FaulTM: Fault-Tolerance Using Hardware Transactional Memory

Gulay Yalcin, Osman Unsal, Ibrahim Hur, Adrian Cristal, Mateo Valero

FaulTM: Fault-Tolerance Using Hardware Transactional Memory

Gulay Yalcin Osman Unsal Ibrahim Hur Adrian Cristal Mateo Valero
Barcelona Supercomputing Center
{gulay.yalcin, osman.unsal, ibrahim.hur, adrian.cristal, mateo.valero} @ bsc.es

Abstract

Fault-tolerance has become an essential concern for processor designers due to increasing soft-error rates. In this study, we are motivated by the fact that Transactional Memory (TM) hardware provides an ideal base upon which to build a fault-tolerant system. We show how it is possible to provide low-cost fault-tolerance for serial programs by using a minimally-modified Hardware Transactional Memory (HTM) that features lazy conflict detection, lazy data versioning. This scheme, called FaulTM, employs a hybrid hardware-software fault-tolerance technique. On the software side, FaulTM programming model is able to provide the flexibility for programmers to decide between performance and reliability. Our experimental results indicate that FaulTM produces relatively less performance overhead by reducing the number of comparisons and by leveraging already proposed TM hardware. We also conduct experiments which indicate that the baseline FaulTM design has a good error coverage. To the best of our knowledge, this is the first architectural fault-tolerance proposal using Hardware Transactional Memory.

1. Introduction

Soft errors, caused by alpha particles from package radioactive decay, neutrons or protons from cosmic rays or power supply noise, are becoming a major concern for processor designers. Soft errors do not result in permanent circuit faults. However, unless they are detected and corrected, soft errors may cause incorrect results, data corruption or system crashes. Although the frequency of the soft error occurrence in current processors is not high, due to increasing transistor integration densities and clock frequencies as well as decreasing supply voltages, it is expected that this type of errors will become more prevalent in future systems [19].

On the other hand, Transactional Memory (TM) is a promising technique which aims simplifying parallel programming by executing transactions atomically and in isolation. Atomicity means that all the instructions in a transaction either commit as a whole, or abort. All TM proposals implement two key mechanisms; data versioning and conflict detection. Data versioning manages all the writes inside transactions until the transactions successfully complete or abort. Conflict detection tracks the addresses of reads and writes in transactions to identify concurrent accesses that violate atomicity. In this study, we propose *FaulTM*, a soft-error detection and recovery technique based on Transactional Memory. To the best of our knowledge, this is the first architectural fault tolerance proposal using Hardware Transactional Memory (HTM).

Transactional Memory in general, and Hardware Transactional Memory in particular provide an ideal base platform for building a fault-tolerant system. 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 for conflict detection [8, 12, 25]. In these mechanisms the write sets of transactions are compared in order to detect if there is any conflict. While comparison of addresses is sufficient for conflict detection, some systems also send data along with addresses [8, 25]. FaulTM adapts these conflict detection mechanisms for fault detection. Second, HTM systems provide mechanisms to abort transactions in case of a conflict, thus they undo all the tentative memory updates and restart the execution from the start of the transaction which is a checkpointed stable state.

FaulTM uses the TM abort mechanisms for fault recovery.

In the FaulTM approach, we execute a vulnerable section of an application in two redundant threads and in two special-purpose reliable transactions. The FaulTM approach classifies a mismatch between the write sets of such reliable transaction pairs as a soft error, and aborts both transactions which are then restarted. In the case of a complete match, one of the transactions commits the changes to the main memory. This requires minimal hardware modifications to an existing HTM design. Therefore, FaulTM synergistically leverages already proposed TM hardware for reliability. In this sense, it is similar to other proposals that leverage HTM for other purposes such as speculative multithreading [18], scouting threads [27] or data race detection [7].

Most of the conventional fault tolerant systems validate results of all store instructions, because any error is benign unless it propagates out of the core. Since FaulTM only compares the write-sets which have fewer amount of entries than number of store instructions due to multiple stores to the same address, FaultTM has less comparison overhead than the previous systems.

Another advantage of FaulTM is supplying flexibility, if it is desired, to define vulnerable sections by using the TM-like programming model of FaulTM. Note that in this first study, we mark entire applications as vulnerable to soft errors.

This paper makes the following contributions:

- We introduce FaulTM to make multi-threaded systems fault-tolerant for sequential applications. To detect soft errors and to recover from them, this approach utilizes the conflict detection and the abort mechanisms of Hardware Transactional Memory systems. The FaulTM approach has three desirable properties: (1) it has low performance overhead in execution time, (2) it requires only minor hardware modifications, because it uses already proposed TM implementations, and (3) it provides the option for the programmer to determine code regions that are vulnerable to the soft errors.
- We evaluate our approach using the MediaBench Benchmark Suite. We find that FaulTM results in 3.2% overhead on average for sequential applications which is only 52.1% of the overhead of lock-stepping technique [23, 28].

