-set to the shared memory. Since backup rel-tx does not commit any data to shared memory, it does not present any correctness issue that it operates on a locked data by its original rel-tx. Similar to the thread creation and lock acquiring, rel-tx pairs are also forced to validate/commit at barriers and thread join instructions.

3.2 Input Incoherence

The second challenge for multi-threaded applications is maintaining identical instruction streams to replicated threads called input incoherence problem. Input incoherence occurs in FaulTM-multi when a value is changed by another thread in the system between the time it is read by the first rel-tx and it is re-read by the second rel-tx. In Figure 3, we demonstrate an input incoherence in FaulTM-multi and how the conflict detection mechanism of TM solves it. In the example, there are two original threads and two backup threads in the system which are executed in rel-tx pairs namely rel-tx(1) and rel-tx(2). The original thread in rel-tx(2) reads the value A before the rel-tx(1) pair commits the new value of A to the shared memory. After rel-tx(1) commits the new value of A, the backup thread in rel-tx(2) reads A. Thus, in rel-tx(2) the original thread reads the old version of A while the backup thread reads the new version written by rel-tx(1). rel-tx(2) detects an error since the original and the backup rel-txs operate on different values. Note that this case occurs in an application with a data race. However, the fault tolerance scheme treat this kind of race as an error and it calls the recovery scheme. If the race occurs repetitively in a fault tolerant architecture, the system treats the race as a permanent error and disables the hardware structure although it is not faulty.

In the example, although error-detection is adequate to abort and restart rel-tx potentially solving the input incoherency, conflicts should also be detected after error detection to distinguish input incoherence, transient faults and permanent faults. For instance, two consecutive input incoherences on the same processor is considered as a permanent fault if conflict detection is not done. Note that both the original and the backup rel-txs should accomplish conflict detection since the late reader can be either the original or the backup rel-tx.

4. TM APPLICATIONS

Many researchers have developed applications using TM with the purpose of benchmarking different implementations, and studying whether or not TM is easy to use [10, 13]. We believe that by the time the TM programming model is adopted by programmers and HTM systems are implemented, there will be many TM applications to be executed on HTMs. Providing reliability to TM applications on HTM by using the conventional fault tolerance schemes is infeasible since they require additional comparison and checkpointing over TM itself. In this study, we provide error detection and recovery for TM applications in lazy-lazy HTM by leveraging the existing structures on the hardware (Figure 4). We leave supporting reliability with other HTMs such as eager-lazy, eager-eager or hybrid-policy [11] systems as future work.

When a thread encounters the instruction that starts a transaction, FaulTM-multi first validates/commits the rel-tx before it starts the transaction. Then transaction starts both in the original and the backup processors and they are executed in rel-tx. Since transactions already write the produced values to their write-sets, rel-txs do not need to register the write values again. Rel-txs do not start the validation until the transactions commit. If the TM transaction commits, the reliability validation is carried out before the TM transaction publishes its write-set to avoid any error propagation out of the core. In case the TM transaction aborts due to a conflict, rel-tx is also aborted to avoid any correctness issues. After both transactions reach the commit stage, their write-sets and register files are compared before collision detection in order to abort erroneous transactions before they cause other transactions abort erroneously.

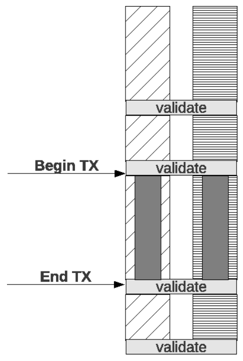


Figure 4: Design for TM Applications

There are two key implementation questions for FaulTM-multi for TM applications: 1)irrevocable operations in transactions and 2) nested transactions.

In TM, if there is an irrevocable operation in a transaction, TM marks this transaction as such and the transaction does not abort. There can be only one irrevocable transaction in the system. If a transaction conflicts with an irrevocable transaction, the conflicting revocable transaction aborts and the irrevocable one commits. When an irrevocable transaction is executed with rel-tx, rel-tx is validated before the irrevocable operation. rel-tx creates a checkpoint of the register file and the write-set in order to ensure that if an error is detected after the irrevocable operation in the irrevocable transaction, it can roll-back after the irrevocable operation.

