Runtime Validation of Memory Ordering

Using Constraint Graph Checking

Abstract

With recent advances in parallel architecture research and VLSI technology, modern processors are evolving into highly parallel, multi-core shared-memory systems. An important correctness issue for these 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. This presents a significant validation challenge, as growing system complexity makes it increasingly infeasible to cover all possible parallel execution scenarios in pre-silicon verification, which relies on simulation or formal verification techniques to expose all deep-state logic bugs. Further, aggressive scaling makes hardware more vulnerable to particle hits, circuit marginalities and aging effects, resulting in 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 simulation/formal 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 [12, 13]. By combining these we can 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 met to make this approach practical. We describe these as well as optimization techniques for reducing the hardware overhead. We evaluate our approach through simulation of a chip multiprocessor system. Estimates obtained from preliminary 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-core shared memory systems, for correct system behavior, the memory accesses must conform to the memory ordering rules specified in the architecture's memory consistency model [1]. This defines the legal orderings of shared-memory operations as observed by the different processors. This problem is complicated by the fact that memory operations may be performed concurrently by the processors and their relative timing is affected by subtle interactions among many system components including the processor pipeline, interconnection network, and cache/memory controller.

Maintaining the 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. There is an increasing recognition of a growing gap between processor design complexity and verification capability [2]. In addition to possible failures due to design errors, processors are becoming more vulnerable to dynamic faults resulting from thermal conditions, aging, or particle hits [3]. For multi-core shared-memory systems, the close interaction among the processing units can cause memory ordering errors when any component is affected by dynamic faults.

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 [5]. However, this has limited practical success due to the inherent intractability of checking memory ordering [6]. In practice, designers rely on simulation and testing based methods to examine the memory system behavior, which involves running specially generated test programs and checking the program results for possibly exposed memory errors (e.g., [7, 8]). However, these methods have the following limitations:

- The coverage is limited because it is hard for the test program to exercise all possible runtime behaviors. It is also ineffective in detecting possible dynamic errors that occur post-deployment.
- They are not applicable to general applications, because the test programs need to satisfy special data properties (e.g., each store in the test program writes a unique value) to establish the ordering between a store operation and a load operation.
- Since only the store-to-load ordering is explicitly known from the program execution results, the verification process involves expensive complex iterative analysis to infer all other necessary ordering relationships [8]. Therefore, it is not suitable for online checking and cannot handle large programs.

Our contributions: In this paper, we propose an approach for runtime validation of memory ordering, which allows us to overcome the above limitations and provides 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 [12, 13] that represents the memory ordering rules. This ensures the end-to-end correctness of the memory operations, instead of limited checking of each individual system component.

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

1. The first challenge 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 challenge 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., [10, 11]). 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 approach of our runtime validation method. Section 4 addresses the implementation issues and design

optimization techniques for further reducing the validation overhead. The experimental results are presented in Section 5. Section 6 discusses the related work, and Section 7 provides our conclusions.

