Runtime Validation of Memory Ordering Using Constraint Graph Checking

Kaiyu Chen, Sharad Malik and Priyadarsan Patra*
Dept. of Electrical Engineering, Princeton University **Intel Validation Research Lab*
{kchen, sharad}@princeton.edu *priyadarsan.patra@intel.com*

Abstract

An important correctness issue for emerging multi/many-core shared memory systems is to ensure that the inter-processor communication through shared memory conforms to the memory ordering rules, as specified by the architecture's memory consistency model [1]. This presents a significant validation challenge. Growing system complexity makes it increasingly hard to identify all deep-state logic bugs in pre-silicon verification. Further, aggressive technology scaling makes hardware more vulnerable to dynamic errors that can only be detected at runtime.

In this paper, we propose an approach for runtime validation of memory ordering. This allows us to survive bugs that escape pre-silicon verification, as well as deal with emerging dynamic errors. Our solution consists of two parts: 1) at the microarchitecture level, we add efficient hardware support to capture the observed ordering among shared-memory operations; 2) we perform online verification of the observed memory ordering by checking for cycles in the constraint graph [11, 12]. We combine these to achieve end-to-end correctness validation of the system execution with respect to the memory ordering specification. There are several challenges that need to be addressed to make this approach practical. We describe these, as well as optimization techniques for reducing the hardware overhead. Estimates obtained from preliminary chip multiprocessor simulation experiments show that the proposed techniques are very effective in achieving acceptable hardware overhead and minimal performance impact.

1. Introduction

Validation challenges for memory ordering: On emerging multi/many-core shared memory systems, the memory accesses must conform to the memory ordering rules specified in the architecture's memory consistency model [1]. Maintaining the correct memory ordering in a shared-memory system requires careful consideration of numerous issues in designing system components and their interfaces, which imposes a significant validation challenge. An indicator of this complexity is that memory interface is the largest subclass of silicon errors reported in processor errata [2]. In addition to possible failures due to design errors, due to aggressive technology scaling, processors are becoming more vulnerable to dynamic faults resulting from thermal conditions, aging, or particle hits [3].

Limitations of existing techniques: There has been substantial previous research on the problem of verifying shared memory systems. Formal verification techniques such as model checking have been used to assist the analysis of memory protocols [4]. However, this has limited practical success due to the inherent intractability of checking memory ordering [5]. In practice, designers rely on simulation and testing based methods to examine the memory system behavior [6, 7]. However, with these methods the coverage is limited because it is hard for the test program to exercise all possible runtime behaviors. Both the formal verification and simulation methods are ineffective in detecting possible dynamic errors that occur post-deployment.

Existing systems have also employed other techniques to improve reliability, such as using specialized techniques for protecting local components. However, these are ad-hoc and expensive to adequately protect the various system components against operation-time errors.

Our contributions: In this paper, we propose an approach for runtime validation of memory ordering, which allows us to overcome the above limitations and provide a runtime guarantee of the correctness of the actual system behavior. Our solution consists of two parts: 1) at the micro-architectural level, we add efficient hardware support at each processor node and the cache controller to capture the ordering among shared-memory operations; 2) we perform online verification of observed architectural results by checking for cycles in the constraint graph [11, 12, 13] that represents the memory ordering rules. This ensures the end-to-end correctness of the memory operations.

A straightforward implementation of this scheme would track all executed memory instructions and their dynamic ordering relationships. However, there are significant challenges to make this approach practical.

978-1-4244-2070-4/08/$25.00 ©2008 IEEE

1. The first is to bound the scope of the checking. In theory, the size of the cycle in the constraint graph may be unbounded. Thus, the check seems possible only after the program completes execution. Checking the entire program execution as done in offline analysis has limited value because real applications may run billions of instructions before completion. Ideally we would like to check short execution intervals of the program and recover from errors promptly.
2. The second arises from the size of the constraint graph. Even if we could overcome the first challenge and check short program execution intervals, each memory operation would result in a vertex in the constraint graph. Due to the high instruction execution rate in modern processors, including all executed memory instructions in the constraint graph results in very large storage requirement and latency for cycle checking, even for relatively short execution intervals.

We describe techniques to address these challenges and to allow error recovery using check-pointing schemes (e.g., [9, 10]). We evaluate our design through simulation of a chip multiprocessor (CMP) system with selected parallel programs from the SPLASH2 benchmark suite [24].

The rest of the paper is organized as follows. Section 2 provides some background for this work. Section 3 presents the overall runtime validation approach. Section 4 addresses the implementation issues and design optimization techniques. The experimental results are presented in Section 5. Section 6 discusses the related work, and Section 7 provides some conclusions. Additional details on design issues and experimental results, omitted here due to lack of space, are available in an extended technical report [30].

2. Background

2.1. Memory models

The most intuitive memory ordering model is Sequential Consistency (SC). SC requires that all memory operations appear to execute in a total order, where instructions from the same processor must follow the corresponding program order. For conventional shared memory multiprocessors, a straightforward implementation of the SC model would serialize all memory references and preclude high-performance optimizations such as out of order execution and memory bypassing or forwarding. To improve the performance, more relaxed memory ordering models have been proposed. The basic idea is to allow two memory operations to be performed out of program order, so that the subsequent instructions do not have to wait for a stalled memory operation to complete before they can be processed. For example, Total Store Ordering (TSO) relaxes the store-to-load ordering. As one of the most relaxed memory models, Weak Ordering (WO) imposes no ordering constraint between two memory operations on the same processor. When needed in programming, memory barrier (MB)/fence instructions are provided to enforce the ordering between preceding and subsequent memory operations. A categorization of the memory ordering relaxation in existing memory models is provided in the survey by Adve and Gharachorloo [1].

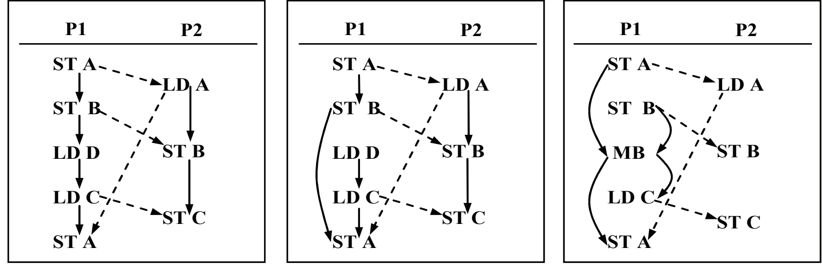
2.2 Constraint graph models

Given the ordering rules specified in different memory models, an effective method for reasoning about the correctness of multiprocessor execution is to use the constraint graph [11, 12, 13]. A constraint graph is a directed graph whose vertices represent the dynamic instruction instances in program execution. (In this paper we use instruction synonymously with operation. In practice an instruction can involve multiple memory operations, in which case we will use separate graph vertices for each operation.) 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 model among instructions in the same processor. For example, there is a consistency edge between any two adjacent memory instructions in the SC model, while the edge from a store to a load is relaxed in the TSO model [1].
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. These may be intra-processor or inter-processor edges.