In TM systems, there can be nested transactions that begin and end within the scope of surrounding transactions. There are two types of nested transactions: closed and open. In a closed-nested TM system using flattening, either all or none of the transactions in a nested region commit. In contrast, in an open nested TM, when an inner transaction commits, its effects become visible for all threads in the system. In FaulTM-multi, validations of the close-nested transactions are performed at the commit of outer most transaction while the validations of an open-nested transactions are performed when the nested transaction commits.

5. BENEFITS AND OVERHEADS OF FAULTM-MULTI

In this section, we explain the benefits and overheads of FaulTM-multi in detail.

5.1 Benefits of FaulTM-multi

FaulTM-multi presents five main benefits;

First, FaulTM-multi reduces the comparison overhead compared to the previous redundancy-based fault-detection schemes [15, 25, 12]. This is because, FaulTM-multi compares the write-sets (instead of each store values) which have a fewer

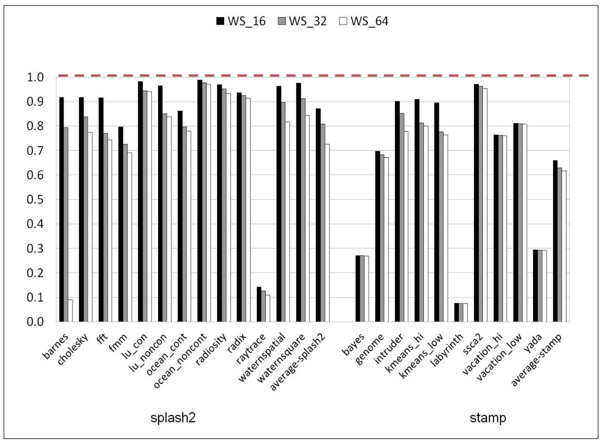


Figure 5: The figure shows the total entries in write-sets which are normalized according to the total number of store instructions.

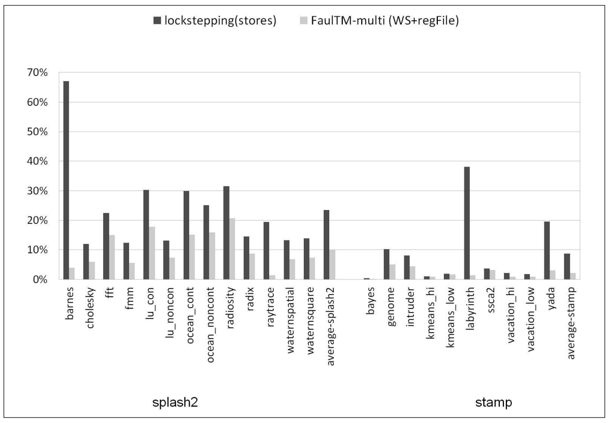


Figure 6: The figure compares the time overhead of FaulTM-multi and lockstepping.

amount of entries than the total number of store instructions due to multiple stores to the same address. Also it compares the register file only at the commit stage of rel-txs instead of after each instruction.

Second, FaulTM-multi presents a lightweight checkpointing scheme with TM hardware. FaulTM-multi recovers from errors using the abort mechanism of TM which rolls back to the beginning of the rel-tx in the erroneous core. In this way, FaulTM-multi does not require complex synchronization mechanisms to guarantee that all structures roll back to the same state. Also, after an error detection FaulTM-multi flushes the local log area and loads back the checkpointed register file for recovery only in the faulty core which has a negligible overhead. Moreover, FaulTM-multi does not require any additional hardware structures on top of transactional memory hardware to save checkpointed state.

Third, FaulTM-multi avoids the error propagation out of the core. FaulTM-multi uses a lazy data versioning HTM in which shared memory does not keep any invalidated data, therefore any error occurring in a certain core does not propagate to the other cores through memory. Thus, only the erroneous core rolls back while the rest of the system keeps running without wasting any error-free work done.