	Data Versioning - Conflict Detection		
	lazy-lazy	lazy-eager	eager-eager
Error Containment	**High**	**High**	Low
Performance Degradation	**Low**	High	High
Error Detection Latency	High	**Low**	**Low**
Error Coverage	**High**	Low	Low

Table 1. Fault Tolerance attributes of different HTM implementations.

- Our experimental results indicate that on average FaulTM has good error coverage of 91.2%. A FaulTM extension which features 100% coverage with 2.8% additional performance degradation is also proposed.

2. Design of the FaulTM System

In this section, we explain the basic steps of our FaulTM approach, and we discuss its hardware and software requirements.

2.1 Determining the TM Design Parameters

There are two main design parameters for HTM systems that make sense for fault tolerance: data versioning and conflict detection policies. Each of these policies can be either lazy or eager. Out of four possible combinations of these policies, only the lazy-lazy [8], lazy-eager [26], and eager-eager [12] schemes generate valid results.

To select one of these combinations, we consider the following four desirable features for fault-tolerant systems: (1) high error containment, to limit the propagation of errors in the system, (2) high error coverage, to detect as many errors as possible, (3) low performance overhead, and (4) low error detection latency, to detect errors as soon as possible.

As we show in Table 1 (bolds are the desired properties), none of the possible three HTM policy combinations has all these features. In this study, we choose to use the lazy-lazy conflict detection and data versioning combination, because this combination has more desirable features than the other two options, and it has the lowest performance overhead.

Note that HTM policy combinations other than lazy-lazy could also be modified for fault-tolerance to quantitatively qualify their fault-tolerance attributes. However, these implementations are beyond the scope of this initial study.

In the rest of this section, we provide a succinct discussion of the impact of HTM policies on fault-tolerance features.

In HTM implementations with eager data versioning, main memory keeps the latest speculative version of the data. If we use eager data versioning for FaulTM, some data in the shared memory which is not validated, can be read by other cores. Assuming any of these data or any address is erroneous, this error might then propagate to other cores. Therefore, error propagation in eager data versioning is marked high in Table 1.

For performance degradation we need to analyze the number of comparisons required for fault tolerance. The higher the number of comparisons, the higher the potential for performance degradation. In lazy conflict detection only the final write sets of the transactions are compared. However, in eager conflict detection a comparison is necessary for every transactional store. Since some stores write to the same addresses multiple times, the number of entries in the write set could be significantly less than the number of stores in a transaction. Thus, the number of comparisons in lazy conflict detection would be less than in eager conflict detection. Therefore, we could conclude that potential performance degradation of lazy conflict detection is lower.

On the other hand, in lazy conflict detection any error occurring earlier in the transaction will only be detected at the commit stage, so error detection latency will be higher. In eager conflict detection, however, the error could be detected earlier when a transactional store containing the error is compared.

To provide full error coverage, we need to compare the register files along with the write sets. While lazy conflict detection would have reasonable overhead for making this comparison only on commits, for eager eager conflict detection, the overhead of comparing the register files at every transactional store would be unacceptably high. Therefore, the error coverage of lazy conflict detection is higher compared to eager conflict detection.

2.2 Basic Steps

The FaulTM approach consists of four steps: (1) defining the vulnerable sections of the code, (2) creating a *backup thread* for each vulnerable section, (3) executing both the original and the backup thread in transactions, and (4) comparing the write sets of the transactions at the commit stage. We now briefly explain these steps.

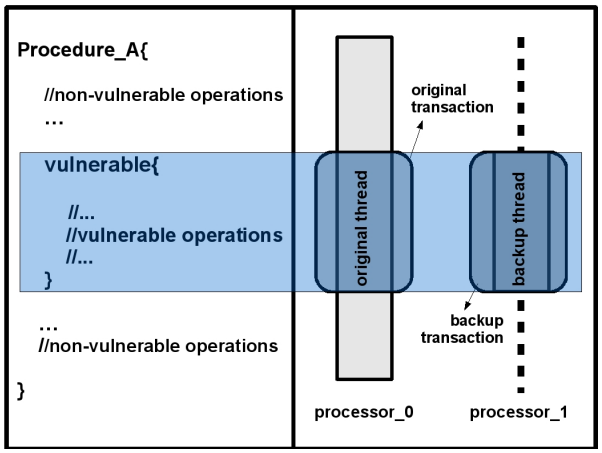

Figure 1. FaulTM design for sequential applications.