Figure 1 shows examples of constraint graphs for different memory models. We denote the consistency edges by solid lines, and the dependence edges by dotted lines. The ordering relation is transitive, and redundant edges are not shown for clarity. As has been shown in previous work, this graph is acyclic iff the parallel execution satisfies the memory ordering rules. Thus this acyclic property can be used to detect a memory ordering error.

This graph checking scheme has been effectively used in industry in testing and simulation based verification. For example, it has been applied for checking Alpha's Weak Ordering (WO) model [6], Sun's TSO model [7] and Intel's Itanium [8]. However, it has a limitation due to the lack of modeling for write non-atomicity. On architectures that allow the writes to be visible in different orders to different processors (e.g., IA-32), we may get a false-positive error (where a constraint graph cycle is introduced by a store's

inconsistent visibility orders on different processors). Such false-positive errors will incur additional performance penalty due to unnecessary error handling, but will not affect the execution correctness. Even though TSO does not have write atomicity, previous work has shown the false-positive errors can be avoided by not including the intra-processor store-to-load dependence edge in the global constraint graph [15].



(a) Sequential Consistency (b) Total Store Ordering (c) Weak Ordering

Figure 1: Constraint Graph Examples

2.3 Targeted errors and design assumptions

We target multiprocessor execution errors that violate the memory ordering model. These may be caused by various design bugs or dynamic faults in any shared-memory system component [6, 7, 8], e.g., a micro-architecture design flaw that allows a memory operation to be performed too early or a soft-error in the interconnection network that causes two messages to be incorrectly re-ordered. Our design builds on other well-established techniques to address additional requirements for correct operation of a multi-core shared-memory system. We make the following assumptions about the target architecture.

1. Similar to other work in the context of memory ordering verification [6, 7, 8, 13] we are only concerned with the memory ordering issues. To ensure the complete correctness of the multithreaded execution, one would also have to make sure the data/control flow dependences among intra-processor ALU/branch instructions are preserved. We assume these are enforced by local schemes at each processor core. In practice, uniprocessor verification has been well-addressed in both pre- and post-silicon verification. Moreover, recently dynamic methods such as DIVA [14] have been proposed to improve uniprocessor reliability.

2. We assume that the target system is cache coherent. Most modern multiprocessor systems support this with a hardware cache coherence protocol. In some of the literature, cache coherence is confused with memory consistency, but strictly speaking they are different issues [1, 22]. The cache coherence problem per se is less challenging, as it is only concerned with the access ordering of *a single location*. In pre/post-silicon verification, techniques such as random test generation using false sharing and action/check pairs have been developed to effectively address this problem [16]. Several dynamic verification techniques have also been proposed to ensure cache coherence at runtime, by using a validation protocol or checking the system wide invariants (e.g., [17, 18]). We also assume that the cache/memory is protected by techniques such as ECC, so that for a validated memory access order, the data is not corrupted between memory accesses and the correct value flow is guaranteed.

3. Runtime validation method

Due to subtle interactions among many complex system components, a violation of the memory ordering model may be due to various reasons, and it is tedious and expensive to examine each individual component to detect the error. Further, local observation of memory operations may not accurately reflect their global behavior, e.g., two memory operations performed locally in program order may be perceived as out-of-order by remote processors. These factors motivate us to perform global checking of the memory ordering constraints.

To accurately capture the global behavior, we propose to dynamically construct and check the constraint graph on-the-fly. Naively, we should collect all executed memory instructions, add the consistency and dependence edges among them as observed at runtime, then build the constraint graph and check for a cycle. However, as pointed out in Section 1, there are two major problems that render this straightforward implementation impractical. First, the size of the cycle may be unbounded. Thus, the check seems possible only at the end of program execution, which is infeasible for large programs. Second, we need to reduce the graph size. Even for relatively short execution intervals, due to the high instruction execution rate in modern processors, if we include all executed memory instructions, it will still result in very large storage requirement and latency for cycle checking.

To enable the proposed runtime validation approach, it is essential to solve these problems and come up with low runtime/hardware overhead graph construction and checking schemes. The first challenge is addressed by a periodic graph slicing technique described in Section 4.4. In this section, we address the second challenge, i.e., reducing the graph size.

Constraint graph reduction: We propose to use an equivalent but significantly reduced graph for detecting the existence of cycles. Specifically, we show that for SC and TSO we only need to include instructions with inter-processor dependence edges. For WO, we also need to include the memory barrier instructions. We show the correctness of this technique as follows: for each considered memory model (SC/TSO/WO), we enumerate the types of consistency edges and construct the equivalent reduced graph for detecting cycles. Formal proofs are omitted here for brevity (available in the extended version [30]).

Algorithm and Proof sketch: Let each memory access have a unique identifier <P, ID>, which indicates the processor number P and program order ID. Let <p, i1> and <p, i2> be two successive memory accesses that have observed inter-processor dependence edges for processor p, with i1 < i2. There may be other memory accesses <p, i> with i1< i < i2, however we now show that we do need not include them and the associated edges. In the reduced graph, we only need to keep the vertices with inter-processor dependence edges (e.g., <p, i1> and <p, i2>) and can derive the necessary local edges among them as follows:

1. For SC: Since there is a consistency edge between any two consecutive memory accesses in the SC model, in the reduced graph, we add an edge from <p, i1> to <p, i2> which is the transitive closure of all consistency edges and intra-processor dependence edges between <p, i1> and <p, i2>.