2. Background

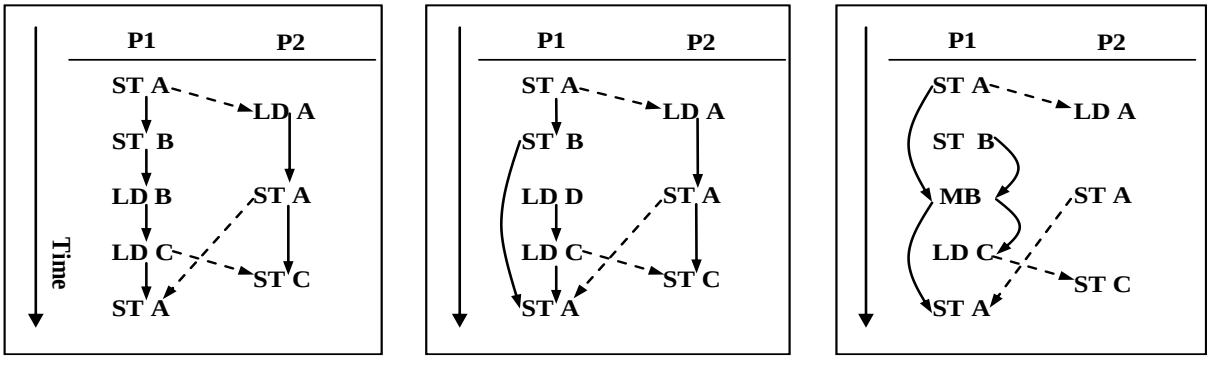
2.1 Memory Models

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. In the following discussion, we will focus on these models to illustrate the design ideas. Most existing architectures have implemented SC (e.g., SGI-MIPS, HP-PA-RISC), variations of TSO (e.g., IA-32, Sun-SPARC) and WO (e.g., Alpha, IA-64, IBM-PowerPC). 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 for Memory Ordering

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 [12, 13] (also named the access graph [14] or analysis graph [8]). 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



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

Figure 1: Constraint Graph Examples

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.

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, an important property of this graph is that it is acyclic iff the multiprocessor execution satisfies the memory ordering rules. Therefore, we can exploit this acyclic property to detect a memory ordering error.

A motivation for us to adopt this graph checking scheme is that it has been effectively used in industry. For example, it has been applied to checking Alpha's Weak Ordering (WO) model [7], Sun's TSO model [8] and Intel's Itanium [9]. 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].

2.3 Design Assumptions

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 works in the context of memory ordering verification [7, 8, 9, 13, 14] we are only concerned with the multiprocessor memory ordering issues. The constraint graph we check consists of the dynamic memory instructions and their ordering relationships. 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 validation. Moreover, recently dynamic methods such as DIVA [4] 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 been also 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 Using Constraint Graphs

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 the use of global checks to address 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, there are two major problems that render this straightforward implementation impractical. First, as pointed out in Section 1, the size of the cycle may be unbounded. Thus, the check seems possible only at the end of program execution. Checking the entire program execution as done in offline analysis has limited value because real applications may run billions of instructions before completion. We would like to perform efficient online checking periodically and detect and recover from errors shortly after they occur. 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 that will be described in the next section. In this section, we address the second challenge, i.e., reducing the graph size.

Constraint graph reduction: We note that the global constraint graph of an execution including all dependence and consistency edges between any two memory instructions will be huge even for relatively short execution intervals. So, we propose to use an equivalent (for detecting the existence of cycles) but significantly reduced graph. Further, only the information required for constructing this reduced graph needs to be observed and stored. In the following discussion we show that for SC and TSO this is just the inter-processor dependence edges. For WO, in addition to the inter-processor dependence edges we need to store the memory barrier instructions. Let each memory access have a unique identifier <P, ID> which indicates the processor and unique program order ID. Let <p, i1> and <p, i2> be two successive memory accesses in the 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 observe or store them and the associated edges.

1. For SC: Since there is a consistency edge between consecutive memory accesses in p, 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 this case the consistency edge from a ST to a

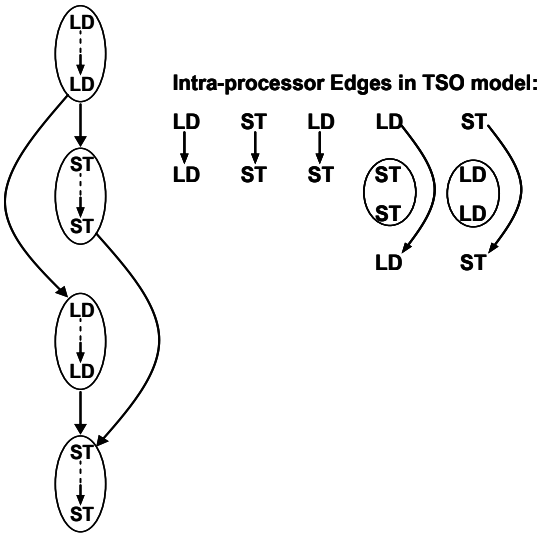


Figure 2: Intra-Processor Edges in TSO Model

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 should also not be included in the global constraint graph [15]. Thus, the intra-processor edges between memory operations are those captured by the patterns in Figure 2 (an oval in the figure denotes a group of consecutive LDs/STs). This enables us to determine the transitive closure of the intra-processor edges as follows. If <p, i2> is a store or <p, i1> is a load, 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, 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, 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 similarly constructed.