Fourth, FaulTM-multi eliminates the requirement of input replication, since the conflict detection mechanism of Transactional Memory guarantees that there would not be any modifications in the loaded values by other threads. In this sense, FaulTM-multi is similar to ReUnion [22] which does not require an extra hardware component for input replication. ReUnion recovers from the input incoherence by benefiting from the fact that if redundant threads read different values, they produce different results, otherwise the difference is benign. Therefore, it solves the input incoherence by utilizing an error detection mechanism. However, ReUnion is not convenient for permanent fault detection since it can not differentiate if two successive mismatch of results are due to a permanent fault or an input incoherence. FaulTM-multi, on the other hand, solves the input incoherence issue between redundant rel-txs by benefiting from the lazy conflict detection of HTM which identifies concurrent accesses by tracking the addresses in read-sets and write-sets.

Fifth, FaulTM-multi supports full-state comparison (i.e comparison of register file as well) at the validation points in order to guarantee the error-free execution since the last validation.

5.2 Overheads of FaulTM-multi

FaulTM-multi presents 100% core overhead to supply the capability of detecting both transient and permanent errors. This overhead is paid by all previous redundancy based fault tolerance schemes on CMPs [15, 25, 12, 22, 23] .

The second overhead is spinning overhead. Whenever a rel-tx 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.

Note that FaulTM-multi does not present a rel-tx creation overhead since it leverages lazy-lazy HTM in which creating a transaction means starting to write the values to the local buffer area instead of writing to the shared memory. Moreover, although the backup rel-tx is obliged to copy the register file and TLBs from the original thread to be able to produce the same results, this copy operation does not need to be done when the transactions are back-to-back.

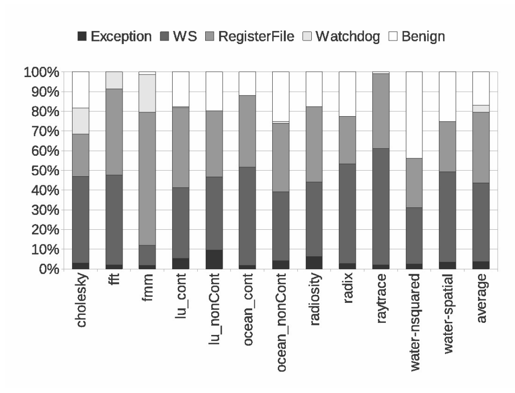
Finally, FaulTM-multi validates the execution by comparing the write-sets and register files of paired rel-txs by the core responsible for the backup transactions which presents a comparison overhead.

6. EVALUATION

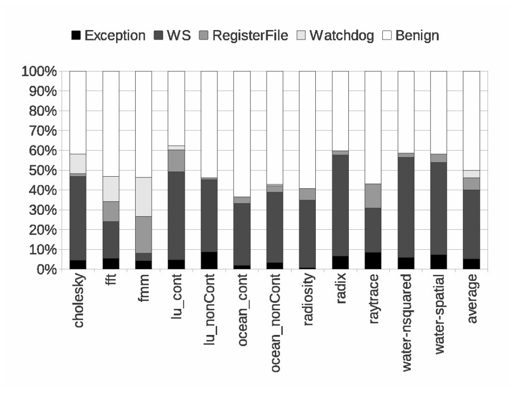
In this section, we evaluate the reliability performance of FaulTM-multi and its time overhead for fault detection and for fault recovery. First we explain the evaluation environment then we present our experimental results.

6.1 Evaluation Enviroment

For the evaluation of reliability performance and time overhead for error detection, we use the M5 full-system simulator [5] with an implementation of a HTM system that uses lazy data versioning and lazy conflict detection [20]. We evaluate our approach with 4 original and 4 backup threads using splash2 [27] and stamp [6] benchmark suites which are the representative of lock-based parallel and TM applica-



(a) Transient Faults



(b) Permanent Faults

Figure 7: The figure presents the error detection performance of FaulTM-multi for transient and permanent errors.

tions. We evaluate our technique in the context of a CMP with 8 in-order Alpha 21264 cores [1] running at 1GHz with private L1D and L1I caches and a unified L2 cache. Each level-1 cache is 64KB with four-way set associativity, 64B line size, and a two-cycle hit latency. The L2 cache is 2MB with eight-way set associativity, a line size of 64B, and 10 cycles of hit latency. All caches are write-back and main memory latency is 100 cycles.

To measure the reliability performance of FaulTM-multi, we use fault injection technique for both transient and permanent faults. We inject 100 faults to each hardware structure (i.e. register file, ITB, DTB, bypass logic, opcodes) for each applications in splash benchmark.