2. For TSO: In TSO the consistency edge from a ST to a succeeding LD operation is relaxed. Arvind and Maessen show that the intra-processor dependence edge between a ST to a succeeding LD operation to the same address also need to be excluded in the global constraint graph [15]. Thus, the intra-processor edges between memory operations are those captured by the patterns in Figure 2: (a) gives the canonical form of a general instruction sequence, where an oval in the figure denotes a group of consecutive LDs/STs; (b) summarizes the 5 cases of possible intra-processor edges. This enables us to determine the transitive closure of the intra-processor edges as follows. If <p, i2> is a store (case 2&3) or <p, i1> is a load (case 1&3), then there is an edge in the reduced graph from <p, i1> to <p, i2>. In addition, if <p, i1> is a load and <p, i2> is a store (case 4), we will also need to add an edge from <p, i1> to the first load instruction after <p, i2> that appears in the observed inter-processor dependence edges. Similarly, if <p, i1> is a store and <p, i2> is a load (case 5), then we will also need to add an edge from <p, i1> to the first store instruction after <p, i2> that appears in the observed inter-processor dependence edges. We note that for the "naïve" constraint graph modeling of TSO [15], where the intra-processor dependence edge from a ST to a succeeding LD to the same address is included in the constraint graph, this intra-processor dependence edge will need to be stored locally at an additional overhead. With this additional information the reduced graph can be constructed similarly.

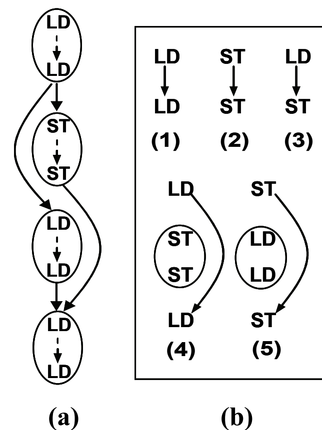


Figure 2: Intra-Processor Edges in TSO Model

3. For WO: If there is a memory barrier instruction, <p, m> between <p, i1> and <p, i2>, we add an edge in the reduced graph between <p, i1> and <p, m> and also between <p, m> and <p, i2>.This follows directly from the semantics of the barrier instruction.

Our experiments show that the reduced graph has orders of magnitude fewer vertices than the naively built graph for the complete instruction execution trace. This is critical to make this approach practical. This reduction benefits from the fact that real applications are often optimized to reduce inter-processor interaction through shared data for performance reasons. In the case that different processors run heterogeneous applications that have no inter-processor dependences, no global constraint graph needs to be built and checked.

4. Hardware design and implementation

4.1 Design overview

The overall system architecture is shown in Figure 3. For clarity in this discussion, we assume our baseline architecture is a single-chip CMP system with the SC model. The hardware support for runtime validation consists of the following components:

1) We augment each processor pipeline to assign a monotonically increasing Memory Instruction Identifier

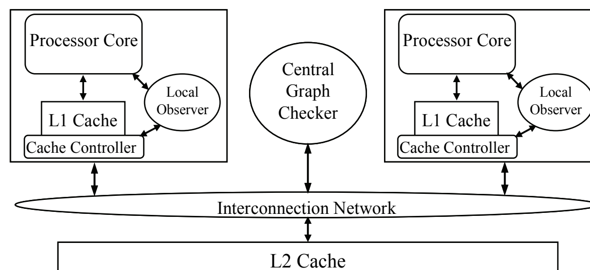


Figure 3: Overall System Architecture

(MID) to each dynamic memory instruction when it is dispatched. Since instructions are dispatched in program order, given any two memory instructions, we can determine their relative order in the program by comparing their MIDs. The wraparound of MID can be handled by stalling the processor until all its outstanding memory operations are retired and validated at a time established by the method described in Section 4.4.

2) We add additional hardware to the cache structure as follows. (Their functionality is described in Section 4.2.) We add additional fields to L1 cache blocks to record the local memory access history and augment the L1 cache controller logic to record the locally observed inter-processor dependence edges. To cope with cache eviction and message forwarding in directory-based coherence protocols, we add a small fully associative cache at L2 to keep the access record of evicted L1 data blocks, and also augment the L2 cache directory/controller to pass the message sender's identity to forwarded coherence request and acknowledgement messages.

3) We store the dependence edges observed by each processor in a local hardware buffer. The locally collected information is sent to a central graph checker periodically to test the acyclic property. The operation of the central graph checker is described in Section 4.3. If an error is detected, the central checker notifies all processor nodes to invoke the error recovery mechanisms described in Section 4.5.

To effectively hide the runtime validation latency, the constraint graph construction and checking is performed in parallel with the normal computation and check-pointing process. Figure 4 shows the timing diagram (Gantt Chart) of our proposed scheme. Using the constraint graph slicing technique proposed in Section 4.4, we can perform checking for short program execution intervals. When the parallel program chunk1 is executed during the time interval [T0, T1], the edges for the resulting constraint graph k are observed locally at each processor. During the next interval [T1, T2], while the program chunk2 continues execution and the resulting constraint graph k+1 is observed, the constraint graph k is built at the central checker. During the interval [T2, T3], the constraint graph k is examined by the central checker. Suppose we find a cycle in graph k+1 at T4, then there is an error detected and we resume the system execution from the last check-point created at T1.

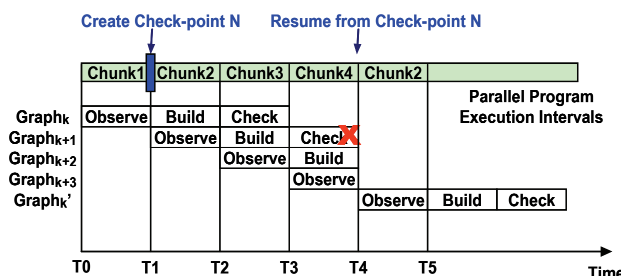


Figure 4: Validation Timing Diagram

4.2 Constraint graph edge construction

In constructing the inter-processor edges, we exploit the fact that each type of edge results in different cache coherence events:

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.

Therefore, we can piggyback on the cache coherence transactions associated with these events to achieve low performance overhead in constructing the inter-processor dependence edges. Similar dependence tracking mechanisms have been used in previous works in the area of deterministic replay and race detection [25, 26, 27]. 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 when it generates a new coherence message, which is augmented to reflect a global ID (consisting of a tuple <PID (Processor Identifier), MID>) of the instruction involved in the coherence activity. When the receiving processor sees this message, it can construct the corresponding inter-processor dependence edge by looking up the piggybacked information and its own local access history. For example, in Figure 5(b), when "ST Y" is performed on P2, we will piggyback its ID <2, 2> to the generated invalidation message. When this message reaches P1, the observer at P1 can then construct the dependence edge <1, 4>→<2, 2>. One can guarantee the correct delivery of the augmented coherence messages by using ECC-like or residue-code techniques, and prevent dropped messages by assigning consecutive IDs to each message or by enforcing a request time-out scheme [19].