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

Our experiments show that the reduced graph has orders of magnitude less 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 number called the Memory Instruction Identifier (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

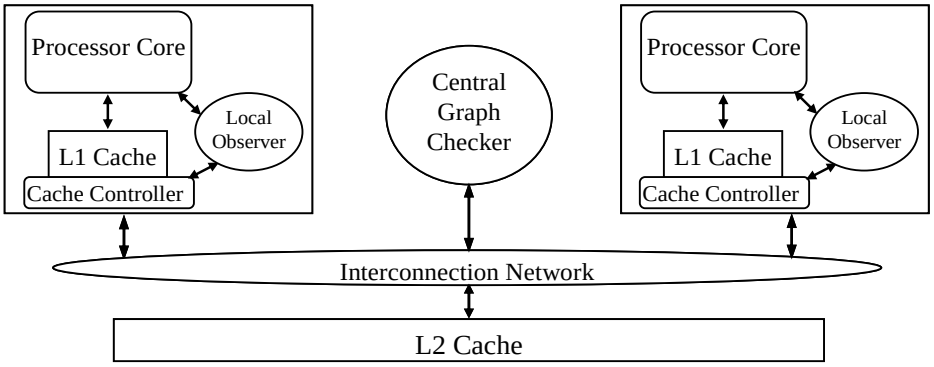


Figure 3: Overall System Architecture

memory operations are retired and validated at a time established by the method described in Section 4.4. (Further optimization is possible by recycling the Ids; this discussion is omitted for brevity.)

2) We add additional hardware to the cache structure as follows. (Their functionality is described in detail 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/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 4.5, 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, then there is an error detected and we resume the system execution from the last check-point created at T1.

4.2 Constraint Graph Edge Construction

To make the runtime validation method described in Section 3 practical, we need efficient dynamic construction schemes for the inter-processor dependence edges. (For WO we will also need to store the memory barrier instructions, which is straightforward. For the "naïve" constraint graph model of TSO [15], the intra-processor dependence edge will also need to be observed and stored locally, with additional overhead.) In constructing the inter-processor edges, we