To measure the time overhead, we compare FaulTM-multi with lockstepping and Rebound. For the time overhead of error detection in the error-free execution, we compare FaulTM-multi system against lockstepping, a standard error detection method. In the lockstepping approach, after every store instruction, two threads synchronize and the results of the store are compared. For the time overhead of error recovery, we compare FaulTM-multi with Rebound [2], the state of the art checkpointing scheme. In Rebound, whenever a core checkpoints its execution, it writes the dirty lines to the shared memory (or main memory) which is protected by other means such as error correcting codes. This scheme is quite similar to FaulTM-multi in which write-sets are written to shared memory in every checkpoint interval (i.e. in every rel-tx). In Rebound, when core A checkpoints, all other cores which produce data used by core A checkpoint (i.e. all *producers* of core A checkpoint). Also, when core A needs to recover due to a detected error, all the cores which has consumed any data produced by core A also recovers (i.e. all *consumers* of core A recovers). For comparing FaulTM-multi with Rebound, we execute splash applications in 32 threads by using PIN [14], a dynamic binary instrumentation tool to instrument the execution.

6.2 Experimental Results

FaulTM-multi reduces store value comparison overhead since it compares the write-sets (instead of each store values) which have fewer amount of entries than total number of store instructions in transactions due to multiple stores to same addresses. In Figure 5, we present this reduction on different write-set sizes (i.e. 16, 32 and 64 entries which is determined during the design time of an HTM). In the figure, each bar represents the normalized value of the total amount of entries in write-sets as compared to total stores. The dashed lines in the graph shows the normalized value stores in applications. We find that, in our benchmarks, the percentages of entries in write-sets are smaller than the percentages of all stores (on average, 35% less among all benchmarks when WS=64 entries), because some store instructions in transactions write to the same addresses multiple times. For instance, in raytrace, labyrinth and barnes with WS_64, write-sets have around 90% less entries than stores. Due to the temporal locality of the stores, our FaulTM-multi technique requires fewer comparisons compared to lockstepping in order to check errors. It is obvious that when we increase the size of write-sets, the ratio of compared data reduces.

In Figure 6, we compare the performance overhead of error detection of FaulTM-multi in error-free case with lockstepping including comparison and spin overheads. Note that we did not include the error recovery overhead in the graph. We use 64 entries in the write-sets of FaulTM-multi. We find that, on average, the performance degradation of our approach in error detection is %10 for splash and %2 for stamp which are 56% and 75% less than lockstepping for splash2 and stamp applications respectively.

Figure 7 presents the reliability performance of FaulTM-multi. For parallel applications, the potential problem is shared variables that might propagate errors to other cores. We avoid this problem by performing the reliability verification just before publishing the write-set to shared memory. According to our fault injection experiments with a 10M-cycle tracking window (we wait 10M-cycle after injecting a fault), our FaulTM-multi design provides 100% error coverage for both transient and permanent faults. In the figure, we present the rate of the faults which is detected in the relevant mechanism. These mechanisms are detecting 1) a fatal trap exception in a rel-tx, 2) a mismatch between write-sets of rel-txs, 3) a mismatch between register files of rel-txs, 4) an error which cause a time-out in the watchdog mechanism. Also, FaulTM-multi avoids detecting benign faults which is 20% of injected transient faults and 32% of injected permanent faults.

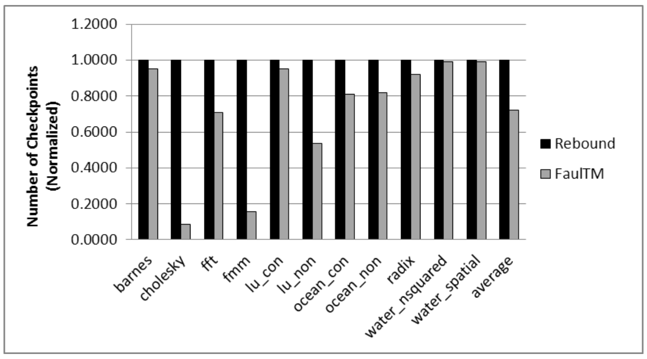


Figure 8: The figure compares the number of checkpoints created in FaulTM-multi and Rebound in the error free execution