Defining vulnerable sections: In this study we assume that the entire application is vulnerable and needs to be protected. Note that it is also possible for the programmer to determine the reliability-critical sections of the code and to define only these sections as vulnerable for a tradeoff of performance for reliability. Figure 1 shows this case for a sequential application. Other alternatives for this step can be employing a compiler-driven approach to move the burden from the programmer to the compiler or using binary instrumentation to avoid recompilation of the application.

Creating transactions: As soon as the definition of a vulnerable section is reached, the FaulTM system creates a backup thread for that section, which is identical to the corresponding section of the original thread. Then the original and backup thread are executed as two separate transactions, that is, they execute atomically and in isolation from the other threads in the system.

Executing transactions: In lazy data versioning TM systems, such as TCC [8], all stores are written to a special buffer instead of the shared memory. In our approach, during the execution of the transactions, there will not be any conflicts between the original and the backup transactions, because the backup transaction is only for validation and it will not modify the shared memory.

Ending transactions: Both the original and backup transactions wait for each other to reach the commit stage. Since both threads have identical instructions, in

the absence of a fault, their read/write sets have to be identical as well. At the validation stage, the transactions compare addresses and data values of their write sets. If they match, the original transaction commits to the memory, and the backup transaction is cleared as it aborts and it does not execute again. If the write sets do not match, both the original and backup transactions abort and they restart execution.

2.3 Special Cases for Ending Transactions

Besides ending transactions at the end of the vulnerable section, we define three additional situations causing ending transactions: interrupts, system calls and buffer overflow.

Handling interrupts and executing system calls in our special transactions are not straight forward because of the problems associated with executing the same interrupt or system call twice, once in the original and once in the backup transaction. For example, let's suppose that there is a "printf" statement inside a vulnerable section that needs to be protected. We can not execute this command in both original and backup transactions, because otherwise we can not rollback the backup transaction and we will observe two outputs on the screen. Thus, we only protect user level operations assuming that the operating system is protected by other means. As soon as any system call or interrupt is detected in a reliable transaction, `Ending Transaction` stage starts in the processor to validate and commit the operations up to that point. Only one thread executes the system call or handles the interrupt. After returning from the operating system new original and backup transactions are started.

In HTM designs, an important consideration is the limited size of readset and writeset buffers. In FaulTM, we set a threshold for the buffer size and whenever the writeset of a transaction exceeds that threshold, the validation and the commit processes start. As soon as the whole write-set is committed, new original and backup transactions start with empty buffers.

If a soft error might appear to cause an infinite loop because of incorrect execution path in a transaction, eventually the readset/writeset buffer overflow forces the commit and the error check, therefore recovering from the error. Note that it is not necessary to have the identical reasons in transaction pairs for detecting the error.

2.4 Overheads of FaulTM

Core 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 reliability purpose which supplies the capability of detecting both soft and permanent errors. Although this design costs 100% of a core overhead, an SMT design of the TM is not adequate for detection of permanent errors. Moreover, since we use hardware threads, creating a backup thread is simply activating the suspended hardware thread in the allocated idle core which has negligible overhead.

Transaction Creation: In an HTM, creating a transaction means starting to write the values to the local buffer area instead of writing to the shared memory without thread creation overhead. However, in FaulTM, the backup transaction is obliged to copy the register file and TLBs from the original thread to be able to produce the same results. Note that this copy operation does not need to be done when the transactions are back-to-back. Even if any strike changes the value of any data on the bus, that would cause the final results of pair transactions to be different than each other.

Spinning Overhead: Whenever a transaction reaches to the end, it spins, waiting for its pair to reach to the same point.

Comparison Overhead: Comparison overhead of FaulTM is less than other fault tolerance methods which compare the results of all store instructions. This is because FaulTM compares only the entries in the writeset of transactions which are always less than the number of stores due to multiple writings to the same address in a transaction. Also, FaulTM needs less costly comparators than our base lazy-lazy TM [21] conflict detection under the assumption of changing the comparator design. Because, that TM needs to compare the address of each entry in a transaction with the addresses of all entries in the other transactions. However, it is enough for FaulTM to compare only the entries in the pair transactions' write-sets. Moreover, this comparison is only done between the entries in the same positions. For example, assume that there are T transactions in the system and each transaction has N entries in their write-sets. In the base TM system there should be (T-1)N*N comparison of address for detecting any conflicts. However, in FaulTM there would be only N

comparison of addresses and data for detecting any mismatch between pair transactions.

Committing Overhead: After validating the data, committing is done in the same way as in standard HTM by publishing the stores.