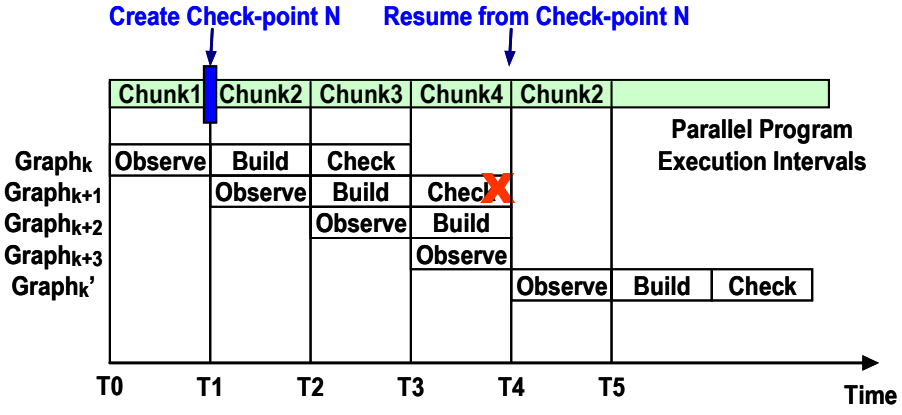


Figure 4: Timing Diagram of the Runtime Validation Scheme

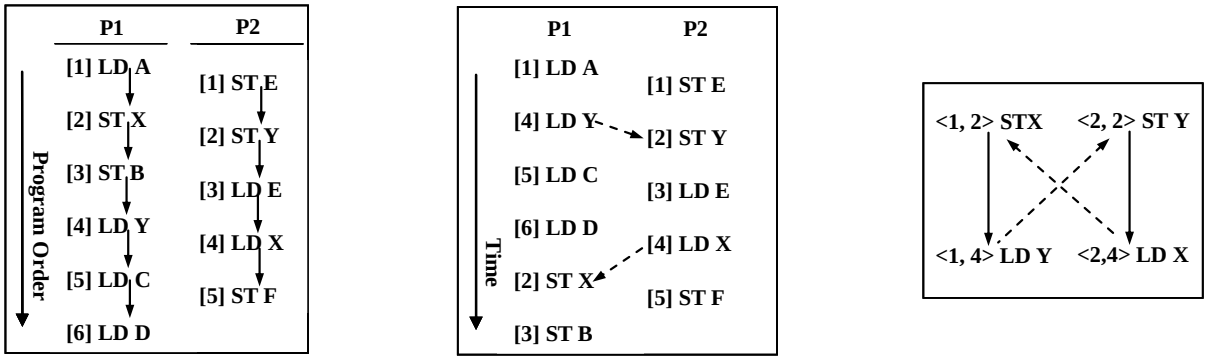
exploit the fact that each type of edge results in different cache coherence events:

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. 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. Such an edge is represented by a pair of the source and destination instructions' global IDs, and is recorded in a local hardware buffer. For example, suppose for the program given in Figure 5(a), the dynamic execution sequence is as shown in Figure 5(b). When "LD Y" is executed first on P1, we will record its ID <1, 4> in the access log associated with the cache block containing Y. 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 will then construct the dependence edge [<1, 4>, <2, 2>]. Similarly, when the data reply message is generated and sent back to the requester, we can add the responder's identity to the message, which allows the dependence edge to be observed at the requester's side.

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. 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, as discussed



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

Figure 5: Example of Dynamic Constraint Graph Construction for SC Execution

below:

1) False sharing: When a cache line contains multiple bytes of data, 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. For example, if data A and B have the same cache line address, a "ST A" on processor 1 followed by an "LD B" on processor 2 will be considered as a RAW edge. 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 the following techniques to reduce the size of the evicted set required for constraint graph checking, which work extremely well in practice:

a) Using the method described in Section 4.4, we can dynamically identify a validation boundary where instructions executed before it and the resulting constraint graph can be decoupled from subsequent execution and verified separately. Once this boundary is determined, we can remove the prior access records from the evicted set.
b) The evicted access records are filtered such that those older than the currently identified validation boundary (described in Section 4.4) are not saved.
c) When an evicted data is brought back to the L1 cache, the associated record is removed from the evicted set.

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). To facilitate the construction of intra-processor consistency edges, the entries are sorted in ascending order of the local instructions' MIDs (e.g., for the dynamic execution shown in Figure 5(b), P1 will store a list of two observed edges: <1,2>←<2,4> and <1,4>→<2,2>) . 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 re-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 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.

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). Figure 6 shows the diagram of the central checker. It consists of three pipeline stages: 1) In the first stage, the list of locally observed edges from each processor is entered into the input buffer, and a mapping is set up between each unique global instruction ID (GID) to the graph vertex ID (VID). 2) In the second stage, the global constraint graph is built by scanning the input edge list. The internal hardware representation for the graph consists of a vertex array, and each entry in the vertex array contains a pointer to a list of all its successors. In this stage the transitive closure of the intra-processor edges is inferred and added to the constraint graph as described in Section 3. 3) In the third stage, the graph is traversed using DFS by following the successor list using a dedicated hardware engine. A vertex is marked as visited during the traversal and a cycle is detected if a visited node is reached again. This has complexity O(E), where E is the number of the graph edges. As shown in the experiment section, the central checker has more than enough time to finish its operation during the typical validation intervals.

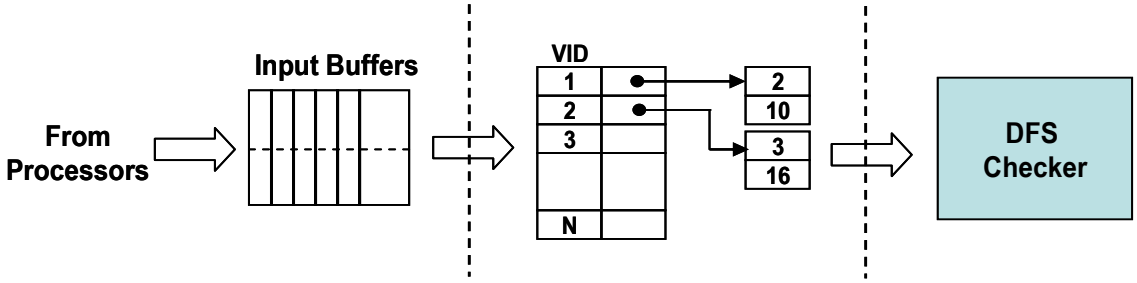


Figure 6: Central Graph Checker Diagram

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 and check a complete constraint graph. 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 computation continues. This simplifies the hardware design of 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.

