FaulTM-multi: Fault Tolerance for Multithreaded Applications Running on Transactional Memory Hardware

Gulay Yalcin*,§ Osman S. Unsal* Adrian Cristal*,† Mateo Valero*,§
* Barcelona Supercomputing Center
§ Universitat Politecnica de Catalunya
† IIIA - Artificial Intelligence Research Institute - CSIC - Spanish National Research Council
{gulay.yalcin, osman.unsal, adrian.cristal, mateo.valero}@bsc.es

ABSTRACT

Fault-tolerance has become 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 and silent data corruptions. However, comparing the results of the instruction streams, checkpointing the entire system and recovering from the detected errors present high performance degradation in execution time. This paper presents FaulTM-multi, transactional memory based fault detection and recovery scheme for multi threaded applications running on transactional memory hardware in order to reduce these performance degradations.

1. INTRODUCTION

Fault tolerance has become a first-class design constraint for processor designers since it is foreseen that technology trends will increase the transient and permanent fault rates in future processors [3, 2]. Tolerating errors in processors' core logic presents more difficult challenges than errors in storage devices which can be mitigated by error detection and correction codes (ECC). Lockstepping [24, 18], a conventional error detection scheme for processor cores, executes the instruction streams in two synchronized and lockstepped processors redundantly and checks if both produce identical results. This scheme provides the highest reliability since it can detect permanent faults and Silent Data Corruptions (SDC) as well as transient faults. However, coupling cores tightly becomes unfeasible as device scaling continues. Thus, researchers have proposed several alternatives to lockstepping [13, 22, 11, 19, 20, 25].

Besides error detection, error recovery is also an indispensable aspect of fault tolerance. Global checkpointing is a well-known error recovery scheme in which the detected errors are recovered by rolling back all processors to an earlier validated state [15, 21]. However, it is foreseen that in future many-core processors, global checkpointing may take even more time than the execution of applications [14]. Thus, providing recovery for parallel applications becomes harder since an error in one processor may propagate to the others easily through shared variables between threads. Moreover, maintaining identical instruction streams to the redundant executions, called the input replication problem, is a challenge in these applications as well. Thus, there are only a few reliability studies which conduct experiments on multithreaded applications without using global checkpointing [19, 17].

Yalcin et al. propose FaulTM [25], an architectural error detection and recovery proposal for sequential applications leveraging Hardware Transactional Memory (HTM). It is proved that FaulTM reduces the overhead of comparison and synchronization which is required in order to detect divergent execution. However, the design for multi-threaded applications is not presented in FaulTM. In this study, we present FaulTM-multi, a fault tolerance scheme for multi threaded applications running on multi core architectures. FaulTM-multi leverages HTM in order to reduce the comparison overhead and to avoid global checkpointing. 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 canceled; the 48-core Vega2 chip from Azul systems uses HTM [8] to accelerate Java is widely used. Recently, AMD detailed their Advanced Synchronization Facility (ASF) HTM proposal for X86 [7], and patents filed by IBM [6] indicate that HTM is likely to be implemented in the 16-core PowerPC processor which will be featured in IBM's next generation BlueGene/Q supercomputer. FaulTM-multi utilizes HTM to provide reliability for parallel applications (non-TM and TM) against transient and permanent faults. It 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.

2. BACKGROUND FOR TM AND FAULTM

Transactional Memory (TM) is a promising technique which aims to simplify 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 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 the transactions successfully commit or abort. Conflict detection tracks the 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

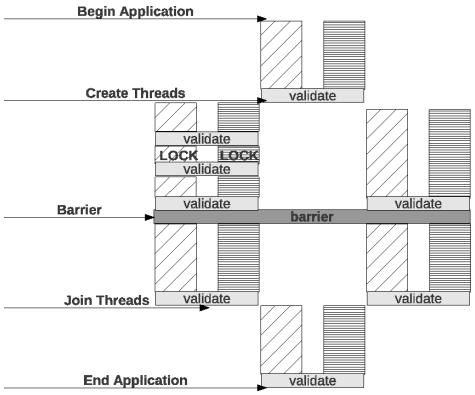


Figure 1: FaulTM-multi for Parallel Applications

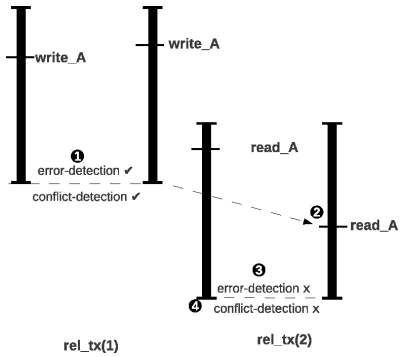


Figure 2: Solving Input Replication in FaulTM

version of the data is written (published) to the shared memory at the end of the transaction.