2.5 Programming Model and Software Extensions

FaulTM adds the keyword ‘‘`vulnerable`’’ to denote sections of code that should be protected against soft-errors. Using this keyword, programmers only need to define the vulnerable sections in their applications. They can insert vulnerability boundaries as if they define atomic sections in TM applications. The vulnerable sections can be either fine-grained, lasting for a few instructions or coarse-grained such as the entire application. While the fine-grained approach causes less performance degradation, coarse-grained approach provides more reliability. For instance, for an airplane control application, the programmer could identify that the code that is responsible for controlling the flaps should be protected coarsely, whereas the code regarding the on-flight entertainment system is not protected at all. Alternatively in the fine grained version of flap controlling code, the programmer decides to protect only the calculation of desired flap angle but he leaves the graphic user interface unprotected.

To be able to define vulnerable sections inside applications, we add two instructions to the Instruction Set Architecture, namely *begin_Reliable_Region* and *end_Reliable_Region*. Having these instructions provides flexibility, because reliability is provided only for the defined parts of the applications, and the other parts are executed without any overhead.

When a *begin_Reliable_Region* instruction is executed, the backup thread and the original/backup transactions are created; and when an *end_Reliable_Region* instruction is executed, the commit phases of the transactions start.

2.6 Architectural Extensions

The FaulTM approach extends an existing lazy-lazy design, such as TCC, with minor modifications, namely an *isReliable* bit, an *isOriginal* bit, and *peerCPU* bits, require only $(\lceil \log_2 n \rceil + 2)$ bits where n is the number of cores in the Chip Multiprocessor (CMP).

isReliable (1-bit): In the FaulTM system, we may have additional TM applications running concurrently with our reliability-critical applications. Therefore, we use the isReliable bit to distinguish between the transactions for parallelism and transactions for reliability. Unless the isReliable bit is set, the other two extensions, isOriginal and peerCPU, are not used.

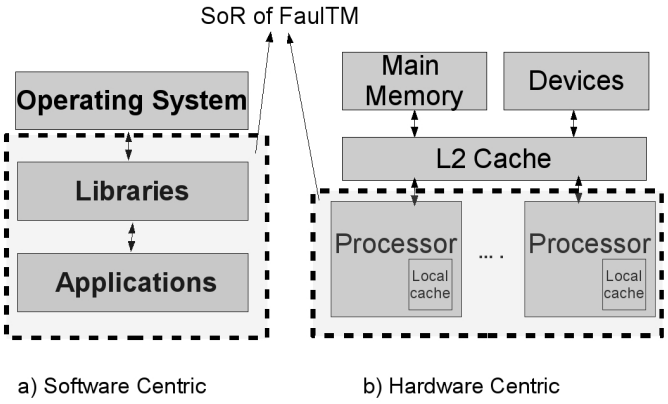

Figure 2. Sphere of Replication(SoR) of FaulTM in both software and hardware point of view.

isOriginal (1-bit): When a transaction is for reliability and if the isOriginal bit is set, this implies that this transaction is the original transaction, otherwise it is a backup.

peerCPU ($\lceil \log_2 n \rceil$)-bits for an n-core system): Transaction pairs have to know each other to compare their results. The peerCPU bits point to the processor that has the peer transaction of the transaction that runs in the current processor.

FaulTM compares store addresses as well as store data values to detect if a soft-error has manifested itself in either the address or data. First, the write set addresses are compared, a match signifies that there is no error in the store address. Once a match in the addresses is found, then the data are compared; in case of a match there is no error in the store data. Since the error case is rare, most of the time the store data comparison will register a match. Therefore, regular dissipate-on-mismatch comparators could be employed here. However, the address comparisons will most likely register a mismatch since a store address is compared against the whole write set. Here special comparators that dissipate little energy on mismatches or partial matches as proposed by Ponomarev et al. [17] could be used to save energy.

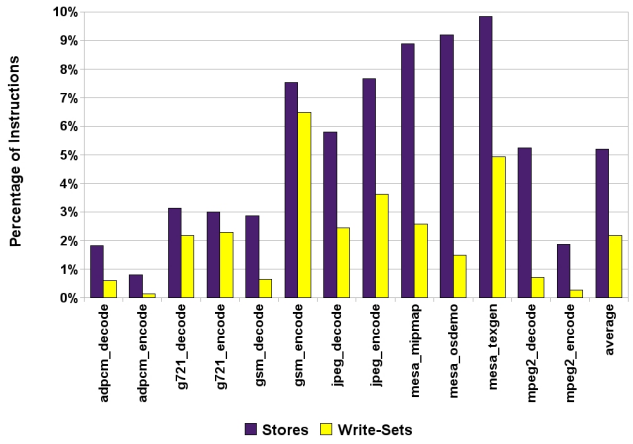

Figure 3. Store instructions vs write sets of the transactions in the FaulTM system. Write sets are smaller than the number of Stores, because some Stores write to the same addresses multiple times.

