Runtime Validation of Transactional Memory Systems

Kaiyu chen, Sharad Malik, Priyadarsan Patra

Abstract

Transactional Memory (TM) has been proposed as a promising solution to effectively harness the increasing processing power of emerging multi/many-core systems [1]. While there has been considerable research on the design and implementation of TM systems, it remains to be shown how to address the validation challenge of such systems in face of increasing design bugs and dynamic errors. This paper proposes a runtime validation methodology for ensuring the end-to-end correctness of a Transactional Memory system. We use an extended constraint graph model to capture the correctness of a transactional execution, and provide efficient hardware support to perform online checking of this constraint graph. We describe the design ideas as well as the key optimization techniques to make this approach practical. Experiments based on a state-of-the-art TM system framework show that our design effectively performs system-level runtime validation with relatively small overhead.

1. Introduction

Transactional Memory (TM) has been proposed as a promising solution to effectively harness the increasing processing power of emerging multi/many-core systems [1]. TM provides a programming abstraction where groups of instructions in a thread appear to execute *atomically* and *in isolation*. It allows the programmer to focus on writing correct sequential segments of code that are expected to execute atomically, and has the performance advantage in that serialization is only enforced when there is actual conflict. Recently there has been considerable research in the design and implementation of TM systems [2, 3, 4]. Although software TM proposals offer greater flexibility, hardware TM implementations are becoming more popular due to their performance advantage. This research focuses on hardware TM systems.

While TM simplifies the programmers' tasks, it also creates new challenges for system verification. First, it relies on complicated hardware support to maintain the desired transactional semantics. This requires solving various implementation issues involved in data version management, conflict detection, etc. In practice, the support for transactional execution often has to be built on top of existing highly complex shared-memory systems, in order to support both legacy code and new transactional applications. This also requires careful design considerations to ensure the correct interaction between transactional and non-transactional code. In recent years we have already seen a growing gap between processor design complexity and verification capability [5]. The emerging complex TM system designs will make it even more difficult to prevent bug escapes in design verification.

The verification problem is exacerbated by the fact that processor hardware is becoming more vulnerable to aging- and environment-induced hard failures and soft errors in the field [6]. Such errors cannot be eliminated through the conventional design verification process or limited local protection. For example, rising soft-errors in combinational logic, normally unprotected by ECC, can cause multi-bit flips and possibly lead to violations of the atomicity property in Transactional Memory.

To help address these challenges, we propose a novel system-level runtime validation method to ensure the end-to-end correctness of Transactional Memory. This paper makes the following contributions:

1) We develop an extended constraint model [7] to capture the correctness of transactional execution, as well as several effective graph optimization techniques to reduce the validation overhead.

2) We show how to implement the proposed runtime validation method with simple hardware support in a state-of-the-art TM system [4].

3) We evaluate our design by detailed simulation using a TM version of the SPLASH-2 parallel benchmarks adapted by us using the LogTM ISA extensions [4]. The results (i) reveal potential bugs in the baseline system and (ii) show that our solution has relatively small overhead.

The rest of the paper is organized as follows. Section 2 describes the extended constraint graph model for TM systems and the graph optimization techniques. Section 3 presents the runtime validation methodology and addresses the implementation issues. Section 4 describes the simulation framework and presents our experimental results. Section 5 discusses related work, and Section 6 provides the conclusions.

2. System model

2.1. Background

To model the correct execution of a TM program, we employ an extension of the constraint graph model [7], which has been used to reason about the correctness of conventional shared-memory multi-processor execution. A constraint graph is a directed graph whose vertices represent the dynamic memory instruction instances. The edges indicate the ordering relationships among these instructions. Specifically, the edges can be classified into the following categories.

1. Consistency edges: These edges reflect the ordering constraints placed by the memory consistency model [8] among instructions in the same processor. For example,

there is a consistency edge between any two adjacent memory instructions in the Sequential Consistency model.

