Managing Soft-errors in Transactional Systems

Mohamed Mohamedin, Roberto Palmieri, Binoy Ravindran
Electrical and Computer Engineering Department
Virginia Tech
Blacksburg, Virginia, USA
mohamedin@vt.edu, robertop@vt.edu, binoy@vt.edu

***Abstract*—Multicore architectures are becoming increasingly prone to soft-errors – i.e., transient faults caused by external physical phenomena such as electric noise and cosmic particle strikes. With increasing core counts, the soft-error rate is growing due to the accelerating transistor density on chips. The impact of these errors on business-critical applications that are being deployed on multicore hardware can be significant. We present an active replication-based approach that fully masks such errors for transactional applications. We partition computational cores, fully replicate objects across partitions, and concurrently execute transactional requests on all partitions, thereby enabling completely local object accesses. Transactional requests are globally ordered and delivered across partitions using optimistic atomic broadcast. Hardware message passing – an important emerging trend in multicore architectures – is exploited to mitigate communication costs. We report preliminary results obtained with an implementation of our approach on a 36-core Tilera TILE-Gx hardware, with an on-chip scalable mesh network.**

***Keywords*-Transaction Processing; Soft Errors; Active Replication**

I. Introduction

Soft-errors [1] are transient faults that may happen anytime during application execution. They are caused by physical phenomena [2], e.g., cosmic particle strikes, electric noise, which cannot be directly managed by application designers or administrators. As a result, when a soft-error occurs, the hardware is not affected by interruption, but applications may crash or behave incorrectly.

The trend of building smaller devices with increasing number of transistors on the same chip is allowing designers to assemble powerful computing architectures – e.g., multicore processors. Although the soft-error rate of a single transistor has been almost stable over the last years, the error rate is growing in current and emerging multicore architectures due to rapidly increasing core counts [2]. As increasing number of enterprise-class applications, especially those that are business-critical (e.g., transactional applications), are being built for, or migrated onto such architectures, the impact of transient failures can be significant.

A soft-error can cause a single bit in a CPU register to flip (i.e., residual charge inverting the state of a transistor). Most of the time, such an event is likely to be unnoticed by applications because they do not use that value (e.g., unused register). However, sometimes, the register can contain an instruction pointer or a memory pointer. In those cases, the application behavior can be unexpected. An easy solution for recovering from transient faults is a simple application-restart, but such solutions are unacceptable for transactional applications due to availability/reliability constraints, and also due to performance and business reasons.

Replication is a widely considered approach for ensuring fault-tolerance. However, replicating centralized systems for tolerating faults results in significantly degraded performance (e.g., 100–1000×) [3]. This is primarily due to costs for remote synchronization and communication that are incurred for ensuring replica consistency. Also, faults addressed by replication techniques are not transient; they often target hardware deficiency or node crashes. Further, a distributed architecture comprising of multicore nodes may not often be cost-effective.

Besides the multicore trend in computer architecture, another interesting trend is emerging on the interconnect fabric (of multicore chips). With growing core counts, bus-based interconnects, and hardware cache-coherence protocols that use such interconnects to provide the illusion of shared memory are increasingly becoming a scalability bottleneck [4]. Network-based communication using on-chip message passing architectures are beginning to appear in many multicore chips – e.g., Intel SCC [5], Intel Xeon Phi [6], Tilera [7]. This hardware trend opens up new possibilities at the algorithmic-level: communication-intensive distributed algorithms (e.g., consensus), which are expensive in a distributed setting, are likely to become cost-effective in multicore.

Motivated by these observations, we propose a replication-based methodology for tolerating transient failures of transactional applications running on massively multicore architectures. Our key idea is to partition the available resources, and run the application's transactions on all the partitions in parallel, according to the *active replication* paradigm [8] (Section II). As in active replication, transaction requests are wrapped into network messages that the application submits to an ordering layer. This layer implements a version of *Optimistic Atomic Broadcast* [9] (OAB), and ensures a global serialization order (GSO) of messages among partitions (Sections III–IV).