One way to enable this desired optimization is to leverage the check-pointing mechanism, which is required to support error recovery. A global checkpoint represents a consistent state of the system execution, so our constraint graph may be built for the execution interval between two consecutive global checkpoints with appropriate system quiescence mechanisms. However, the check-pointing interval is likely to be way too long for recording events and building the constraint graph. In addition, the check-pointing scheme may be decoupled from the runtime validation scheme, so we have no direct control of adjusting the check-pointing intervals to support the desired optimizations.

To tackle this problem, we propose a dynamic graph reduction technique using a time-slice that can be independent of the checkpoint interval, as illustrated in Figure 7. We are only concerned with detecting a cycle in the constraint graph, which 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 can exploit the following observation: if a sub-graph in the constraint graph can be identified to not ever have an incoming edge, this sub-graph can be pruned away since it can never participate in a cycle crossing the sub-graph. This forms the basis 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 [10] and is relatively easy to implement on a CMP system. We will propose a validation protocol at the end of this section to relax this assumption.

We perform the proposed validation at the end of fixed logical time intervals. Let us consider the case illustrated in Figure 7, where at the end of a time-slice T, the execution trace of the two processors is as shown. 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 other words, 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 7. 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. Note that here by "retire", we mean the instruction has left 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

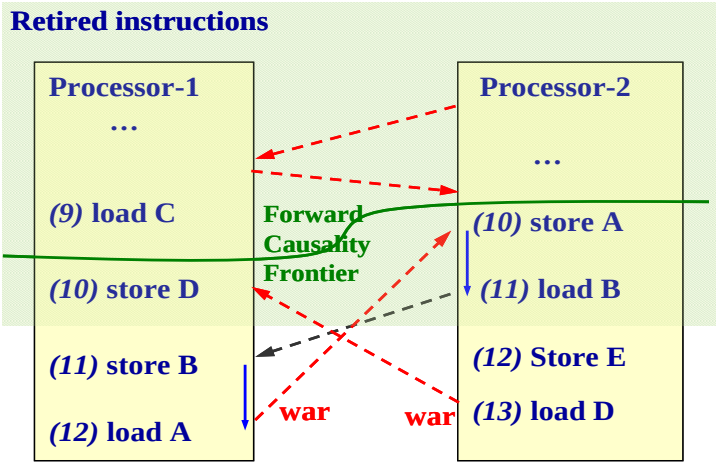


Figure 7: Example of Constraint Graph Slicing

Figure 7, we can efficiently implement a dynamic reduction protocol as follows:

1) At the end of a time-slice 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 7, at the end of time slice 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 7, 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 7, then <1, 9> is reachable from <1, 12> through the path <1, 12> → <2, 10> → <2, 11> → <1, 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 7, 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 only one or a few iterations.

The validity of the forward causality frontier derived in the above scheme can be proved from the following facts:

1) Since all instructions before the identified boundary have already retired and are globally performed by the end of the time-slice T, they should have been made visible to all other processors (i.e., the coherence transactions generated by a 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 time slice T.

2) When a new instruction is executed after the time slice T, the generated coherence requests may infer an additional causal dependence on a retired instruction (e.g., for the example shown in Figure 7, 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 have not 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.