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, 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. Since data in cache and memory structures are protected by ECC which is generated during the execution of store instructions, "*reading/writting 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. Also, since input messages should be replayed after recovery, input commit presents problem in fault tolerance as well. In the first design of FaulTM [25], as soon as any I/O instruction is detected in a rel-tx, operations in rel-txs are validated and commited up to that point. Only one thread executes the I/O operation. After returning from the operating system new original and backup rel-txs are started. However, in FaulTM-multi design, we adopt the standard solution in which output values are delayed until validation (end of rel-txs in our 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 multithreaded applications which are (1) handling instructions dedicated to maintaining synchronization such as locks, barriers or create/join threads and (2) maintaining identical instruction streams of redundant threads.

In FaulTM, the system can not be aware of the instructions in rel-txs until it commits. 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 deadlock. 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 FaulTM-multi (Figure 1), when a rel-tx encounters one of these special instructions, it first validates and commits before executing the instruction and starts a new rel-tx pair. After the execution of the special instruction, FaulTM-multi starts the commit process in order to make the system aware of the instruction. This way, even if one of the rel-txs aborts due to a detected error, it will not try to reacquire the lock that it already held, which could have led to a deadlock. Note that lock release operations do not force rel-tx to commit since delaying lock release instructions until the end of rel-tx does not present any forward progress issues. Also, 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.

The second challenge is maintaining identical instruction streams. In particular, input incoherence can be a problem which occurs 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 2, we demonstrate how the conflict detection mechanism of TM solves the input incoherence problem. In the example, one of the rel-tx(2) reads the old version of A while the other 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. 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

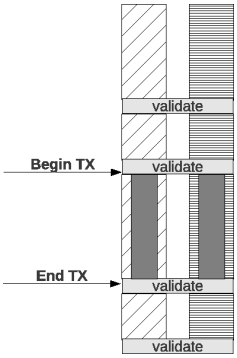


Figure 3: Design for TM Applications

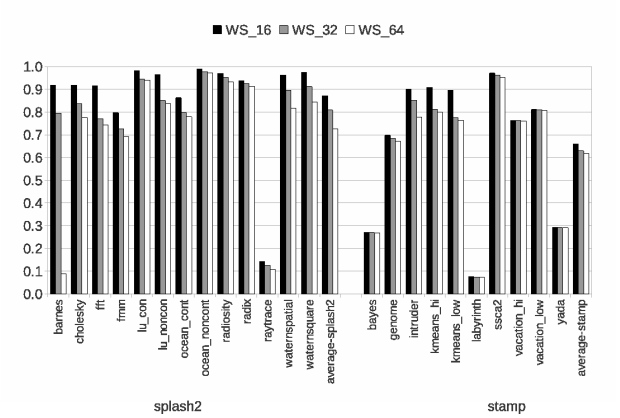


Figure 4: Total entries in write-sets normalized according to the total number of store instructions.

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 [9, 12]. 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 3). We leave supporting reliability with other HTMs such as eager-lazy, eager-eager or hybrid-policy [10] systems as future work.

When a thread encounters the instruction that starts a transaction, FaulTM 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. If there is an irrevocable operation in a transaction, TM marks this transaction as such and the transaction does not abort. 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 at the end of the irrevocable transaction, it can rollback 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, 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.

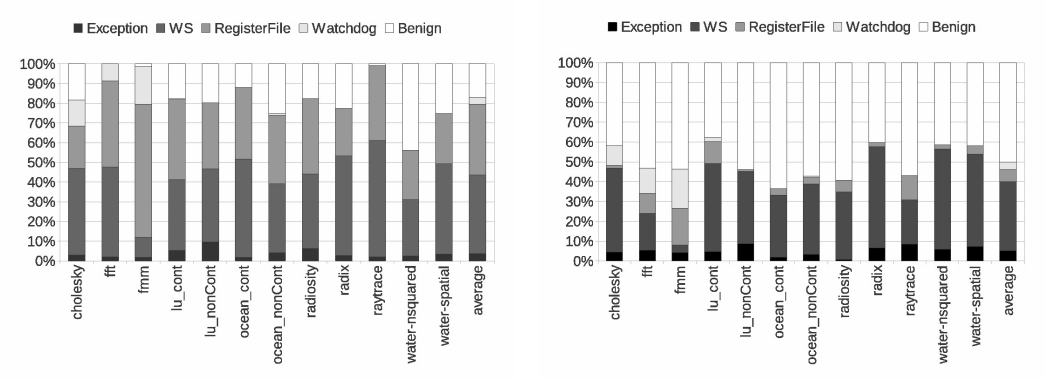
5. EVALUATION

We use the M5 full-system simulator [4] with an implementation of a Hardware Transactional Memory system that uses lazy data versioning and lazy conflict detection [16]. We extend this simulator with our FaulTM-multi implementation. We evaluate our approach with 4 original and 4 backup threads using splash2 [23] and stamp [5] benchmark suites which are the representative of non-TM parallel and TM applications. 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. Main memory latency is 100 cycles. We use fault injection to measure the reliability performance of FaulTM-multi for both transient and permanent faults.