Each partition is thus notified with the same sequence of transactions and is able to process those transactions independently from the other partitions. Objects accessed by transactions are fully replicated so that each partition can access its local copy, avoiding further communications. Transactions are processed according to the GSO. Additionally, we use a *voter* for collecting the outcome of transactions and delivering the common response (i.e., majority of the replies) to the application. If one or more replies differ, actions are triggered to restore the consistent status of data in those faulty partitions.

We implemented this framework in C++. It consists of five pluggable components: application support, ordering layer, dispatcher, concurrency control, and voter. All components share a synchronized clock, which is available in hardware in all modern (multicore) architectures.

We use in-memory transactional applications (e.g., [10]) as an initial candidate for evaluating the proposed approach. Such applications are good candidates as they typically use multicore architectures and do not use stable storage to log their actions, both for achieving high performance. Moreover, they are susceptible to soft-errors due to their in-memory nature. We used a 36-core manycore processor of the Tilera TILE-Gx family [7], which is equipped with an on-chip message passing hardware, for an early experimental evaluation (Section V). Our preliminary results reveal that the proposed approach ensures fault-tolerance with competitive performance of centralized (i.e., non fault-tolerant) systems, but without paying the cost (both in terms of performance degradation and financial cost) of deploying a distributed infrastructure.

II. The Protocol

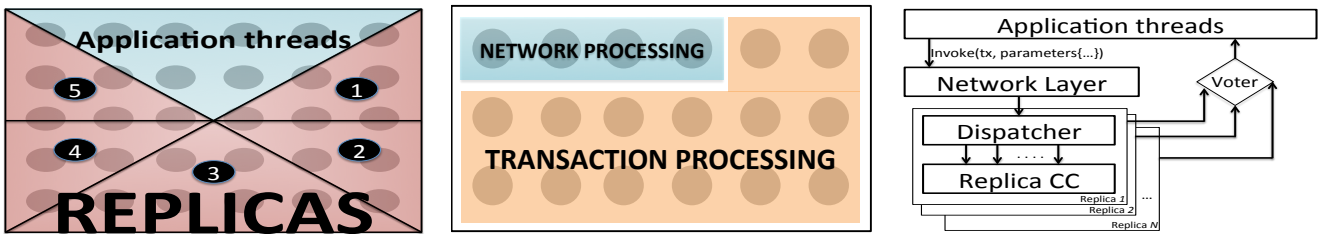
Our basic idea for tolerating soft-errors is to logically partition computational resources into groups. Each group is composed of a subset of available cores, and is responsible for processing transactional requests issued by application threads. We use the active replication paradigm [8] for processing transactions. A transaction request is sent to all the groups (or *replicas* hereafter), relying on a group communication system (GCS). (Applications do not execute transactions in the same requesting thread.) GCS, implementing a total order service (e.g., Atomic Broadcast [11]), is responsible for delivering the same set of transaction requests to all the replicas, in the same order. Processing the same sequence of transactions allows the replicas to reach the same state. The approach thus masks failures, and is generally used in distributed computing for preventing loss of data due to crash or service interruption.

Unfortunately, total order protocols involve the exchange of several network messages by the replicas for each transactional request to agree on a common delivery order [3], which is often a performance bottleneck. To alleviate this, we rely on an optimistic version of total order protocol presented in [9]. This new protocol includes an additional delivery, called *optimistic delivery*, which is notified by the total order layer to the replicas before the (*final*) notification of the message, along with its order. This early delivery has a twofold benefit. First, early delivery notifies that a new message has been previously broadcast, and that message is currently in the coordination phase for defining its final order. Second, early delivery defines an implicit optimistic order. Although this order can be used for executing transactions speculatively, overlapping their processing with their coordination, it is not reliable and cannot be considered for the transaction commit phase. As a result, when the optimistic order and the final order coincide, the transaction can be validated and committed (if completely executed) without paying the cost of its re-execution from scratch.