Nevertheless, there are additional complexities that need to be addressed in a practical cache design:

1) <u>False sharing:</u> False sharing occurs when two processors reference different data items within the same cache line. This may create a false dependence edge if we only use the cache line address to identify conflicting accesses. If this edge is involved in a cycle, we will get a false positive error, which does not affect

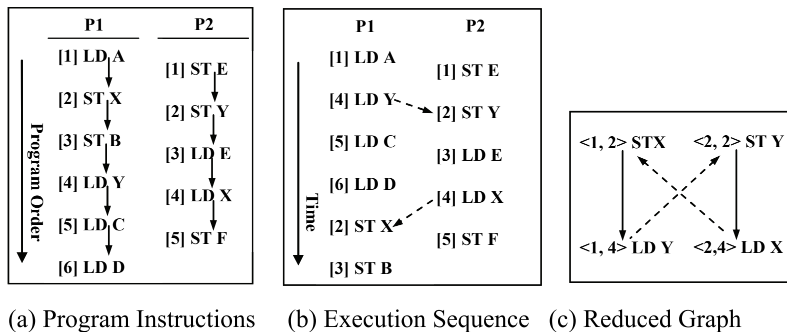
correctness but incurs a performance penalty. To avoid this, we can augment the coherence message with the offset of the actual address of the requested data in the cache line, and track the dependence at a finer data granularity level. This involves a performance vs. storage tradeoff.

2) Cache eviction: When a data block is evicted from a processor's local cache due to a conflict, the associated local access history is lost and we may not be able to construct the corresponding edge for a later coherence request. To address this problem, we add a small fully associative "evicted" cache to keep the access record of the evicted L1 data blocks from each processor. When this evicted cache is full, we will need to stall the pipeline until the validation is done for previous execution, which causes a performance penalty. To avoid such stalls, we use several optimization techniques to reduce the evicted cache size by filtering and recycling the entries. These are omitted here for brevity (available in the extended version [30]).

Using the above scheme, each processor core collects a list of the locally observed inter-processor dependence edges and records them in a hardware buffer. Each entry in the buffer contains the local instruction MID and its type (e.g., LD or ST), the remote instruction's global ID, plus additional fields to denote the edge attributes (e.g., incoming or outgoing). As described in Section 3, we can construct the transitive closure of the intra-processor edges according to the specific memory ordering rules. To save the hardware and communication bandwidth overhead, we do not explicitly store this transitive closure at each processor, but construct it at the central checker. For the example shown in Figure 5, the edge <1,2>→<1,4> will be added for P1's instructions. The resulting constraint graph is shown in Figure 5(c). We can see that in this case the execution violates the SC ordering, and the resulting cycle is detected using the following method.

4.3 Constraint graph checking

After each processor has collected the locally observed edges, the records are periodically transferred to a central checker to build the complete graph and perform the checking. To speed up this process, the record transfer and global graph construction is done on-the-fly using dedicated point-to-point links to the central checker. This is feasible as the communication between the local nodes and the central checker does not involve complicated arbitration schemes. If no such link is available, we can utilize the existing inter-connect network when there is available network bandwidth.



(a) Program Instructions (b) Execution Sequence (c) Reduced Graph

Figure 5: SC Constraint Graph Construction Example

To come up with efficient design for the central graph checker, we measured the actual constraint graph size using simulation of the selected SPLASH2 benchmarks. Our experiment shows that the graph is fairly sparse, with the number of edges linear in the number of vertices. Thus, 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). This method 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. As shown in the experiment section, the central checker has more than enough time to finish its operation during the typical validation interval.

4.4 Constraint graph slicing

Although the constraint graph reduction method effectively reduces the graph size by several orders of magnitude, for a real application that may contain billions of instructions, it is still too expensive or impossible to construct/check the complete constraint graph online. To support effective online checking, it is desirable to limit the length of the execution interval that needs to be tracked continuously. For this we need to know when an observed sub-graph can be checked for cycles and pruned away safely without affecting the correctness of future validations as the program execution continues. This simplifies the hardware design for the graph checker, reduces the checking latency and supports prompt error detection/recovery. It also enables us to de-allocate stale records in hardware resources such as the edge buffer and the evicted cache.

To tackle this problem, we propose a dynamic graph slicing technique, as illustrated in Figure 6. The motivation is that a cycle in the constraint graph is potentially caused by instruction reordering in a local processor, cache hierarchy or interconnection network. Intuitively, given the limited instruction reorder window

size and message traversal time in practical systems, it is unlikely to have an unbounded cycle in real execution. Further, we exploit the following observation: if a sub-graph in the constraint graph can be identified to not have an incoming edge from subsequent instructions, this sub-graph can be pruned away since it can never participate in a cycle that involves instructions executed in the future. This forms the basic idea of our graph slicing technique described here.

To be able to reason about the execution states of different processors based on a common logical time base, we assume that a loosely synchronized physical clock is available on the target system. This has been conveniently used in previous multiprocessor research work [9] and is relatively easy to implement on a CMP system. We also developed alternative techniques that relax this assumption. These are omitted here for brevity (available in the extended version [30]).

We perform the proposed validation at the end of fixed logical time intervals. Let us consider the case illustrated in Figure 6, which shows the execution trace of the two processors at the end of a validation interval T. Processor 1 has retired instructions up to instruction 10. Instruction 11 is not executed yet, while instruction 12 is executed out-of-order before instruction 11. Processor 2 has retired instructions up to instruction 11, and we have observed a WAR edge from <1, 12> to <2, 10>. *Our goal is to identify a boundary in the retired instructions, such that the constraint sub-graph observed before the boundary can be safely validated and removed from future checking.* In this case there can be only forward directed edges from instructions before the boundary to instructions after the boundary. Since such a forward edge represents the "happens before" causal relationship in a parallel system [20], we call this boundary the Forward Causality Frontier (FCF). The key property of FCF is that *there is no back-edge across the boundary, so that it is not possible to have a cycle that contains both instructions before and after the boundary*. The FCF for the given example is shown in Figure 6. We need to exclude instruction <2, 10>, since there is a back edge from <1, 12> to it. In fact, we can see that as time moves forward, there may be another WAR edge from <2, 11> to <1, 11>, which will form a cycle and violate the memory ordering model.