2. Dependence edges: These edges represent the data dependence order among conflicting instructions (accesses to the same address), including the usual Read-after-Write (RAW), Write-after-Write (WAW), and Write-after-Read (WAR) dependences.

As shown in previous works [7], the constraint graph must be acyclic iff the multiprocessor execution satisfies the memory consistency rules. This property has been effectively exploited to examine conventional shared-memory system behavior using simulation and testing based methods [9, 10]. We describe how to extend it to model transactional execution at runtime in the next section.

2.2. Constraint graph model for TM execution

Our constraint graph extension is based on the formal specification of a practical TM system [11], which allows the program to have both transactional code and non-transactional code. The non-transactional code execution should conform to the conventional memory consistency rules, which has been modeled in previous works [7]. For clarity in this discussion, we assume that the non-transactional code conforms to Sequential Consistency.

For a complete model of the target system execution, we need to extend the constraint graph to capture additional execution constraints on the transactional code, as well as the interaction between transactional and non-transactional code. These are specified by the following axioms [11]:

TransOpOp: The memory operation order within a transaction must be consistent with the program order.

TransMembar: The transaction boundary enforces the memory barrier semantics (i.e., all preceding operations must be completed before any following operation can proceed).

TransAtomicity: A transaction must execute atomically without any other intervening operations.

The original TM specification makes the simplifying assumption that transactional code and non-transactional code access non-intersecting memory locations [11]. We relax this and assume that the system maintains strong atomicity, i.e., the transaction's atomicity and isolation properties are still guaranteed when there is a conflicting non-transactional operation.

We make the following modifications to the constraint graph to model the additional rules above. 1) For the **TransOpOp** rule, we add an edge between any two adjacent memory operations within a transaction; 2) For the **TransMembar** rule, we need to make sure that there is a path from any preceding memory operation to the transaction start instruction, and also a path from the transaction end instruction to any succeeding operation. In the case where the non-transactional code conforms to Sequential Consistency, we only need to add an edge between the transaction start/end instruction and their immediate predecessor/successor instruction, as all other edges are transitively implied; 3) For the **TransAtomicity** rule, we represent a complete transaction as a meta-vertex and connect it to all inter-processor dependence edges incident on any contained transactional operation.

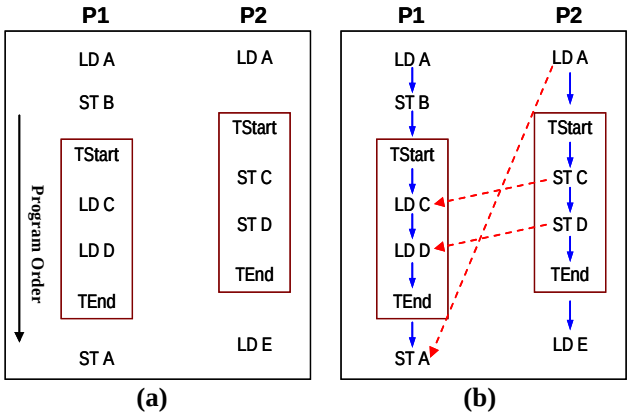

Figure 1. Constraint graph example

An example of transactional code for two processors P1 and P2 is given in Figure-1 (a). For the observed execution scenario, the extended constraint graph is shown in Figure-1(b). The consistency edges (shown in solid lines) are added among instructions in the same processor. These only depend on the semantics of the architecture's memory model, and the transitively implied edges are not shown for clarity. We can see that the added consistency edges sufficiently reflect the TransOpOp and TransMembar rules described above. The dependence edges (shown in dotted line) are added among conflicting instructions (accesses to the same address). In this case, the intra-processor dependence edges are implied by the consistency edges. The inter-processor dependence edges depend on the runtime order in which two conflicting instructions are performed from different threads. These are to be constructed using the methods described in Section 3. The resulting graph still preserves the property that it is acyclic iff the transactional execution is correct. Intuitively, from the acyclic constraint graph we can find a total order of interleaving transactions and non-transactional operations that satisfies the transactional memory semantics. For the given example, a legal total order is such that all P2's transactions/operations are performed before P1's.