An ordering-based concurrency control (ObCC) protocol, running locally on every replica, is responsible for committing transactions following the order defined by the sequence of final deliveries issued by the GCS. A subset of cores in each replica is dedicated for the execution of ObCC. In order to maximize the overlapping, transactions are processed in parallel as soon as they are optimistically delivered. When a conflict arises, ObCC resolves it for meeting the order defined by the sequence of optimistic deliveries. Whenever the optimistic order is the same as the final order, or the transaction does not conflict with others, the transaction is likely completely executed and committed after validation.

The entire shared data-set is replicated on each replica. This enables ObCC to process transactions locally, making each replica independent from the others. When a soft-error occurs on a replica, the others are able to serve the transaction request without additional coordination i.e., the failure is *fully masked*. A *voter* is in charge of collecting outcomes from the transactions processed on all the replicas, and returns the majority outcome to the application. Even though this approach potentially increases the end-to-end transaction latency, all the replicas are part of the same architecture, running at the same clock speed, and are exposed to the same workload. As a consequence, the transactions' results are received almost simultaneously by the voter, without affecting the overall performance. The voter also identifies the replicas affected by soft-errors (i.e., presenting an outcome that differs from the one delivered to the application), and restores a consistent state from a correct replica.

Figure 1 schematizes the architecture. Figure 1(a) shows how the computational resources are partitioned: a group of cores is reserved for running application threads, and the remaining cores are grouped for running replicas. Figure 1(b) focuses on each replica: a small number of cores is reserved for running the total order protocol, and the rest are used for running ObCC. Figure 1(c) illustrates the software architecture. Application threads invoke transactions through the network layer. A *dispatcher* is responsible for fetching



(a) System (b) Replica (c) Software

Figure 1. System, replica and software architecture.

transaction requests from the network layer and processing them under the control of ObCC. A voter collects the results and replies to the original application's invoking call.

Though this approach overcomes soft-errors in a transparent manner from the application's point of view, it has two main drawbacks. The first is the reduced number of cores available for transaction processing. However, our approach targets massively multicore architectures, where the number of cores is sufficiently large for exploiting application concurrency. Additionally, transactional applications typically do not consume all the resources available because of logical contentions (e.g., locks). In our approach, the number of replicas can be tuned according to the fault resilience desired, and also according to the expected resource utilization. The second drawback is the increased memory consumption. In order to support our architecture, each group replicates all the shared objects, increasing the total memory utilized. However, this is not a major restriction, because, memory cost is rapidly decreasing and many medium-level servers are now equipped with 16/32/64Gb.

III. Network Layer

According to the active replication paradigm, transactions need to be totally ordered before they are delivered to each replica. Our approach includes an optimized network layer ordering transaction requests issued by application threads. This layer is decentralized, each replica reserves one core for executing its part of the work.

Our network algorithm is a variant of Optimistic Atomic Broadcast [9]. We assume a monotonically increasing synchronized clock (called *timestamp*), which is typically available in centralized architectures. The hardware that we use for experimental evaluation (Section V), the Tilera TILE-Gx, provides this service by the *cycle-counter-register*[1].

Each application thread sends its requests to other replicas and application threads. The request is composed of: application ID, the transaction's name with its parameters, and the sender's timestamp. Other application threads acknowledge the request using the local timestamp as ACK message. Reading a timestamp from the cycle-counter-register ensures that the next call retrieves a greater value.

When replicas receive the request, they immediately deliver it optimistically. The optimistic order reflects the request's timestamp. After the early delivery, replicas wait for messages tagged with higher timestamp (either ACK or new requests). After one message is received from every application thread, the final delivery is issued.

This protocol defines an order among requests issued by application threads. We do not need to synchronize replicas because timestamps are generated from the *cycle-counter-register*, which ensures monotonicity.

IV. Concurrency Control

Each replica is equipped with local concurrency control. Transactions are activated as soon as they are optimistically delivered (*opt-del* hereafter). When a conflict with other opt-del transactions occur, it is resolved using the optimistic order of the conflicting transactions. Consider two transactions T_1 and T_2. Let their opt-del order be T_1 followed by T_2. When a conflict occurs, T_2 is aborted and restarted in order to allow T_2 to access data written by T_1. Following this basic rule, the conflicting opt-del transactions are processed according to their opt-del order.