A key observation for us to identify an FCF is that the instructions at the FCF boundary are the oldest retired instructions that are reachable from any unretired instruction. By "retire", we mean the instruction has left the local write buffer and has been globally performed (the coherence transactions generated by the read/write request should have all been acknowledged). Based on this, for the example shown in Figure 6, we can efficiently implement a dynamic graph slicing protocol as follows:

1) At the end of a validation interval T, each processor Pi sends the following information to the central checker: a) the ID of the oldest unretired instruction at Pi (where all instructions before it must have been retired); b) the list of locally observed edges in the order sorted by the IDs of Pi's local instructions that appear in the recorded edge list.

2) The central checker collects the records reported by all the processors and computes the FCF as follows: for each processor Pi, it scans the reported edge list and constructs a vector ID that contains an instruction ID for each processor Pj ($j \neq i$), which reflects the minimal ID among all Pj's instructions that are connected by inter-processor edges originated from Pi's unretired instructions. For simplicity, we can set the entry for Pi itself in the vector as the ID of Pi's oldest unretired instruction. For the example shown in Figure 6, at the end of interval T, the vector ID for P1 is [11, 10], and the vector ID for P2 is [10, 12]. It then calculates the point-wise minimum of all vector IDs, which should be the corresponding FCF. For the example shown in Figure 6, the FCF is calculated as [10, 10]. In this case the sub-graph formed by instructions before instruction 10 on P1 and instruction 10 on P2 can be reduced.

3) The central checker validates the acyclic property of the identified sub-graph. If no error is detected, the central checker notifies each processor about the ID of instructions at the identified FCF, so that they can free the observed records for the already validated instructions. This notification serves as the acknowledgement of this validation phase.

In theory, there may be cases where there is a long path from an unretired instruction to a retired instruction through edges between other retired instructions. For example, suppose there is an edge from <2, 11> to <1, 9> in Figure 6, then <1, 9> is reachable from <1, 12> through the path <1, 12> → <2, 10> → <2, 11> → <1,

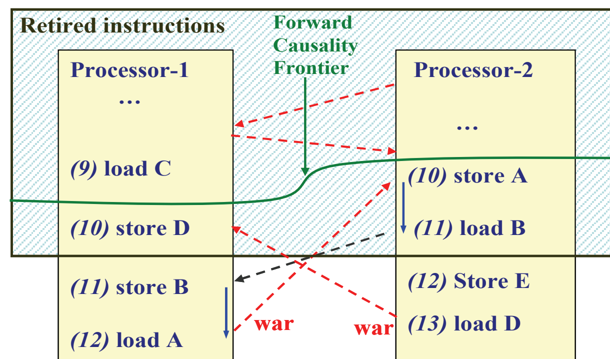


Figure 6: Example of Constraint Graph Slicing

9>. In this case the FCF should be located at instruction <1, 9> and <2, 10>. To count in these special cases, we extend Step 2 above to perform a fixed-point iterative computation of the FCF: the procedure is the same except that at the beginning of each iteration, we need to update a processor Pi's vector ID, such that it reflects the minimal ID of other processor's instructions that are successors of any Pi's instruction after the previously computed FCF. For the example in Figure 6, the FCF computed in the first iteration would be [10, 10]. Suppose we have the edge from <2, 11> to <1, 9>, then in the second iteration, we first need to double check P2's recorded edges after <2, 10> and update P2's vector ID as [9, 10], because <1, 9> is reachable from <2, 11>. Then we repeat the remaining operation described for step 2 above. The computation stops when the calculated FCF no longer changes. In the worst case, the FCF may get back to the starting instructions. However, in practice there are rarely long dependence chains from unretired instructions to retired instructions, and the computation of FCF converges quickly in mostly one and occasionally two or three iterations.

To prove the correctness of the proposed graph slicing technique, below we show that there is no back-edge across the FCF derived in the scheme above. Formal proofs are omitted here for brevity (available in the extended version [30]).

Proof sketch: 1) Since all instructions before the identified boundary have already retired and are globally performed by the end of the validation interval T, they should have been made visible to all other processors (i.e., the coherence transactions generated by the read/write request should have all been acknowledged). Therefore, all incoming edges to these instructions should have been observed using the methods described in Section 4.2 and it is not possible for them to have an additional incoming edge that originates from instructions executed after the interval T.

2) When a new instruction is executed after the interval T, the generated coherence requests may infer an additional causal dependence on a retired instruction (e.g., for the example shown in Figure 6, a "load D" executed after T at P2 will result in a RAW edge from <1, 10>). However, they will only be perceived as "happened after" the retired instruction, and cannot result in a back edge across the identified boundary. Note that this argument does not hold for the unretired instructions. An unretired instruction may be stalled in the pipeline and waiting to be executed, and the generated in-flight coherence messages may not have been observed by other processors yet. Therefore, it is possible that a back-edge to such an instruction is observed later and results in a cycle. This is why we exclude these instructions from the sub-graph that can be reduced.

3) The vector operation performed by the central checker in step 2 of the validation protocol effectively performs a backward reachability analysis in the currently observed constraint graph, which ensures that any instruction that is reachable from the unretired instructions is after the identified boundary. Therefore, there should be no back edge from instructions after the identified boundary to instructions preceding the identified boundary. Combined with fact 1) and 2), we can see that the identified boundary satisfies the requirements for the forward causality frontier.

4.5 Error recovery

When an error is detected, we rely on commonly used check-pointing schemes to resume the execution from a previously validated state. Since we perform online checking, the check-pointing scheme should also be fast and incur low overhead. A good match for these requirements is the SafetyNet scheme [9].

While it is not the focus of this work, we briefly discuss the resumption of computation based on the nature of the expected errors. For transient errors (e.g. soft errors) the computation is simply repeated, the probability of the error recurring is low. For permanent errors (e.g. design errors or permanent device defects), additional steps will be needed to ensure that the error does not recur. For example, enforcing a less aggressive execution mode (e.g., by temporarily serializing the execution) provides for an alternate constraint graph that can avoid the recurrence of the error.

5. Experimental results

Simulation environment: We evaluate our proposed approach through simulation of a dual-core chip multiprocessor system. Our simulation environment is built on the Wisconsin Multifacet GEMS simulator [23]. The baseline system parameters are summarized in Table-1.