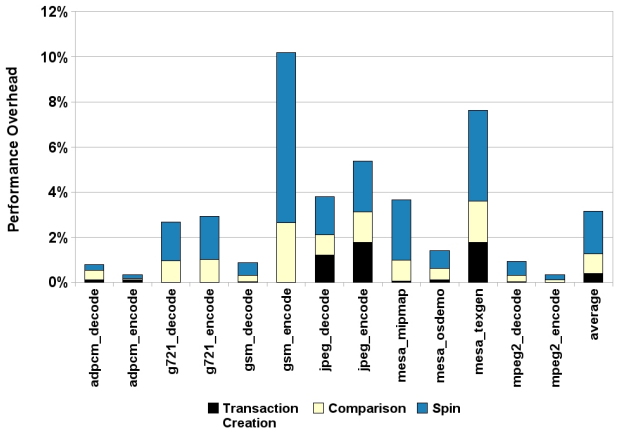

Figure 4. Performance overhead of the FaulTM technique. The commit overhead is correlated with the size of the write sets. The spin overhead is the time elapsed while one thread is waiting for the other thread to reach the commit stage. Transaction Creation overhead is the overhead of copying inner state of the original transaction to the backup one.

3. Evaluation

We now describe our simulation methodology, our simulated system, the benchmarks that we use to evaluate our techniques, and the results from our empirical evaluation of the FaulTM system.

3.1 Simulation Setup

We use the M5 full-system simulator [5] with an implementation of a Hardware Transactional Memory system that uses lazy-lazy specifications [21]. We extend this simulator with our FaulTM technique. We evaluate our approach using the Mediabench benchmark suite [10] by marking the entire applications as vulnerable.

We evaluate our technique in the context of a CMP with two Alpha 21264 cores [1] running at 1GHz. Our simulator models two levels of cache, an L1D and an L1I cache that are private to each core and a unified L2 cache that is shared by the two cores. 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 use the write-back policy. We also assume a 100 cycles of memory latency. We limit the buffer for each transaction to 20 entries.

In this initial evaluation of the FaulTM approach, we use a set of 13 sequential programs from the Mediabench benchmark suite. We run these programs in their entirety, and we mark entire programs as vulnerable.

3.2 Results

First, we show the logical boundary of FaulTM in Figure 2 by using the the Sphere of Replication(SoR) concept. Since FaulTM is a hybrid software/hardware fault tolerant method, we should depict the SoR in both software and hardware point of view. FaulTM assumes that operating system in terms of software and shared memories in terms of hardware are protected by other means.

We compare our FaulTM system against two different configurations: (1) a single thread of an application which runs on the base system and (2) two threads run on two processors using lockstepping. The first configuration corresponds to the non fault-tolerant case; the second configuration corresponds to a standard fault-tolerance method.

In the lockstepping approach [28], after every Store instruction, two threads synchronize and the results of the same instruction are compared. In this study, we assumed one cycle latency for comparing one entry in either lockstepping or FaulTM.

Figure 3 compares the ratios of Stores out of all instructions to the ratios of entries in write sets. We find

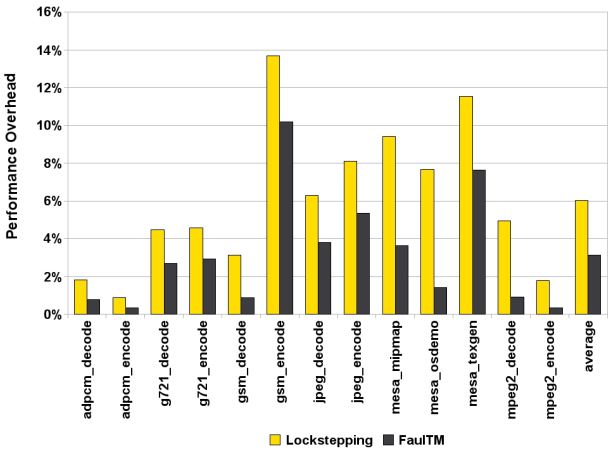

Figure 5. Comparison of the performance overheads of the FaulTM and the lockstepping techniques. Our approach performs faster than the lockstepping technique in all cases.