Algorithms for processing uncommitted transactions in-order has been proposed in [12]–[15]. In our approach, we rely on the conflict detection technique of SwissTM [16], and extend it to enforce ordering. Each shared object has the committed and a list of completed versions (written by opt-del transactions).

Similar to SwissTM, we use write-locks. Each writing transaction must acquire the write-lock on the memory location. If it is locked by an older transaction (according to the opt-del order), then it aborts. Otherwise, the transaction holding the lock is aborted. When reading a location, if it is locked by an older transaction, then the reader waits until that transaction finishes.

The commit is split into two phases. The first is called the *complete phase*, where the transaction is not yet final-delivered. The transaction is validated and the write-set values are written into a *complete version* buffer. The subsequent transactions can therefore read the forwarded

[1]cycle-counter-register is well documented in the Tilera's handbook

values without waiting for final commit. The second phase is the *commit phase*, where values are written to shared memory after validation, or a cascaded abort occurs if validation fails (e.g., the opt-del order is not confirmed by final delivery order). Additionally, we manage the priority of threads for avoiding instrumentation overhead for the threads processing the next final (and not optimistically) delivered transaction. Such threads can directly operate in memory without synchronizing with others since they are processing the next committing transaction non-concurrently on the committed versions.

V. Preliminary Evaluation

We developed a preliminary implementation of the proposed approach and conducted an early experimental evaluation. We implemented all of the modules in C++ The network layer, in particular, was designed and implemented to be platform-independent: it can run transparently on both message-passing and shared memory hardware. Our ordering-based concurrency control approach (ObCC) was implemented on top of RSTM library [17]. It uses platform-specific assembly instructions, so we ported the original implementation to our test-bed.

Our evaluation used a 36-core board of the Tilera TILE-Gx family [7]. This hardware is commonly used as an accelerator or intelligent network card. Each core is a full-featured 64-bit processor (1.0 GHz clock speed, 8 GB DDR3 memory), with two level of caches, and a non-blocking mesh that connects the core to the Tilera 2D-interconnection system. We installed the Tilera board as a co-processing platform to a 64-core AMD Opteron machine, interconnecting it via the PCIe bus.

Network Layer. To understand the performance of the network layer without transactional workloads, we conducted an experiment in which we reserved 2 cores, and ran each replica on one core, exclusively for logging the messages optimistically and finally delivered. (Our implementation currently does not batch messages, which is a common technique for increasing performance.)

We injected transactional requests, varied the number of replicas, and calculated the throughput (i.e., number of messages totally ordered per second) from the time needed for delivering all the requests to the slowest replicas. Since each replica executes with the same clock, the gap between fastest and slowest replicas is negligible.

Figure 2 shows the results. They reveal reasonable performance (considering that messages are not batched), especially with 4–8 replicas. After 8 replicas, the throughput is affected by the saturation of the sending/ delivery queues. However in centralized systems having more than 8 replicas is not desirable because the available computing resources are significantly limited.

The average latency between the optimistic and final deliveries was found to be 8.5*msec*. This latency magnitude

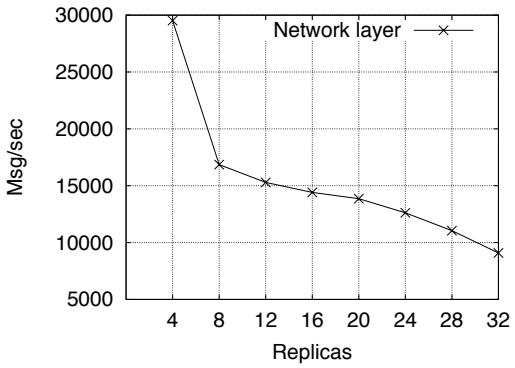


Figure 2. Throughput of network layer.

enables concurrency control to process transactions speculatively in the optimistic delivery order.