2.3. Graph optimizations

A straightforward application of the extended constraint graph model is to collect all dynamic memory operations in the entire program execution and add all the consistency/dependence edges. However, this is impractical as large programs may run billions of instruction before completion and the resulting graph size will be too large to be handled. To alleviate this problem,

we apply the following graph optimization techniques to reduce the size of the graph to be checked:

Graph reduction: This reduces the number of graph vertices that we have to track for a given execution interval. The basic ideas are illustrated in Figure 2(a). We have two observations: 1) since all intra-processor edges are implied by program order, for the observed execution trace on P1 to form a cycle, there must be an incoming edge at the top and an outgoing edge at the bottom. Therefore, we can ignore vertex 1/6, as they cannot be involved in a cycle; 2) the vertex 3 and 4 can be compressed. To detect a potential cycle, we only need an edge from 2 to 5, which is effectively the transitive closure of the removed edges.

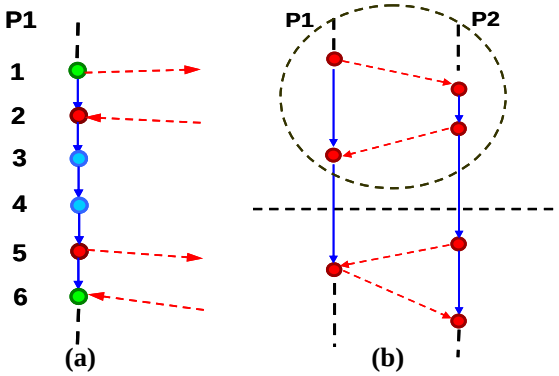

Figure 2. Constraint graph optimizations

Graph slicing: This reduces the scope of the checking. The basic idea is illustrated in Figure 2(b). Instead of waiting for the entire program to finish execution, we dynamically identify a sub-graph for the currently observed executions that can be safely checked and pruned away. As shown in the example, for the check to be valid, the graph slice boundary must satisfy the property that there are only forward edges across the boundary. Since there is no backward edge, it ensures that there is no cycle that involves graph vertices both before and after the boundary. Therefore, testing the acyclic property of the sliced graphs is equivalent to testing the complete graph.

Note that for transactional execution, we allow the slice to be in the middle of a transaction. In this case we will record the causality relationship observed between two transactions before the graph slice, and take this into account when checking the succeeding execution to detect potential violation of the atomicity property.

Combining these techniques enables us to perform practical online checking of the transactional memory system execution. We describe our design methodology in the next section.

3. Design methodology

As discussed in Section 1, a violation of Transactional Memory semantics may be caused by subtle interactions of many system components. Current studies show that the memory interface has already become the largest subclass of silicon errors reported in processor errata [12]. Adding transactional memory support will further increase the system complexity, and limit the effectiveness of existing design verification techniques and local component protection schemes. Therefore, we propose to dynamically construct and check the constraint graph, which ensures the end-to-end correctness of the observed system execution results.

Our targeted errors include any design bug/dynamic fault induced execution errors that break the Transactional Memory semantics modeled by the constraint graph. Section 4 provides a case study of the detected bug. Note that there are other system correctness aspects to be ensured for complete correctness of transactional execution, and we leverage existing techniques to guarantee those requirements, e.g., the local computation within a transaction must be performed correctly. This is well-addressed by existing uniprocessor verification techniques. In our approach, we do not explicitly check the data values. We assume the storage structures are protected by techniques such as ECC, so that for a validated total access order, the data is not corrupted between memory accesses and the correct value flow is guaranteed.

We assume the target system is a cache-coherent shared-memory multiprocessor system with transactional memory support. The system parameters for the baseline architecture are summarized in Section 4. We assume the baseline architecture perform in-order execution, which is typically modeled in current Transactional Memory system prototypes [3, 4]. We briefly address the general solution for the more aggressive implementations in the end of this section.