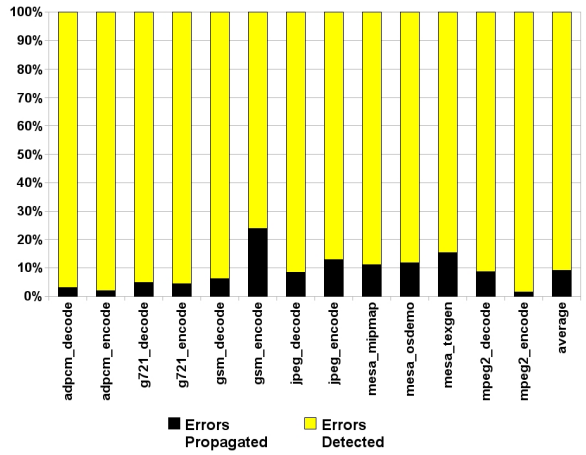

Figure 6. Percentage of the errors that can be detected inside transactions vs the errors that can propagate out of transactions. Our base FaulTM system can detect most of the possible errors inside transactions.

that, in our benchmarks, the percentages of entries in write-sets [0.1-6.5%] are smaller than the percentages of all Stores [0.8-9.8%], because some Store instructions in transactions write to the same addresses multiple times. Due to this temporal locality of the Stores our FaulTM technique requires fewer comparisons compared to the previously proposed techniques [28] in order to check soft errors.

Figure 4 shows the performance overhead of FaulTM over the single-threaded baseline system. For our benchmarks, the total performance overhead of the FaulTM is between 0.3% and 10.2%, and on average it is 3.2%. The performance overhead has three components: the transaction creation, the spin, and the comparison overheads. First, the transaction creation overhead is caused when backup transaction copies the inner state of original transaction after system calls or interrupts which is only 1.8% in the worst case. Second, the spin overhead is the time elapsed while one thread is waiting for the other thread to reach the commit stage, and in our experiments it is between 0.2% and 7.6%. Third, the comparison overhead is correlated with the size of the write sets, and it is in the range of 0.05% and 2.63%.

In Figure 5, dark bars show the performance overhead of FaulTM over the single-threaded baseline system while the light bars depict the performance overhead of lockstepping configuration. Note that, to be more conservative, we assume identical spin overheads in both systems. We find that the performance degradation of our approach is 0.3-10.2%, on average 3.2%, for sequential applications which is only 52.1% of the overhead of lockstepping technique.

3.2.1 Avoiding Error Propagation from Transactions

The base FaulTM system can detect soft errors that occur inside a transaction as long as these errors affect the instructions that attempt to modify memory. Otherwise, erroneous data in the registers may propagate out of the transaction and affect overall system reliability. We evaluate an extended FaulTM technique that compares all registers as well as write sets at every commit in order to achieve full coverage of soft errors.

Figure 6 shows that our base FaulTM approach can detect 76.2-98.5%, on average 91.2%, of all possible soft errors inside transactions. We find that, when we compare the register files of the original and backup transactions at each commit, overall performance degrades, on average, by 2.8% while providing full error coverage, see Figure 7.

4. Related Work

Although this paper only focuses on the soft errors in the circuit level caused by particle strikes, the work done by Oplinger et al. [15] which proposes reliability against software errors caused by programmers, is quite inspiring. In that system, programmers define the

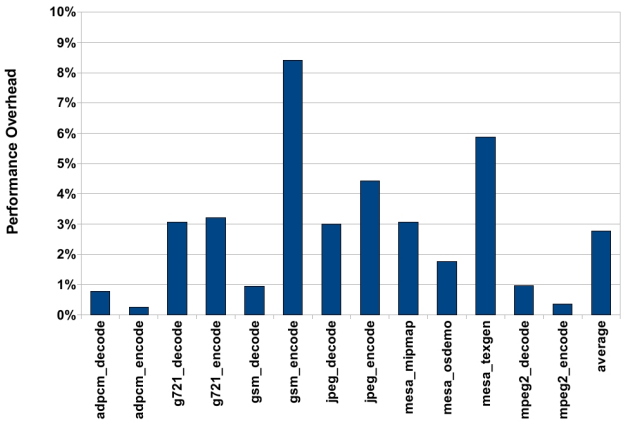

Figure 7. Additional performance overhead of the Extended FaulTM approach that compares register files at every commit.

potentially problematic code regions by using transactional memory like paradigms taking into account the transactions granularities. Errors are detected by monitoring functions that is executed speculatively in parallel. Recovery is done by benefiting from the abort mechanism of TM.

Koren and Krishna [9] provide a comprehensive introduction to fault-tolerance including sections on fault/error detection and recovery. A discussion of error containment for soft-errors can be found in Gold et al. [6]. Soft-error detection latency and coverage are discussed in [24].

There are well-known and widely used error detection and correction techniques, such as ECC [4], for storage areas and data networks. However, these techniques are not applicable to combinational circuits in microprocessors.