Concurrency Control. ObCC enforces a deterministic commit order of transactions, which is mandatory for implementing active replication-based protocols. However, this impacts performance. For example, consider two independent transactions T_1, which is long running, and T_2, which is short, delivered in this order. Under ObCC, T_2 cannot commit before T_1, even if the transactions are non-conflicting, which degrades performance, but ensures a deterministic commit order. To understand this overhead, we compared ObCC with SwissTM [16]. We used two benchmarks in this study: *Hash-Set* and *ReadWriteNBench*. The latter generates threads that read and write at random locations in a shared matrix.

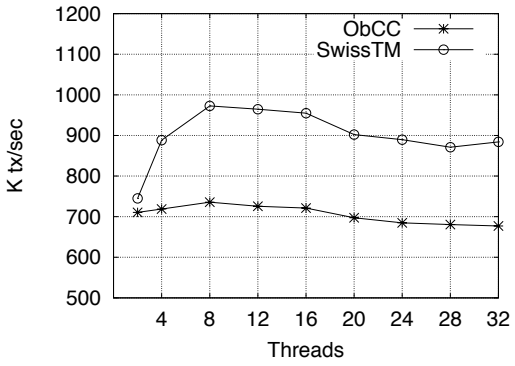


Figure 3. Throughput of ObCC for Hash-Set.

Our current implementation does allow transactions to process speculatively when optimistically delivered, and they are activated only upon final delivery. Overlapping transaction processing with coordination is one of our future tasks. For the purpose of these experiments, transactions were generated locally without any ordering layer, and their starting time defined the commit order.

Figure 3 shows the results for Hash-Set. We observe a

maximum performance degradation for ObCC to be ≈25%. This degradation is much lower than the gap in performance between the concurrency control and the ordering layer (i.e., the performance upper bound in actively replicated systems for write-transactions [18]). As a result, ObCC will not impact the overall performance when the network layer will inject transactional requests.

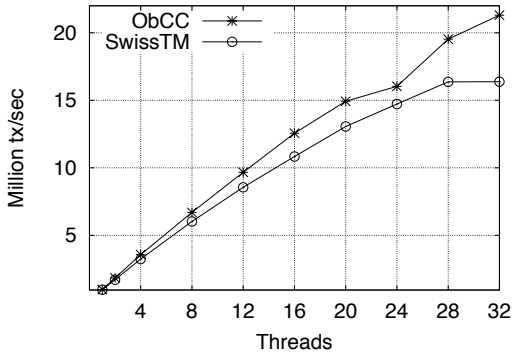


Figure 4. Throughput of ObCC for ReadWriteNBench.

Figure 4 shows the results for ReadWriteNBench. Here ObCC outperforms SwissTM. The reason is twofold. First, this benchmark has low conflicts, and therefore transactions can be committed in order without being aborted for processing in the wrong order. Second, the cost of *CAS* operations in SwissTM for managing the timestamps are replaced with Tilera's cycle-counter-register, which avoids invalidation and cache misses.

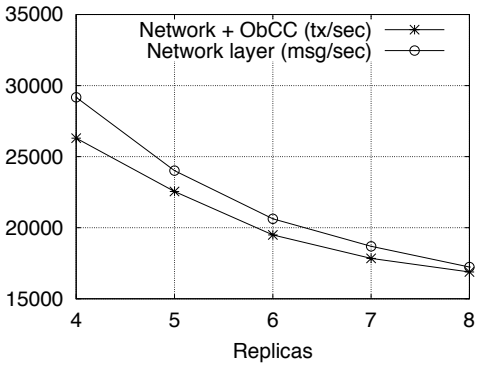


Figure 5. Throughput of integrated infrastructure.

Integrated Infrastructure. Figure 5 shows the performance of the integrated infrastructure using Hash-Set benchmark. In this preliminary implementation, ObCC processes transactions after their final delivery, instead of starting their speculative execution exploiting the optimistic delivery. The maximum throughput of this system is bound by the performance of the total order service (the second line in the plot). The results show a limited gap between the performance of the network layer and the overall system. This is promising, as overlapping coordination and processing will likely reduce the gap.