We compare our FaulTM-multi system against lockstepping, a standard fault-tolerance method. In the lockstepping approach, after every store instruction, two threads synchronize and the results of the store are compared.

FaulTM 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 the same addresses. In Figure 4 we present this reduction on different write-set sizes (i.e. 16, 32 and 64 entries) which should be determined during the design time of HTM. In the figure, each bar represents the normalized value of the total amount of entries in write-sets as compared to total stores. It is obvious that when we increase the size of write-sets, the ratio of compared data reduces.

We find that, in our benchmarks, the percentages of entries in write-sets are smaller than the percentages of all



(a) Transient Faults

(b) Permanent Faults

Figure 6: Error Recovery Performance of FaulTM

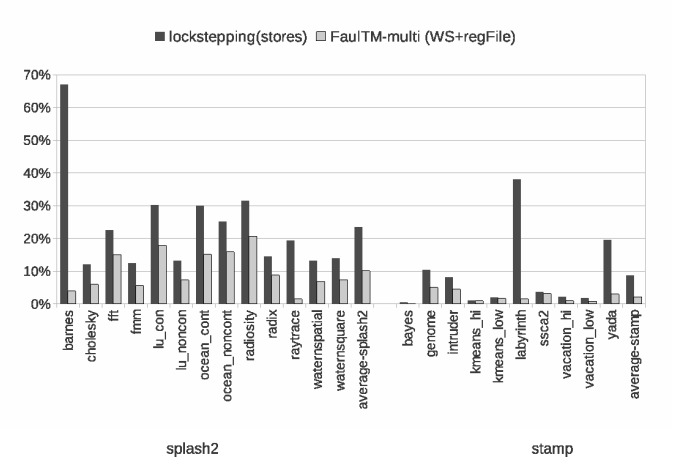


Figure 5: The overhead of FaulTM vs lockstepping in the execution time of the applications

stores (35% less on average among all benchmarks), 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 technique requires fewer comparisons compared to lockstepping in order to check errors.

In Figure 5, we compare the performance overhead of FaulTM with lockstepping including comparison and spin overheads. We use 64 entries in the write-sets of FaulTM. Note that, lockstepping compares only the store values (not register file) that can not guarantee the error free execution since the last validation. Also, lockstepping requires additional checkpointing mechanisms (we did not take into account the checkpoint creation overhead of lockstepping). Comparatively, FaulTM has a negligible transaction creation overhead since it utilizes HTM. We find that, on average, the performance degradation of our approach is %10, %2 for splash2 and stamp applications which is 56%, 75% less than lockstepping. Register file comparison of FaulTM costs only 1.7% overhead on the execution time on average (it is included in FaulTM bar). Register value comparison of lockstepping would cause extremely high overhead (not included

Benchmark	Size	Benchmark	Size
barnes	2548	cholesky	909
fft	149	fmm	1634
lu_con	516	lu_noncon	387
ocean_cont	642	ocean_noncont	623
radiosity	110	radix	705
raytrace	5801	waternspatial	458
waternsquare	421	bayes	152716
genome	567	intruder	591
kmeans_hi	1886	kmeans_low	1043
labyrinth	50314	ssca2	505
vacation_hi	6545	vacation_low	6520
yada	7668		

Table 1: Average Number of Instructions in a rel-tx.

in the evaluations here) due to the frequency of instructions which updates the register file.

Figure 6 presents the reliability performance of FaulTM. 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, our FaulTM 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 avoids detecting benign faults which is 20% of injected transient faults and 32% of injected permanent faults. The error recovery overhead of FaulTM comes from the re-execution of the instructions from the beginning of the rel-tx. Table 1 shows the average number of instructions in transactions which is very low (generally less than 10K instructions) compared to system-wide checkpointing mechanisms (10M instructions) such as ReVive [15] or SafetyNet [21]. Note that, smaller checkpoint interval is essential to support I/O operations [20].

6. 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 (non-TM 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.

7. ACKNOWLEDGMENTS

We would like to thank Gokcen Kestor, Roberto Gioiosa, Ruken Zilan and 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), by the G8 Group of countries Enabling Climate Simulation at Extreme Scale project, and by the European Commission FP7 project VELOX (216852). Gulay Yalcin is also supported by a scholarship from the Government of Catalonia.

8. 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, 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.

[5] 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.

[6] 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.

[7] 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. *Microarchitecture, IEEE/ACM International Symposium on*, 0:39–50, 2010.

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] R. A. Oldfield, P. J. Teller, M. R. Varela, P. C. Roth, S. Arunagiri, S. Seelam, and R. Riesen. Modeling the Impact of Checkpoints on Next-generation Systems. In *Proceedings of the 24th IEEE Conference on Mass Storage Systems and Technologies*, pages 30–43, 2007.

[15] 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.

[16] 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.

[17] S. K. Sastry Hari, M.-L. Li, P. Ramachandran, B. Choi, and S. V. Adve. mSWAT: Low-Cost Hardware Fault Detection and Diagnosis for Multicore Systems. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 122–132, 2009.

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.