To enable practical implementation of the proposed runtime validation scheme, we need efficient hardware support for online tracking of the observed execution record and accelerated graph checking. We address these implementation issues as follows.

Consistency edge construction: To support efficient construction of consistency edges, we augment each processor pipeline with a counter in the dispatch stage. When each dynamic memory instruction is dispatched in program order, it is assigned a monotonically increasing number called the Memory Instruction Identifier (MID). Since the consistency edges are implied by program order, given any two memory operations, we can determine their consistency edge by comparing their MIDs.

To support the meta-vertex scheme used to model the TransAtomicity rule as described in Section 2.2, we simply assign the same MID to all instructions dispatched between transaction start/end instructions, which are typically architecture specific ISA extensions. The MID is recycled if a transaction gets aborted. The wraparound of

MID is handled by stalling the processor until all its outstanding memory operations are retired and validated at a graph slice boundary.

Note that based on the graph reduction technique, we do not need to explicitly store the locally observed consistency edges. Instead, we only have to keep the vertices with inter-processor dependence edges, and the transitive closure of intra-processor edges among those vertices are implicitly encoded by their MIDs. To save the local hardware storage overhead, we will re-construct them at the central graph checker as described below.

Dependence edge construction: Since we perform online checking, we need low overhead dynamic recording schemes for the inter-processor dependence edges. We achieve this by exploiting the fact that each type of these edges results in different cache coherence activities: 1). A RAW edge corresponds to a read miss, and involves transferring the data block modified by the writer to the reader's local cache. 2). A WAW edge corresponds to a write miss, and also involves updating the second writer's local cache with the modified data block. 3). A WAR edge corresponds to an upgrade of the cache access permission if the second writer already has the data block in shared state, or a write miss otherwise.

This allows us to piggyback on the cache coherence transactions associated with these events to construct the corresponding inter-processor dependence edges. Similar dependence tracking mechanisms have been used in previous works in the area of deterministic replay and race detection [13]. The basic idea is that we first augment each cache block to record the MID of the last local load/store instruction that accessed this block. Then the cache controller piggybacks this information to a new coherence message. When the receiving processor sees this enhanced message, it can construct the corresponding inter-processor edge by looking up the piggybacked information and its own local access history. Due to the limited space, here we omit the detailed discussion on design issues and protection schemes for the required verification messages. These are addressed in a separate technical report [14].

While there are subtle differences in various cache coherence protocols, we piggyback on a common subset of the coherence transactions to observe the dependence edges, e.g., the invalidation message or data reply message that must be generated in case of a write/read miss. The constructed edges reflect when the data is actually sent/received by different processor nodes. We store the list of observed inter-processor edges in a local hardware buffer at each processor. We made this design decision because in a practical system, there is no central point that allows us to observe all distributed events, and each processor may only observe a partial subset of the dependence edges. On the other hand, we must examine the complete graph to verify the global system behavior, as described below.

Central graph checking: The locally observed edges are periodically transferred to a central checker to build the complete graph and perform the checking. To avoid performance penalty, the graph checker operates in parallel with the usual computation and checkpointing process. Our experiment shows that the actual constraint graph is fairly sparse, with the number of edges is linear to the number of vertices. Based on this observation, we use an adjacency-list (edge-list) representation for the graph and use a dedicated hardware engine to check for cycles using depth-first search (DFS).

Here we provide a brief description of the pipelined design to establish the viability of implementation. The checker consists of three pipeline stages: 1) Collect the list of locally observed edges from each processor 2) Construct the global constraint graph. The transitive closure of the intra-processor edges is inferred and added to the constraint graph. 3) DFS graph traversal using a dedicated hardware engine. This has complexity O(E), where E is the number of the graph edges. Details on straightforward implementations are omitted for brevity, e.g. hardware linked list to represent the edge list and a state machine for a DFS-based cycle checker.