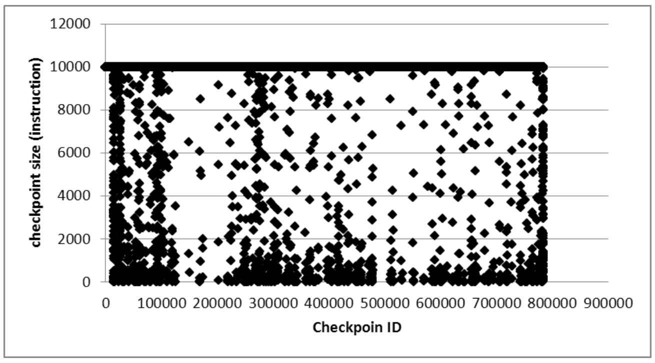


Figure 9: The figure presents the size of the checkpoints created by Rebound for cholesky. In Rebound, when a core checkpoints, its producers also checkpoint, thus, some checkpoints have less instructions than the checkpoint interval (i.e. 10M instructions).

The error recovery overhead of FaulTM-multi comes from two reasons. First, it needs to write the write-sets to the shared memory. Second, after an error is detected, it re-executes the instructions from the beginning of the rel-tx. Similarly, in Rebound, each core writes the dirty lines to the shared memory whenever they create a checkpoint at every 100M cycles.

In Figure 8, we compare the number of checkpoints created in FaulTM-multi and in Rebound. To be able to provide a fair comparison, we create checkpoints in Rebound and set the size of rel-tx according to number of instructions executed. Thus, FaulTM-multi sets the size of rel-txs as 10000 instructions and Rebound checkpoints a core when it executes 10000 instructions since the last checkpoint creation. Note that, in the normal case, FaulTM-multi checkpoints a core when the write-set is full. In Rebound, when a core checkpoints, all producers of the core also checkpoint thus some checkpoints include less than 10000 instructions. FaulTM, on the other side, presents a lightweight checkpointing scheme which reduces the number of checkpoints by 28% compared to Rebound. Note that, besides writing back the dirty lines, before and after each checkpoint creation Rebound has an additional overhead of synchronization.

In order to provide a close look, in Figure 9, we present the number of instructions executed in each checkpoint for cholesky (In cholesky, ∽800M checkpoints are created). As it can be seen from the figure, many checkpoints are created with the instruction numbers much lower than the normal checkpoint interval.

After the error is detected, FaulTM-multi re-executes the instructions only in **one** rel-tx. Rebound, on the other hand, recovers all the consumers of the erroneous core if there is any. Obviously, FaulTM-multi presents less recovery overhead than Rebound. In Figure 10, we present this reduction visually for one application, barnes. In the Figure, we present the number of cores which requires recovery for barnes application in case of an error is detected in the related checkpoint.

7. RELATED WORK IN RELIABILITY

Some fault-tolerance mechanisms just detect errors and alert the OS or the user (the so-called fail-stop systems) while other mechanisms both detect and recover from errors (the so-called fail-safe systems). In the next subsections, we cover previous work in error detection and recovery.

7.1 Error Detection

To protect processor logic from transient faults, some studies utilize Simultaneous Multi-threading (SMT) by executing two identical threads in the same core and comparing their results [18, 19, 26]. 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 [12, 15, 21, 28]. In lockstepping [21, 28], a classical fault-tolerance 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. Driving the clock signal across cores becomes an increasing burden as device scaling continues. Later studies avoid this burden by focusing on two main issues: (1) input replication and (2) output comparison

Input replication ensures that both processors observe identical load values. CRT [15] and CRTR [12] utilize load-value queue (LVQ) to provide an identical view of memory to the redundant executions. However, this strict input replication requires significant changes to the critical components of the processor core and cache hierarchy. Also, it forbids using the cache hierarchy for multi threaded executions. ReUnion [22] relaxes the input replication as it solves the input incoherence between redundant threads by utilizing the soft error detection mechanism. However, ReUnion is not convenient for permanent fault detection, since it can not differentiate if two successive mismatches between redundant executions are caused by input incoherence or a permanent fault. Note that, ReUnion utilizes efficient compression through Fingerprinting [23] for output comparison.