Constraint graph evaluation: We conducted experiments with 5 programs from the SPLASH-2 parallel benchmark suite [24]. The remaining SPLASH-2 programs are not included due to infrastructure issues. Note that the original GEMS

Table 1. System Configuration

Processor Core	SPARC V9 processor 4-way , out-of-order
L1 Cache (Private)	64KB, 4-way 64-byte blocks
L2 Cache (Shared)	4MB, 4-way 64-byte blocks
Memory	1G bytes
Coherence Protocol	MSI_MOSI_CMP_Directory
Interconnection Network	PT_TO_PT

implementation only supports SC execution. The support for non-SC executions is based on GEMS extensions described by Meixner et al. [21], e.g., by adding a write-buffer to model TSO and allowing out-of-order loads in WO. We observed a simulation performance impact similar to what is reported by them. However, when running some SPLASH-2 benchmarks, the program encounters live-locks or execution errors.

To test the error detection capability of our method, we manually introduced errors in the system such that it may execute instructions in illegal order, and found cycles in the resulting constraint graphs in various cases. For example, when performing SC verification, if we allow out-of-order load/store, we found 4 cycles for FFT and 1 cycle for WATER-NSQUARED.

Figure 7 shows the maximum number of vertices of the constructed global constraint graphs when we perform the graph slicing and checking scheme described in Section 4 for SC execution. The X-axis denotes the size of the validation interval T (e.g., performing the checking at every 10K clock cycles). Figure 8 shows the maximum number of graph edges.

We can see that number of graph vertices is relatively small for a given time interval. For example, for benchmark RADIX, the graph that we need to check at the 10K-cycle interval has only 221 vertices at most. This is because our global constraint graph only consists of those memory instructions that have inter-processor dependence edges, as described in Section 3. In comparison, we observed that the naively built constraint graph that consists of all memory instructions has orders of magnitude more vertices (e.g., more than 5K vertices for the 10K-cycle interval).

Second, we can see that there is a rapid increase of the graph size when we perform checking at longer validation intervals, and the checking is performed most cost-effectively at the 10K-cycle interval. For example, for benchmark WATER-NSQUARED, the maximum number of graph vertices is 134 at 10K-cycle validation interval, which increases to 881 at the 100K-cycle interval and 3033 at the 1M-cycle interval. Without the graph slicing method presented in Section 4.4., we will have to check the entire program execution. This takes 556 million cycles for WATER-NSQUARED, and the resulting constraint graph will have over a million vertices.

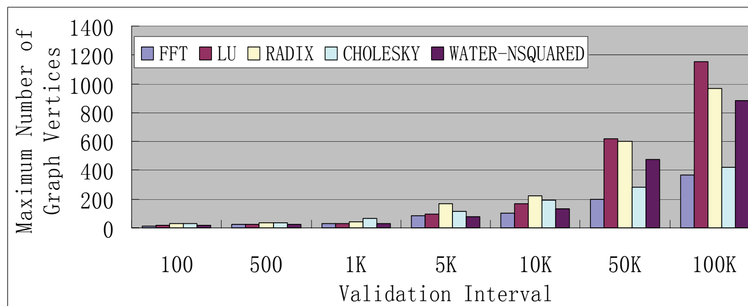


Figure 7: Maximum Number of Global Constraint Graph Vertices

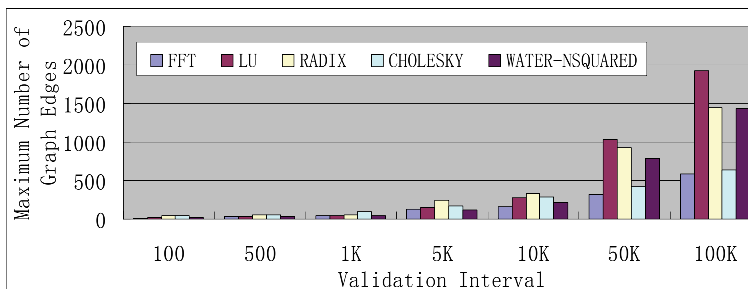


Figure 8: Maximum Number of Global Constraint Graph Edges

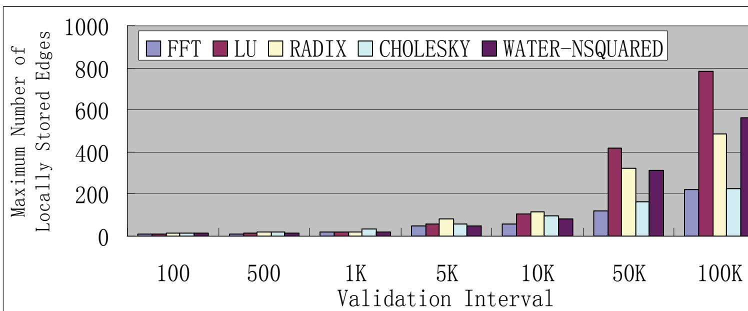


Figure 9: Maximum Size of Locally Recorded Edge List

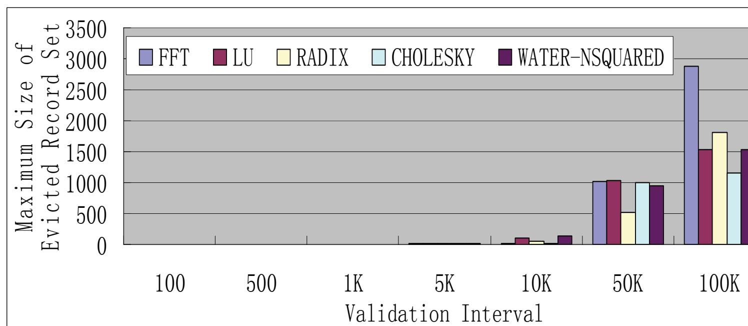


Figure 10: Maximum Size of Evicted Access Record Set

Third, we can see that the graph is quite sparse. In fact, the number of graph edges is a small multiple of the number of graph vertices. This motivates us to use the optimized design described in Section 4.3 to perform online checking of the global constraint graph. As shown in the figures, at the 10K cycle validation interval, we have 10K cycles for the graph checker to check a graph with only a few hundred edges. There is sufficient time to process this with a dedicated hardware engine without stalling.

To evaluate the hardware size required for storing the locally recorded edges and evicted cache access records, we also measured the maximum size of these at different validation intervals. Figure 9 and 10 show the results respectively. We have similar observations as the data shown in Figure 7/8, which demonstrate the effectiveness of applying the optimization techniques.

Bandwidth overhead evaluation: Since our method piggybacks on the cache coherence messages for inter-processor dependence edge observation, it incurs additional communication overhead. To measure this we run the simulation with the augmented coherence messages using 4 bytes to represent the instruction ID, and compare the reported average total traffic size and link utilization with results obtained without applying the runtime validation (i.e., coherence messages unmodified). Figure 11 shows that the average traffic overhead is 4.3%, while the average link utilization overhead is only 4.6% (figure not shown due to limited space).