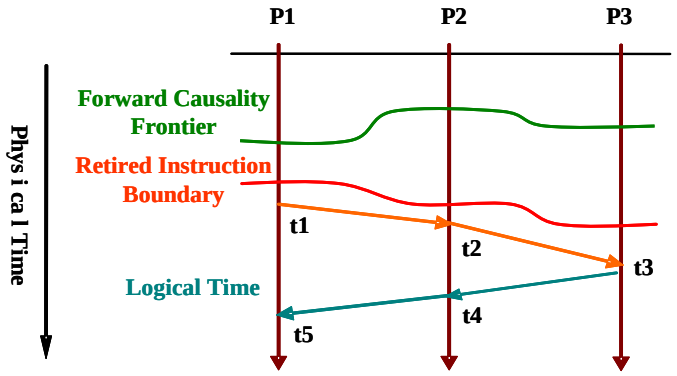


Figure 8: Example of the Validation Protocol

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.

Finally, we address how to relax the assumption on the availability of the loosely synchronized physical clock. It may be difficult to maintain such a clock on a power-efficient CMP architecture, where there are multiple clock domains with frequency or phase differences, or one processor may go to deep sleep mode when idle and its clock will be turned off temporarily. However, we can relax the assumption of a global physical time boundary for the time slice by using the following improved protocol.

1) One processor (or the central checker) periodically initiates the checking process by sending a point-to-point message to another processor, which will then pass the message to the next processor, and so on. When a processor receives this message, it records the retired instruction boundary at that moment. In the example shown in Figure 8, P1 sends a message at t1, while P2 and P3 will be notified at time t2 and t3 respectively.

2) When the last processor receives the above message, it issues an acknowledgement message, which will in turn be passed back to other processors. Whenever a processor receives this acknowledgement message, it will send the locally observed record to the central checker – the edge record should contain all edges observed up to this moment, while the instruction record should indicate the retired instruction boundary marked in phase 1. In Figure 8, this is done by P3, P2 and P1 at t3, t4, and t5 respectively. The other graph refinement and checking operations are the same as described in the previous scheme.

As shown in Figure 8, the logical time established by the validation message serves an analogous role as the synchronized physical clock time T in the previous scheme. Since we conservatively count in all unretired instructions by the time line (t3, t4, t5), the computed FCF still satisfies the property that there are no cross-boundary back-edges.

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 [10].

While it is not the focus of this work, we briefly discuss the resumption of computation. 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/concurrent memory access between processors (e.g., by temporarily switching to in-order execution mode) 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. Since the original GEMS simulator supports only sequential consistency, we first modified it to support simulation of relaxed memory models including TSO and WO, and then implemented the proposed hardware schemes for dynamic construction and optimization of the constraint graph.

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

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 related to baseline simulator limitations. 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, if we perform SC verification when running in TSO mode, we found 6 cycles for CHOLESKY and 21 cycles for WATER-NSQUARED. As discussed in Section 4, the key challenge to make the proposed approach practical is to reduce the size of the graph that needs to be checked. The effectiveness of our method is evaluated below.

Figure 9/10 shows the average /maximum number of vertices of the constructed global constraint graphs when we perform the graph slicing and checking scheme described in Section 4.4 for SC execution. The X-axis denotes the size of the time-slice T (e.g., performing the checking at every 10K clock cycles). Figure 11/12 shows the average /maximum number of graph edges. Note that it is possible that the graph cannot be sliced at certain validation intervals. In that case the observed edges during the current interval must be kept and included in the constraint graph to be checked in the following interval. This is accounted for in the data shown.

We have several important observations from these figures. First, 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 35 vertices on average. 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. For example, for benchmark WATER-NSQUARED, the average number of graph vertices is 10 at 10K-cycle validation interval, which increases to 106 at 100K-cycle interval and 1133 at 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. It is impractical to perform online construction and checking of such a large graph due to both hardware cost and performance penalty. By employing the graph slicing techniques, we can perform the cycle checking effectively at a small validation interval (e.g., 10K cycle), where the small graph size allows practical implementation of the required graph construction and checking logic in hardware.