Output comparison checks the result of redundant executions if they are identical. CRT [15] validates only the mem-

	CRT [15]	**CRTR** [12]	**Fingerprinting** [23]	**ReUnion** [22]	**FaulTM-multi**
Transient Fault Detection Capability	very high	very high	very high	very high	very high
Permanent Fault Detection Capability	very high	very high	very high	high	very high
Fault Recovery Evaluation	any checkpointing	CRTR design	SafetyNet	ROB Rollback	TM abort
Avoiding Benign Faults	very high	high	low	low	high
Comparison Overhead	high	very high	very low	very low	low

Table 1: FaulTM-multi vs. Prior Fault Tolerance Schemes. (Higher is the better except Comparison Overhead in which case Lower is the better)

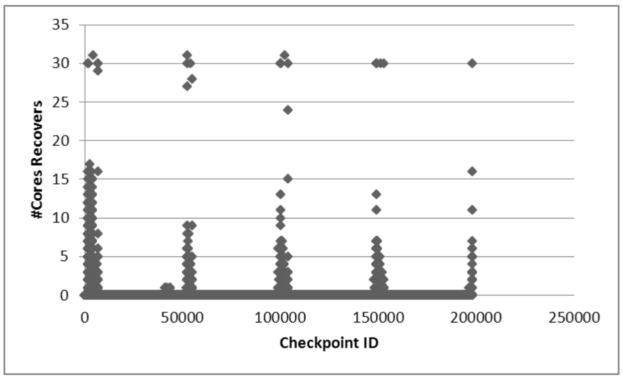


Figure 10: The figure presents the number of cores which require recovery in case of an error detection in barnes application. At some point, all cores(i.e. 32) need to be rolled back due to communication within checkpoint interval while FaulTM-multi rolls back only the erroneous core.

ory 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 [12] compares the results of register instructions as well. It reduces the bandwidth requirement by employing dependence-based checking elision which lets only the last instruction in the queue to use the register value queue. However, it still has a high comparison overhead. Fingerprinting [23] strives to compare the result of all instructions with a very low comparison overhead by producing the signature of execution history. However, it has two main drawbacks. Firstly, it is highly likely that some faults are not detected in the summary value. Secondly, it detects benign faults as well.

FaulTM-multi, our transient/permanent fault detection and recovery technique, provides full state comparison with a low overhead. It also avoids benign faults, thus minimizing false positives and relaxes input incoherence. In Table 1, we summarize the characteristics of FaulTM-multi compared to the prior studies explained above. As it is shown in the table, in FaulTM-multi, detection and recovery is integrated, whereas in most other techniques, recovery requires mechanisms that are external to the technique.

7.2 Error Recovery

ReVive [16] and SafetyNet [24] are well-known global checkpointing mechanisms that create system-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 [23]. Also, large checkpoints suffer from long recovery times.

System-wide checkpointing presents several difficulties. First, they implement relatively complex synchronization mechanisms to guarantee that all structures (e.g. cores) roll back to the same state in case of an error. Second, if erroneous data propagates other error-free cores before it is detected, it causes a loss of error-free operations in the other core before error propagation.

FaulTM-multi 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 validated /error-free data to shared memory. Therefore, FaulTM-multi ensures that error does not propagate to shared memory and recovery is done internally in the core. Also, it eliminates the requirements of an additional synchronization mechanism to agree on a consistent recovery point.

8. CONCLUSIONS

In this study, we introduce FaulTM-multi, an error detection and recovery approach leveraging a lazy-lazy hardware transactional memory (HTM) system for both transient and permanent faults on parallel (lock-based and TM) applications. FaulTM-multi 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. Moreover, it eliminates the requirement of a separate input replication mechanism by utilizing the conflict detection scheme of TM.

9. REFERENCES

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

[2] R. Agarwal, P. Garg, and J. Torrellas. Rebound: Scalable Checkpointing for Coherent Shared Memory. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, pages 153–164, 2011.

[3] 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.

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

[5] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The M5 Simulator: Modeling Networked Systems. *IEEE Micro*, 26:52–60, 2006.