Performance impact evaluation: Currently we do not have the check-point support available in our baseline simulator, so we cannot evaluate the precise performance impact with error recovery enabled. We simply let the simulation continue when an error is detected. In a system with check-point support enabled, the graph checking is done in a pipelined fashion as described in Section 4.1. Since our validation process is not on the system's critical path and is in parallel with the check-pointing process, the checking latency can be effectively hidden.

To get a better idea about what potential performance impact our method may have due to resource stalls, we examine the cumulative distribution of the measured constraint graph size for both benchmark FFT and WATER-NSQUARED at the 10K-cycle interval duration, measured over the different intervals (FFT has a total of 4681 intervals, and WATER-NSQUARED has 55641 intervals). Figure 12 plots the cumulative distribution of the number of graph edges, where the y-axis shows the cumulative fraction of graphs (over the different intervals) that have the corresponding size shown along the x-axis. So for FFT, more than 96% of the intervals have at most 40 edges in the graphs being checked. We have similar observations on other resource usage, including the locally recorded edge list and the evicted access record set, which shows that the expected performance impact in average is much lower than the worst case.

Hardware cost analysis: The hardware overhead required for performing the validation at the 10K-cycle interval is summarized in Table 2 (shown on the next page). The hardware size is determined as follows. For each augmented hardware component, we set the size to an empirical value based on the experiment results. In this experiment we perform dependence tracking at the cache block granularity, and each L1 cache block is augmented with a 4-byte field to record the access instruction information. The hardware buffer for storing the locally recorded edge list has 128 entries, where each entry has 8 bytes. The evicted cache for storing the evicted access records has 256 entries, where each entry has 12 bytes (including the address and instruction ID).

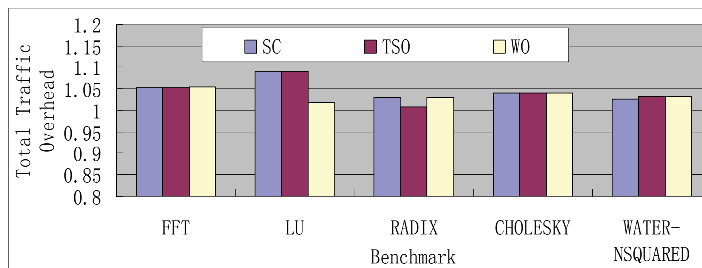


Figure 11: Total Traffic Overhead

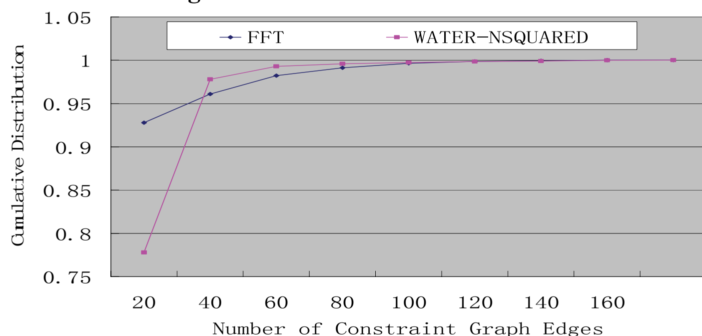


Figure 12: Cumulative Distribution of the Number of Graph Edges

Table 2. Validation Hardware Overhead

Component	Size
Local access record at L1 cache	4K bytes
Locally recorded edge list	1K bytes
Evicted access record at L2 cache	3072 bytes
Central graph checker	4K bytes

In the central graph checker, the size of the input buffer is 2KB, and the internal graph structure has 256 vertices. These are sufficiently large to avoid resource overflow in our experiments. In general, if there is a resource overflow, we handle it as follows: if the resource overflow happens locally at a processor core, we need to stall the processor execution until the observed information is verified at the end of the validation interval; if the evicted cache gets full, we can leverage the existing NACK scheme [22] to stall the cache eviction operation, until the old entries are removed at the end of the validation interval; if the resource overflow happens at the central graph checker, we need to resume the execution from the previous check-point, and perform the checking at a shorter execution interval to avoid recurring overflows or enforce a less aggressive execution mode to effectively perform graph slicing.

6. Related work

Some related work done in the area of deterministic replay and race detection [25, 26, 27] has also utilized coherence hardware based support for recording dependence edges. The authors also proposed some transitive reduction methods, which are complementary to our graph reduction technique, and when applicable, can be employed to further reduce the number of inter-processor dependence edges in our work. However, they do not address the constraint graph verification issues discussed in our work. Cain et al. proposed a dynamic verification algorithm for constraint graphs [28]. However, their work assumes only sequential consistency and in-order execution. No implementation is provided for the algorithm, which has a high space complexity (e.g., it associates each memory location with two vector time stamps).

To our knowledge, the only previous work on runtime validation of general memory consistency models is the DSN'06 paper by Meixner and Sorin [21], which is an extension of their previous work on verifying SC [29]. Their approach is different from ours in that, instead of checking the constraint graph as a direct verification against the memory ordering rules, they perform indirect verification of system invariants that are required to ensure memory consistency. As pointed out by the authors, their indirect checking approach is conservative and subject to false positive errors. Compared with the direct verification of the observed global behavior in our method, a false positive may be introduced when a detected violation of their sub-invariant does not actually cause a cycle in the global constraint graph. Further, their solution is based on the assumption that in a cache coherent system, a memory operation "performs globally as soon as it accesses the highest level of the local cache hierarchy". While they perform global checking of cache coherence property, this does not address possible design bugs or runtime errors that lead to illegal global ordering of memory accesses *to different locations*.

In practice, enforcing the assumed global access order is one of the most complicated issues in shared-memory system design. To improve memory bandwidth and performance, a memory operation is often not performed atomically, but involves several sub-transactions to allow multiple outstanding requests to be performed in parallel. Due to the delay in the interconnection network and intermediate buffers, a memory request may be visible to a local processor and remote processors at different times, during which the observed access order is vulnerable to intervention of other memory requests [22]. In general, it is a very complicated design problem to coordinate them to ensure memory ordering in a parallel system, and it relies on multiple system components to obey subtle design rules to prevent ordering violations among different memory requests. Even if cache coherence is strictly maintained, since it is only concerned with the access order to the same memory location, it does not necessarily guarantee the visibility order of memory requests to different addresses. In our case, we construct the inter-processor dependence edges based on when a load/store request is actually visible to another processor, and the dynamically maintained constraint graph reflects the actual global behavior of the target system. Thus our validation approach is more complete.