One line of research for detecting soft errors in microprocessors focuses on developing software-based techniques [2, 14, 16, 20]. For example, Reis et al. [20] propose a dynamic binary instrumentation technique to provide reliability to binary applications without having their source codes. Although this approach is flexible to allow the programmer to mark any part of the binary as vulnerable, its performance overhead is high.

To reduce the performance overhead of software-only techniques, another line of research focused on hardware-based approaches [3, 11, 13, 23].

Austin [3] proposes a Dynamic Implementation Verification Architecture (DIVA) to detect soft errors as well as permanent errors and design faults in microprocessors. DIVA validates the results of instructions of a complex out-of-order processor using a simple in-order low-frequency checker. This technique requires large amount of new hardware components.

A simpler and less costly soft error detection technique, lockstepping [23, 28], is commonly used, especially in mission-critical applications. In this technique, two synchronized and lockstepped processors run two identical instruction streams, and the results of every Store instruction in these streams are compared. Although lockstepping is an effective method for detecting errors, its performance overhead, due to frequent synchronization and comparisons, is high. Its processor pairs are tightly coupled using exactly the same clock signal although the processors may have the different clock domains.

To improve the performance of lockstepping methods, Mukherjee et al. [13] introduced the Redundant Multithreading (RMT) approach. In this approach, two copies of the same application run in two threads, either on the same processor or on two different processors, and the trailing thread gathers information from the leading thread to improve its execution time. Although the RMT method achieves better performance than the earlier lockstepping techniques, it still requires synchronization and comparison after every Store instruction.

In all these hardware-based approaches, entire applications are executed redundantly, i.e. the programmer does not have the flexibility to define vulnerable sections of the code.

In addition to software-only and hardware-only error detection techniques, there are also hybrid methods. For example, Shye et al. [22] propose the Process Level Redundancy (PLR) technique to adapt parallel hardware resources for soft error fault-tolerance. This method creates a set of redundant threads for the original application, and the operating system schedules these threads to all available hardware resources. Faults that result in incorrect execution are detected through either a mismatch of write data sets of the threads or a timeout mechanism. Similar to hardware-based methods, PLR does not allow the programmer to define only certain sections of the application as vulnerable.

Our FaulTM system is a hybrid approach that combine flexibility of software-based techniques and the performance of hardware-based techniques. Our approach has five main advantages over the previous soft error detection techniques: (1) It has smaller hardware cost than previous hardware methods, because it utilizes already proposed HTM hardware. (2) Unlike previous hardware methods, after every Store instruction, it does not synchronize threads and it does not compare their results, which leads to better performance. (3) It recovers errors in addition to detecting them. (4) It is flexible in the sense that the programmer can define any section of an application as vulnerable to soft errors. (5) Its pair cores are loosely coupled which let the system decide between the parallelism and the reliability.

5. Conclusions and Future Work

We introduced a soft error fault-tolerance approach, FaulTM, that leverages a hardware transactional memory system.

We design and simulate the FaulTM approach for sequential applications, in the context of an HTM system that uses lazy conflict detection and lazy data versioning policies. In this approach, the programmer can define vulnerable program sections at the software level, and these sections are executed redundantly and atomically at the hardware level. In this study we mark the entire applications as vulnerable. The FaulTM approach is flexible and requires only minor hardware modifications, and in our evaluation we found that it has lower performance overhead compared to commonly used lockstepping technique.

Currently, we are in the process of extending our work for parallel applications as well as implementing the eager-lazy and eager-eager conflict detection and data versioning policies. Also, protecting the operating system in FaulTM design is left as a future work.

Acknowledgment

We would like to thank the anonymous reviewers for their useful comments. This work is supported by the cooperation agreement between the Barcelona Supercomputing Center National Supercomputer Facility and Microsoft Research, by the Ministry of Science and Technology of Spain and the European Union (FEDER funds) under contract TIN2007-60625, by the European Network of Excellence on High-Performance Embedded Architecture and Compilation (HiPEAC) and by the European Commission FP7 project VELOX (216852). Gulay Yalcin is also supported by a scholarship from the Government of Catalonia.

References

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

[2] P. Ammann and J. Knight. Data diversity: An approach to software fault tolerance. *IEEE Transactions on Computers*, 37:418 – 425, 1988.

[3] T. M. Austin. DIVA: A dynamic approach to microprocessor verification. *Journal of Instruction-Level Parallelism*, 2:1–6, 2000.

[4] D. J. Baylis. *Error Correcting Codes: A Mathematical Introduction*. Chapman and Hall, 1998.