Comparing Figure 4 with Figure 5 (i.e., pure transaction processing without the total order service for issuing transactions), we conclude that, even with a preliminary, non-optimized implementation, the performance gap observed is about 36x. In fact, with Hash-set, SwissTM's average throughput is ≈900K tx/sec, whereas our fault-tolerant architecture with 4 replicas yields ≈25K tx/sec. This gap is limited, compared to the performance degradation incurred when moving from a centralized to a distributed setting, and can likely be reduced by optimizing the components, in particular the network layer.

VI. Related Work

Transient faults due to radiation effects on semiconductor devices have been widely studied in the last decade [19]–[22]. An excellent survey on their causes, and impact on modern architectures can be found in [2].

Solutions for overcoming soft-errors can be classified into hardware- [23]–[25] and software-based [26], [27]. The former assumes controlling the underlying hardware, and typically involves enhancing it (e.g., redundancy). The latter provides execution frameworks, and often uses online failure detectors that trigger recovery actions when a soft-error occurs (e.g., restarting execution). Our approach is software-based and is transparent from the hardware's specification.

Monitoring frameworks for detecting soft-errors are proposed in [28], [29]. They provide feedback when such errors occur, but do not integrate recovery mechanisms. Our approach analyzes transaction outcomes (no monitoring) and, when a corruption occurs, a restoration of the corrupted partition's state is issued.

Active replication is a well known paradigm in transaction processing [9], [12]–[14]. These works exploit OAB protocols for processing transactions speculatively, overlapping their execution with the coordination time for ordering. They provide resilience to crash/stop failures; in case of soft-errors, data consistency can be corrupted.

Transactional applications are increasingly using Software Transactional Memory (STM) algorithms [30], [31] and frameworks [16], [32] to overcome the programmability and performance challenges of in-memory processing on multicore architectures. The concurrency control algorithm that we present is inspired by SwissTM [16].

VII. Conclusions

Our preliminary results reveal that the active replication paradigm is a candidate for making transactional systems resilient to application crashes due to soft-errors.

A number of future directions can be considered for the proposed framework. For example, each component

can be independently optimized. The network layer can be enhanced using batching for increasing the throughput of the total order service. The ObCC can be extended to speculative processing of optimistically delivered transactions, instead of waiting for their final delivery. It would also be interesting to port other active replication solutions into our overall framework and compare with our proposal.

ACKNOWLEDGEMENT

This work is supported in part by US National Science Foundation under grant CNS 1217385.

REFERENCES

[1] S. Borkar, "Designing reliable systems from unreliable components: the challenges of transistor variability and degradation," *Micro, IEEE*, vol. 25, no. 6, 2005.

[2] R. Baumann, "Radiation-induced soft errors in advanced semiconductor technologies," *Device and Materials Reliability, IEEE Transactions on*, vol. 5, no. 3, pp. 305–316, 2005.

[3] P. Romano, N. Carvalho, and L. Rodrigues, "Towards distributed software transactional memory systems," ser. LADIS '08.

[4] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, "The multikernel: a new os architecture for scalable multicore systems," ser. SOSP '09.

[5] R. F. van der Wijngaart, T. G. Mattson, and W. Haas, "Light-weight communications on intel's single-chip cloud computer processor," *SIGOPS Oper. Syst. Rev.*, vol. 45, no. 1.

[6] G. Chrysos and S. P. Engineer, "Intel® xeon phi coprocessor (codename knights corner)," in *Proceedings of the 24th Hot Chips Symposium, HC*, 2012.

[7] Tilera Corporation, *TILE-Gx Processor Family*, http://www.tilera.com.

[8] F. B. Schneider, *Replication management using the state-machine approach*. ACM Press/Addison-Wesley Publishing Co., 1993.

[9] B. Kemme, F. Pedone, G. Alonso, A. Schiper, and M. Wiesmann, "Using optimistic atomic broadcast in transaction processing systems," *IEEE TKDE*, vol. 15, no. 4, 2003.

[10] N. Shavit and D. Touitou, "Software transactional memory," ser. PODC '95.