[6] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[7] D. Chen, P. W. Coteus, N. A. Eisley, A. Gara, P. Heidleberger, R. M. Senger, V. Salapura, B. Steinmacher-burow, Y. Sugawara, and T. E. Takken. Embedding Global Barrier and Collective in a Torus Network, United States Patent Application, 12/723277.

[8] J. Chung, L. Yen, S. Diestelhorst, M. Pohlack, M. Hohmuth, D. Christie, and D. Grossman. ASF: AMD64 Extension for Lock-Free Data Structures and Transactional Memory. *IEEE/ACM International Symposium on Microarchitecture*, 0:39–50, 2010.

[9] C. Click. Azul's Experiences with Hardware Transactional Memory, January 2009.

[10] V. Gajinov, F. Zyulkyarov, O. S. Unsal, E. Ayguade, T. Harris, M. Valero, and A. Cristal. QuakeTM: Parallelizing a Complex Sequential Application Using Transactional Memory. In *Proceedings of the 23rd International Conference on Supercomputing*, pages 126–135, 2009.

[11] J. R. T. Gil, A. Negi, M. E. Acacio, J. M. García, and P. Stenström. ZEBRA: A Data-Centric, Hybrid-Policy Hardware Transactional Memory Design. In *Proceedings of the international conference on Supercomputing*, pages 53–62, 2011.

[12] R. Gong, K. Dai, and Z. Wang. Transient Fault Recovery on Chip Multiprocessor based on Dual Core Redundancy and Context Saving. *International Conference for Young Computer Scientists*, pages 148–153, 2008.

[13] G. Kestor, V. Karakostas, O. S. Unsal, A. Cristal, I. Hur, and M. Valero. RMS-TM: A Comprehensive Benchmark Suite for Transactional Memory Systems. In *Proceeding of the second joint WOSP/SIPEW international conference on Performance engineering*, pages 335–346, 2011.

[14] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, 2005.

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

[16] Prvulovic, Z. Zhang, and J. Torrellas. ReVive: Cost-Effective Architectural Support for Rollback Recovery in Shared-Memory Multiprocessors. In *Proceedings of the International Symposium on Computer Architecture*, pages 111–122, 2002.

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

[18] S. K. Reinhardt and S. S. Mukherjee. 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 the International Symposium on Fault-Tolerant Computing*, page 84, 1999.

[20] S. Sanyal, S. Roy, A. Cristal, O. S. Unsal, and M. Valero. Dynamically Filtering Thread-Local Variables in Lazy-Lazy Hardware Transactional Memory. In *Proceedings of the International Conference on High Performance Computing and Communications*, pages 171–179, 2009.

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

[22] J. C. Smolens, B. T. Gold, B. Falsafi, and J. C. Hoe. Reunion: Complexity-Effective Multicore Redundancy. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*, pages 223–234, 2006.

[23] J. C. Smolens, B. T. Gold, J. Kim, B. Falsafi, J. C. Hoe, and A. Nowatzyk. Fingerprinting: Bounding Soft-error Detection Latency and Bandwidth. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 224–234, 2004.

[24] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood. SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery. In *Proceedings of the International Symposium on Computer Architecture*, pages 123–134, 2002.

[25] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream Processors: Improving both Performance and Fault Tolerance. In *Proceeding of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 257–268, 2000.

[26] T. N. Vijaykumar, I. Pomeranz, and K. Cheng. Transient-fault recovery using simultaneous multithreading. In *Proceedings of the International Symposium on Computer Architecture*, pages 87–98, 2002.

[27] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. *SIGARCH Compututer Architecture News*, 23:24–36, May 1995.

[28] A. Wood, R. Jardine, and W. Bartlett. Data Integrity in HP NonStop Servers. In *Proceedings of the Workshop on System Effects of Logic Soft Errors*, 2006.

[29] G. Yalcin, O. Unsal, and A. Cristal. FaulTM: Error Detection and Recovery Using Hardware Transactional Memory. In *Design, Automation and Test in Europe*, 2013.

[30] G. Yalcin, O. Unsal, A. Cristal, I. Hur, and M. Valero. FaulTM: Fault-Tolerance Using Hardware Transactional Memory. In *Workshop on Parallel Execution of Sequential Programs on Multi-Core Architecture PESPMA*, 2010.