7. Conclusions

Validation of memory ordering poses a significant challenge for emerging multi-core shared-memory systems. This paper proposes a runtime validation approach to address this problem, which combines efficient hardware schemes for capturing the system behavior and effective end-to-end validation of memory ordering based on the constraint graph model. This allows us to overcome the limitations of conventional testing/simulation based verification methods, as well as to detect dynamic errors resulting from thermal conditions, aging, or particle hits. To make this approach practical, we have presented several optimization techniques that can effectively reduce the runtime validation overhead. As two key enabling techniques, we show how to use constraint graph reduction to effectively reduce the number of graph vertices, and how to use the constraint graph slicing, performed during incremental validation, to reduce the size of the execution interval that needs to be checked periodically.

Acknowledgments

The authors acknowledge the support of the Gigascale System Research Center, one of five research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation program. We thank Albert Meixner, Daniel J. Sorin, and the Wisconsin GEMS group for their help and suggestions.

References

[1] S. V. Adve and K. Gharachorloo, "Shared Memory Consistency Models: A Tutorial", IEEE Computer, vol. 29, pp 66-76, Dec. 1996.

[2] S. Sarangi, S. Narayanasamy, B. Carneal, A. Tiwari, B. Calder, and J. Torrellas, "Patching Processor Design Errors with Programmable Hardware", IEEE Micro 27, 1, Jan. 2007.

[3] S. S. Mukherjee, J. Emer, and S. K. Reinhardt, "The Soft Error Problem: An Architectural Perspective", International Symposium on High-Performance Computer Architecture (HPCA), 2005.

[4] S. Qadeer, "Verifying Sequential Consistency on Shared-Memory Multiprocessors by Model-Checking," IEEE Trans. Parallel and Distributed Systems, vol. 14, no. 8, Aug. 2003.

[5] J. F. Cantin, M. H. Lipasti, and J. E. Smith, "The Complexity of Verifying Memory Coherence and Consistency", IEEE Transactions on Parallel and Distributed Systems, Vol. 16, No. 7, July 2005.

[6] S. A. Taylor, C. Ramey, C. Barner, and D. Asher: "A Simulation-Based Method for the Verification of Shared Memory in Multiprocessor Systems". IEEE/ACM International Conference on Computer-Aided Design (ICCAD), 2001.

[7] S. Hangal, D. Vahia, C. Manovit, J. J. Lu, and Sridhar, "TSOtool: A Program for Verifying Memory Systems Using the Memory Consistency Model", International Symposium on Computer Architecture (ISCA), 2004.

[8] A. Roy, S. Zeisset, C. Fleckenstein, and J. Huang, "Fast and Generalized Polynomial-time Memory Consistency Verification", International Conference on Computer Aided Verification (ICCAD), 2006

[9] 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", International Symposium on Computer Architecture, 2002.

[10] M. Prvulovic, Z. Zhang, and J. Torrellas, "ReVive: Cost-Effective Architectural Support for Rollback Recovery in Shared-Memory Multiprocessors," International Symposium on Computer Architecture (ISCA), 2002.

[11] D. Shasha and M. Snir, "Efficient and correct execution of parallel programs that share memory", ACM Transactions on Programming Languages and Systems (TOPLAS), 1988.

[12] Harold W. Cain, Mikko H. Lipasti, and Ravi Nair, "Constraint Graph Analysis of Multithreaded Programs", International Conference on Parallel Architectures and Compilation Techniques (PACT), 2003.

[13] A. Condon and A. J. Hu, "Automatable Verification of Sequential Consistency", ACM Symposium on Parallel Algorithms and Architectures, January 2001.

[14] T. M. Austin, "DIVA: A Reliable Substrate for Deep-submicron Microarchitecture Design", International Symposium on Microarchitecture (MICRO), Nov. 1999.

[15] Arvind and J.-W. Maessen, "Memory Model = Instruction Reordering + Store Atomicity", International Symposium on Computer Architecture" (ISCA), 2006

[16] D. A. Wood, G. A. Gibson and R. H. Katz, "Verifying a Multiprocessor Cache Controller Using Random Test Generation". IEEE Design & Test of Computers 7(4) 1990.

[17] A. Meixner and D. J. Sorin. "Error Detection via Online Checking of Cache Coherence with Token Coherence Signatures", International Symposium on High-Performance Computer Architecture (HPCA), 2007.

[18] D. J. Sorin, M. D. Hill, and D. A. Wood, "Dynamic verification of end-to-end multiprocessor invariants", International Conference on Dependable Systems and Networks (DSN), 2003.

[19] J. Duato, S. Yalamanchili, and L. Ni. "Interconnection Networks". IEEE Computer Society Press, 1997.

[20] L. Lamport, "Time, Clocks, and the Ordering of Events In A Distributed System", Communications of the ACM, 21-7, 1978.

[21] 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 (DSN), June 2006.

[22] D. E. Culler, J. P. Singh, and A. Gupta. "Parallel Computer Architecture: A Hardware/Software Approach", Morgan Kaufmann Publishers, 1998.

[23] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset", Computer Architecture News (CAN), September 2005.

[24] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations", International Symposium on Computer Architecture (ISCA), June 1995.

[25] R. H. B. Netzer, "Optimal Tracing and Replay for Debugging Shared-Memory Parallel Programs", ACM/ONR Workshop on Parallel and Distributed Debugging, 1993.

[26] M. Xu, R. Bodik and M. D. Hill, "A Flight Data Recorder for Enabling Full-system Multiprocessor Deterministic Replay", International Symposium on Computer Architecture (ISCA), June 2003.

[27] M. Xu, R. Bodik and M. D. Hill, "A Regulated Transitive Reduction (RTR) for Longer Memory Race Recording", International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Oct. 2006.

[28] H. W. Cain and M. H. Lipasti, "Verifying Sequential Consistency Using Vector Clocks", 14th Symposium on Parallel Algorithms and Architectures, August, 2002.

[29] A. Meixner and D. J. Sorin. "Dynamic Verification of Sequential Consistency", International Symposium on Computer Architecture (ISCA), 2005.

[30] K. Chen, S. Malik and P. Patra, "Runtime Validation of Memory Ordering Using Constraint Graph Checking", Tech. Report, Dept. of Electrical Engr., Princeton University, 2007 http://www.princeton.edu/~kchen/rv/papers/RVMO.pdf