[11] X. Defago, A. Schiper, and P. Urban, "Total order broadcast and multicast algorithms: Taxonomy and survey," *ACM Computing Surveys*, vol. 36, 2004.

[12] R. Palmieri, F. Quaglia, and P. Romano, "Aggro: Boosting stm replication via aggressively optimistic transaction processing," in *NCA '10*.

[13] ——, "Osare: Opportunistic speculation in actively replicated transactional systems," in *SRDS '11*.

[14] ——, "Asap: an aggressive speculative protocol for actively replicated transactional systems," in *NCA '12*.

[15] J. a. Barreto, A. Dragojevic, P. Ferreira, R. Filipe, and R. Guerraoui, "Unifying thread-level speculation and transactional memory," in *Proceedings of the 13th International Middleware Conference*, ser. Middleware '12.

[16] A. Dragojević, R. Guerraoui, and M. Kapalka, "Stretching transactional memory," ser. PLDI '09.

[17] V. Marathe, M. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. Scherer III, and M. Scott, "Lowering the overhead of nonblocking software transactional memory," in *TRANSACT*, 2006.

[18] R. Palmieri, F. Quaglia, P. Romano, and N. Carvalho, "Evaluating database-oriented replication schemes in software transactional memory systems," in *Proc. of DPDNS*, 2010.

[19] P. Dodd and L. Massengill, "Basic mechanisms and modeling of single-event upset in digital microelectronics," *Nuclear Science, IEEE Transactions on*, vol. 50, no. 3, pp. 583–602, 2003.

[20] F. Sexton, "Destructive single-event effects in semiconductor devices and ics," *Nuclear Science, IEEE Transactions on*, vol. 50, no. 3, pp. 603–621, 2003.

[21] J. Benedetto, P. Eaton, K. Avery, D. Mavis, M. Gadlage, T. Turflinger, P. Dodd, and G. Vizkelethyd, "Heavy ion-induced digital single-event transients in deep submicron processes," *Nuclear Science, IEEE Transactions on*, vol. 51, no. 6, pp. 3480–3485, 2004.

[22] G. Bruguier and J.-M. Palau, "Single particle-induced latchup," *Nuclear Science, IEEE Transactions on*, vol. 43, no. 2, pp. 522–532, 1996.

[23] S. Reinhardt and S. Mukherjee, "Transient fault detection via simultaneous multithreading," ser. ISCA '00.

[24] M. Gomaa, C. Scarbrough, T. N. Vijaykumar, and I. Pomeranz, "Transient-fault recovery for chip multiprocessors," ser. ISCA '03.

[25] S. Mukherjee, M. Kontz, and S. Reinhardt, "Detailed design and evaluation of redundant multi-threading alternatives," ser. ISCA '02.

[26] A. Shye, J. Blomstedt, T. Moseley, V. Reddi, and D. Connors, "Plr: A software approach to transient fault tolerance for multicore architectures," *Dependable and Secure Computing, IEEE Transactions on*, vol. 6, no. 2, pp. 135–148, 2009.

[27] K. Pattabiraman, Z. Kalbarczyk, and R. Iyer, "Application-based metrics for strategic placement of detectors," ser. PRDC '05.

[28] S. K. S. Hari, S. V. Adve, H. Naeimi, and P. Ramachandran, "Relyzer: exploiting application-level fault equivalence to analyze application resiliency to transient faults," ser. ASPLOS '12.

[29] K. Pattabiraman, N. Nakka, Z. Kalbarczyk, and R. Iyer, "Symplfied: Symbolic program-level fault injection and error detection framework," ser. DSN '08.

[30] L. Dalessandro, M. F. Spear, and M. L. Scott, "NOrec: Streamlining STM by Abolishing Ownership Records," in *PPoPP '10*.

[31] D. Dice, O. Shalev, and N. Shavit, "Transactional locking ii," in *DISC '06*.

[32] G. Korland, N. Shavit, and P. Felber, "Noninvasive concurrency with Java STM," in *MULTIPROG '10*.