Validation protocol: Based on the graph slicing method described in section 2.3, we employ a three phase protocol for the runtime validation:

1) All processors coordinate to establish a graph slice boundary based on a loosely synchronized physical clock maintained on the target system. Such a clock has been conveniently used in previous multiprocessor research work [15]. It is relatively easy to implement, as long as it satisfies the requirement that the clock skew is less than the traversal time between any two processors. Using this as a common logical time base, all processors send the locally collected edges for currently retired instructions at the end of a fixed validation time interval. The validity of the resulting graph slice boundary is ensured by the fact that the processor performs in-order execution; therefore there can be no back-edge from future instructions to already retired instructions.

2) The central checker collects the received local edge lists, then constructs the global constraint graph and verifies the acyclic property.

3) Each local processor receives the acknowledgement from the central checker and gets notified if there is an error detected in previous execution.

If an error is detected, we leverage the commonly used check-pointing schemes to resume the execution from a previously validated state [15].The computation may be resumed with a less aggressive concurrent memory access mode (e.g., by temporarily serializing the execution) to avoid the recurrence of the error.

Finally, we note that in the above discussion we have made the assumption that the non-transactional code obeys Sequential Consistency and the baseline architecture employs in-order execution. While the same validation methodology still applies for relaxed memory models and out-of-order execution, we need to consider additional design complexities. There are also design issues related to cache false-sharing and eviction due to finite cache. We have developed a general solution to address those problems as described in a separate technical report [14]. Due to limited space, the details are omitted here as they are not the focus of this paper.

4. Experimental results

Simulation framework: Our experimental evaluation is based on the recently proposed LogTM infrastructure [4]. Table-1 summarizes the baseline system parameters. We extended the baseline architecture with the proposed hardware support for runtime validation, and ran simulation using modified SPLASH-2 benchmarks [16] having both transactional and non-transactional code. The results are reported below (the benchmark *radiosity* is excluded as its simulation took too long to finish).

Tabel-1. Baseline system configuration

Processors	Two single-issue, in-order, SPARC V9 processor model Single cycle non-memory latency
L1 Cache	16KB 4-way split, 1 cycle latency
L2 Cache	4 MB 4-way unified, 12-cycle
Memory	1G bytes, 80 cycle latency
Coherence Protocol	MESI_SMP_Directory
Interconnection Network	Hierarchical_Switch

Constraint graph evaluation: Figures 3 and 4 show the maximum number of graph vertices/edges used in the constructed constraint graphs; the simulation is performed for different lengths of validation interval.

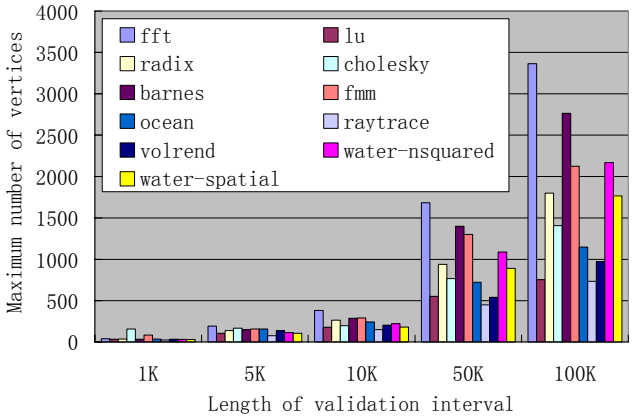

Figure 3. Maximum number of graph vertices

We make the following observations: 1) Although the processor model has an aggressive single-cycle non-memory IPC, the number of graph vertices is much smaller than the number of executed instructions. Compared with a naively built graph for the complete execution trace, the graph optimization allows us to achieve orders of magnitude reduction of the graph size. In addition, we observed that the average number of vertices/edges is much smaller than the maximum number, which means that in most validation intervals we incur small overhead in the graph checking. 2) In general, the checking is performed most cost-effectively at the 10K-cycle interval. It offers a good trade-off between the checking latency and the required hardware overhead. 3) By comparing the number of graph edges and vertices in the same interval, we can see that the graph is fairly sparse and well-suited for the described graph checker design in Section 3.