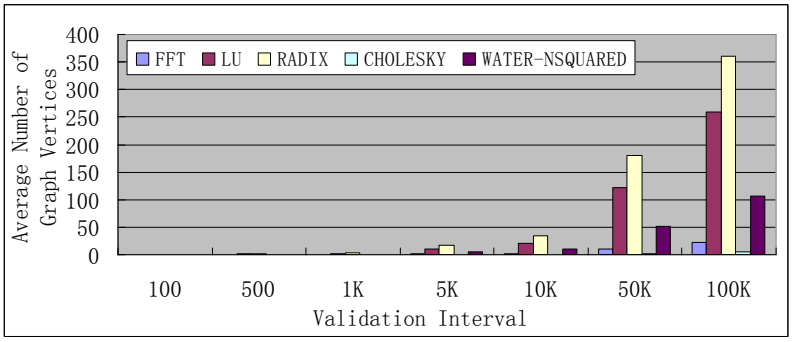


Figure 9: Average Number of Global Constraint Graph Vertices

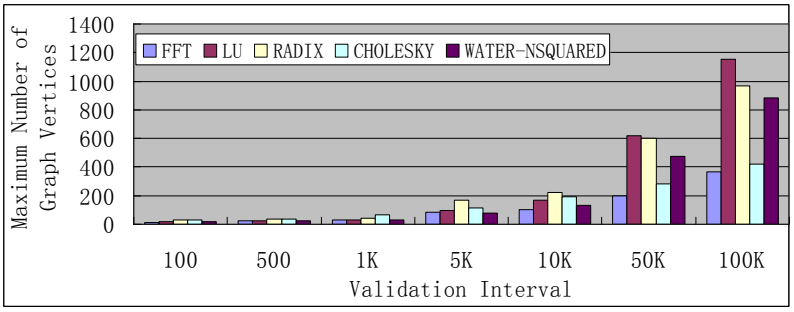


Figure 10: Maximum Number of Global Constraint Graph Vertices

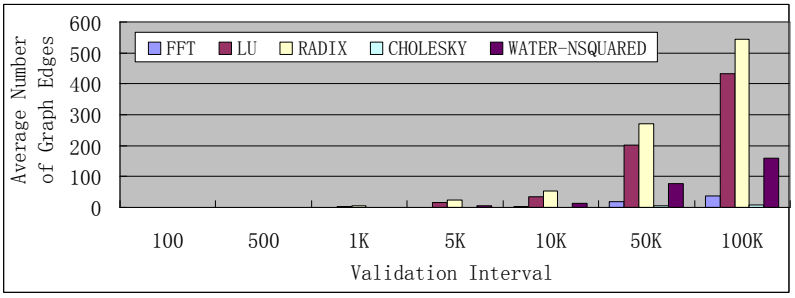


Figure 11: Average Number of Global Constraint Graph Edges

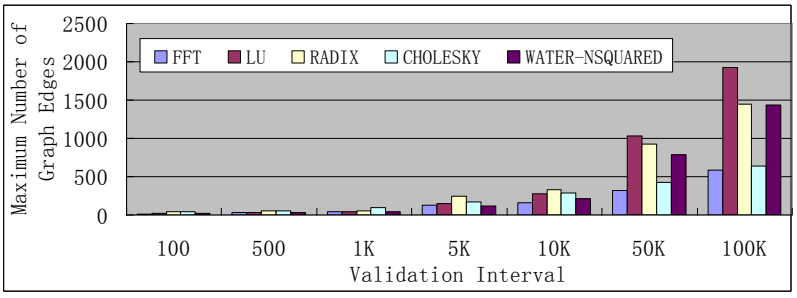


Figure 12: Maximum Number of Global Constraint Graph Edges

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 to for the graph checker the check a graph with only a few hundred edges. There is sufficient time to do this with a dedicated hardware engine.

Another important aspect is the size of the edge list that we need to record locally at each processor. Figure 13/14 shows our measured results. We have similar observations on the data as shown in Figure 9/10, which demonstrate the effectiveness of applying the optimization techniques.

To evaluate the hardware size required for storing the evicted cache access record, we also measured the maximum size of the evicted record set after applying the filtering techniques at different validation intervals. Figure 15 shows the results. We can see that the filtering techniques work extremely well at the small validation intervals. By using a set of 150 entries, we can completely avoid the set overflow for these benchmarks at the 10K-cyle interval.