[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] B. Gold, J. C. Smolens, B. Falsafi, and J. C. Hoe. The granularity of soft-error containment in shared-memory multiprocessors. In *Proceedings of the 2006 Workshop on System Effects of Logic Soft Errors*, Urbana-Champaign, 2006.

[7] S. Gupta, F. Sultan, S. Cadambi, F. Ivancic, and M. Rotteler. Using hardware transactional memory for data race detection. In *IPDPS '09: Proceedings of the IEEE International Symposium on Parallel & Distributed Processing*, pages 1–11, Miami, Florida, 2009.

[8] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. *SIGARCH Computer Architecture News*, 32(2):102, 2004.

[9] I. Koren and C. M. Krishna. *Fault-Tolerant Systems*. Morgan Kaufmann Publishers, 2007.

[10] C. Lee, M. Potkonjak, and W. H. Mangione-smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of 30th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 330–335, Research Triangle Park, NC, 1997.

[11] A. Mahmood and E. J. McCluskey. Concurrent error detection using watchdog processors-a survey. *IEEE Transactions on Computers*, 37:160–174, 1988.

[12] K. Moore, J. Bobba, M. Moravan, M. Hill, and D. Wood. LogTM: log-based transactional memory. volume 12, pages 254–265, Austin, Texas, 2006.

[13] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt. Detailed design and evaluation of redundant multithreading alternatives. In *Proceedings of the 29th Annual*

International Symposium on Computer Architecture, pages 99–110, Washington, DC, 2002.

[14] N. Oh, S. Mitra, and E. J. Mccluskey. ED4I: Error detection by diverse data and duplicated instructions. *IEEE Transactions on Computers*, 51:180–199, 2002.

[15] J. Oplinger and M. S. Lam. Enhancing software reliability with speculative threads. In *Architectural Support for Programming Languages and Operating Systems*, pages 184–196, San Jose, 2002.

[16] J. H. Patel and L. Y. Fung. Concurrent error detection in ALU's by recomputing with shifted operands. *IEEE Transactions on Computers*, 31(7):589–595, 1982.

[17] D. V. Ponomarev, G. Kucuk, O. Ergin, and K. Ghose. Energy efficient comparators for superscalar datapaths. *IEEE Transactions on Computers*, 53:892–904, 2004.

[18] L. Porter, B. Choi, and D. M. Tullsen. Mapping out a path from hardware transactional memory to speculative multithreading. In *PACT '09: Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 313–324, Washington, DC, USA, 2009.

[19] S. K. Reinhardt and S. S. Mukherjee. Transient fault detection via simultaneous multithreading. *SIGARCH Computer Architecture News*, 28(2):25–36, 2000.

[20] G. A. Reis, J. Chang, D. I. August, and S. S. Mukherjee. Configurable transient fault detection via dynamic binary translation. In *Proceedings of the 2nd Workshop on Architectural Reliability*, Orlando, Florida, 2006.

[21] S. Sanyal, S. Roy, A. Cristal, O. S. Unsal, and M. Valero. Dynamically filtering thread-local variables in lazy-lazy hardware transactional memory. In *HPCC '09: Proceedings of the 11th IEEE International Conference on High Performance Computing and Communications*, pages 171–179, Washington, DC, 2009.

[22] A. Shye, V. J. Reddi, T. Moseley, and D. A. Connors. Transient fault tolerance via dynamic process redundancy. In *Proceedings of the 2006 Workshop on Binary Instrumentation and Applications (WBIA) held in conjunction with ASPLOS-12*, San Jose, CA, 2006.

[23] T. J. Slegel, R. M. Averill III, M. A. Check, B. C. Giamei, B. W. Krumm, C. A. Krygowski, W. H. Li, J. S. Liptay, J. D. MacDougall, T. J. McPherson, J. A. Navarro, E. M. Schwarz, K. Shum, and C. F. Webb. IBM's S/390 G5 microprocessor design. *IEEE Micro*, 19(2):12–23, 1999.

[24] 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, Boston, MA, 2004.

[25] F. Tabba, A. W. Hay, and J. R. Goodman. Transactional value prediction. In *Workshop on Transactional Computing*, Raleig, North Carolina, 2009.

[26] S. Tomić, C. Perfumo, C. Kulkarni, A. Armejach, A. Cristal, O. Unsal, T. Harris, and M. Valero. Eazyhtm: eager-lazy hardware transactional memory. In *Micro-42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 145–155, New York, NY, USA, 2009.

[27] M. Tremblay and S. Chaudhry. A third-generation 65nm 16-core 32-thread plus 32-scout-thread CMT SPARC(R) processor. In *International Solid-State Circuits Conference (ISSCC)*, pages 82–83, San Francisco, 2008.

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