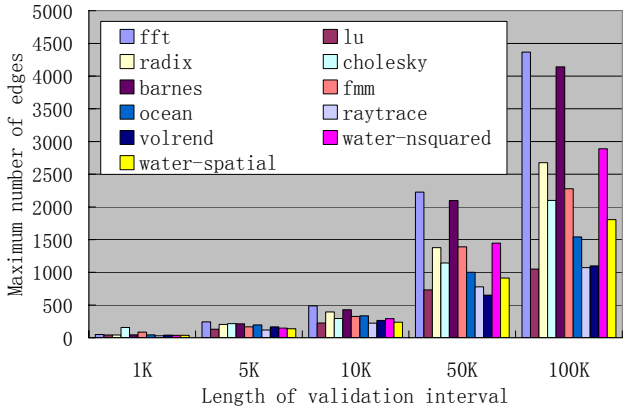

Figure 4. Maximum number of graph edges

Communication overhead: To evaluate the communication overhead due to the augmented verification messages in our scheme, we report the increase in total traffic size in Figure 5. The average traffic overhead is 12%, which is a reasonable tradeoff for improved reliability. We can further reduce this overhead by using a more compact representation of the global instruction ID. The details are omitted here for brevity.

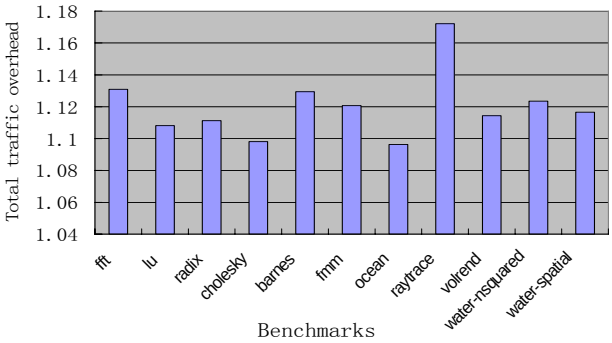

Figure 5. Total traffic size overhead

Performance analysis: The error detection latency is effectively hidden in the proposed validation protocol, as the graph checking is done in a pipelined fashion and in parallel to the normal computation and/or check-pointing.

At the typical 10K-cycle validation interval, the reduced graph has only a few hundred edges, whil allowing the checker nearly 10K cycles to process without stalling.

Scalability study: To evaluate the scalability of the proposed method, we have also run simulations with 4 processor and 8 processor configurations. Overall, we observed that with increasing processors, the number of vertices appear to grow linearly, and the resulting graph is still fairly sparse. For example, for benchmark *fft*, the average number of graph vertices /edges measured at 10K-cycle validation interval is 86/105 with 2-processor, 177/216 for 4-processor, and 300/366 for 8-processor configuration. For *barnes*, the numbers are 16/18, 45/55, and 100/127 respectively. Therefore, we expect the proposed method to still work well in scalable designs.

Error detection case study: When applying the proposed validation method to the baseline system, we detect a cycle [0, 92242]→[1, 241617]→[0, 92242] in running the benchmark *fmm*. This error case is depicted in Figure 6. Upon further investigation, we found that the node [0, 92242] corresponds to a transaction in P1, which contains an atomic read-modify-write operation and a load operation to the same address. [1, 241617] corresponds to a conflicting atomic operation performed by P2. The problem occurs because the atomic operation in P1 does not set the m_Write bit in the transaction cache. As a result, the system allows P2's operation to intervene in the transaction execution, instead of sending a NACK to P2. This is an example where the transaction atomicity is broken by a possible design flaw. It is clear that similar problems may be caused by dynamic faults in the system. We are performing further experiments on error injection, e.g., allowing multiple outstanding operations to be completed out-of-order.

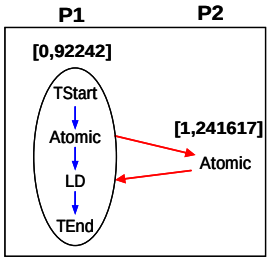

Figure 6. A detected error

5. Related works

Manovit et al. have proposed a testing methodology for TM systems based on their formal specification [11]. However, this testing approach has the following limitations: 1) it relies on running special test programs to expose the bugs, and then performs offline analysis for the collected execution trace. This method is not applicable to large real programs. 2) It relies on data coloring (i.e., each store writes a unique value) to establish the RAW edges. To infer other necessary edges, it must perform an expensive recursive operation, which is not suitable for online checking. Recently there has been other reported work on runtime checking of the uni-processor / multiprocessor systems [17, 18]. However, there has been little work on runtime validation of Transaction Memory systems as addressed in this paper.

6. Conclusions

We present a runtime validation method to address some pre-silicon as well as dynamic validation challenges in emerging transactional memory based multi/many-core systems. Based on an extended constraint graph model for capturing transactional memory semantics, we show how to perform effective online checking of the system execution with key enabling optimization techniques and efficient hardware support. The experiment based on a state-of-the-art TM system shows promising results. Our solution provides a seamless validation framework for both conventional, non-transactional code and new transactional applications, which ensures the end-to-end correctness of the system execution and can be effective against both design bugs and dynamic errors.

7. References

[1] M. Herlihy, J. E. B. Moss, "Transactional memory: Architectural support for lock-free data structures", International Symposium on Computer Architecture (ISCA), 1993.
[2] N. Shavit and D. Touitou, "Software transactional memory", 14th Symposium on Principles of Distributed Computing, 1995.
[3] L. Hammond, et al. "Transactional memory coherence and consistency", Intl. Symp. on Computer Architecture, 2004.
[4] K. E. Moore, et al. "LogTM: Log-based transactional memory", 12th International Symposium on High-Performance Computer Architecture (HPCA), 2006.
[5] B. Bailey, "A New Vision for Scalable Verification", EE Times, March 18, 2004.
[6] S. S. Mukherjee, J. Emer, and S. K. Reinhardt, "The Soft Error Problem: An Architectural Perspective", HPCA, 2005.
[7] H. W. Cain, et al. "Constraint Graph Analysis of Multithreaded Programs", Journal of Instruction-Level Parallelism, vol. 6, April 2004.
[8] S. V. Adve and K. Gharachorloo, "Shared Memory Consistency Models: A tutorial", IEEE Computer, vol. 29, 1996.
[9] S. A. Taylor, et al. "A Simulation-Based Method for the Verification of Shared Memory in Multiprocessor Systems", International Conference on Computer-Aided Design, 2001.
[10] S. Hangal, et al. "TSOtool: A Program for Verifying Memory Systems Using the Memory Consistency Model". 31st International Symposium on Computer Architecture, 2004.
[11] C. Manovit, et al. "Testing implementations of transactional memory", International conference on Parallel architectures and compilation techniques (PACT), 2006
[12] S. Sarangi, et al. "Patching Processor Design Errors with Programmable Hardware", IEEE Micro 27, 1, Jan. 2007.
[13] M. Xu, et al. "A Flight Data Recorder for Enabling Full-system Multiprocessor Deterministic Replay", ISCA, 2003.
[14] Reference removed for blind review.
[15] D. J. Sorin, et al. "SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint /Recovery", Intl. Symposium on Computer Architecture, 2002.
[16] S. C. Woo, et al., "The SPLASH-2 Programs: Characterization and Methodological Considerations", International Symposium on Computer Architecture, 1995.

[17] T. M. Austin, “DIVA: A Reliable Substrate for Deep-submicron Microarchitecture Design”, 32nd Annual International Symposium on Microarchitecture, 1999.
[18] A. Meixner and D. J. Sorin. "Dynamic Verification of Memory Consistency in Cache-Coherent Multithreaded Computer Architectures." International Conference on Dependable Systems and Net-works, June 2006.