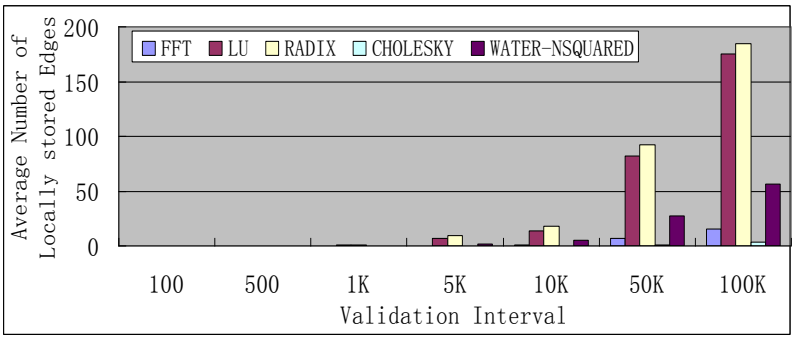


Figure 13: Average Size of Locally Recorded Edge List

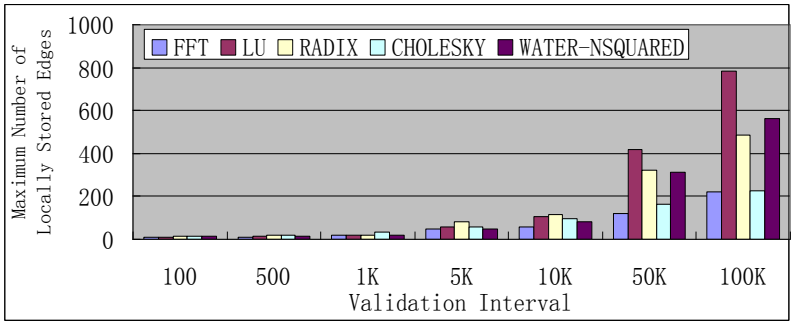


Figure 14: Maximum Size of Locally Recorded Edge List

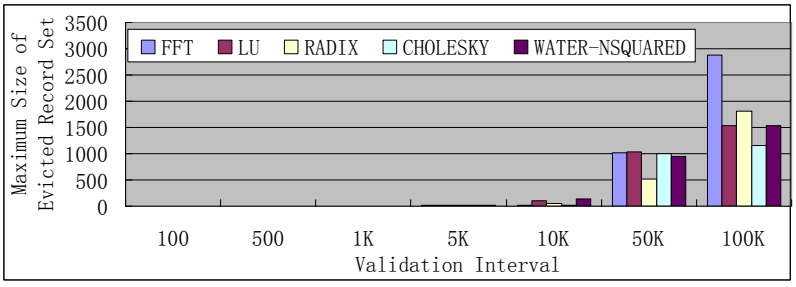


Figure 15: Maximum Size of Evicted Access Record Set

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 link utilization and total traffic size with results obtained without the runtime validation (i.e., coherence messages unmodified). Figure 16 shows the link utilization overhead, which is only 4.6% on average. The total number of bytes transferred during the simulation is reported in Figure 17. We can see that the average traffic overhead is 4.3%, which is also quite small.

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 occupancy of the critical verification hardware components 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 18 plots the cumulative

distribution of the size of the locally recorded edge list, where the y-axis shows the cumulative fraction of edge sets (over the different intervals) that have the corresponding size shown in the x-axis. So for FFT, more than 93% of the intervals have only 10 edges in the edge-list. Figure 19/20 plots the cumulative distribution of the number of graph vertices/edges. Figure 21 plots the cumulative distribution for the size of the evicted access record set. In all cases, we can see that to avoid most stalls due to resource overflow, we can select a hardware buffer size that is much smaller than the maximum storage required.

Hardware cost analysis: The hardware overhead required for performing the validation of 10K-cycle interval is summarized in Table 2 (shown on the next page). The hardware size is determined as follows: since we perform the checking at the 10K-cycle interval, we use 30-bits to represent the instruction ID, which should be sufficiently large to amortize the wrap-around handling penalty (Further optimization is possible by recycling the Id based on the computed FCF; this discussion is omitted for brevity.) For each augmented hardware component, we set the size to an empirical value based on the experiment results. For L1 cache, each cache block is augmented with a 4-byte field to record the last access 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

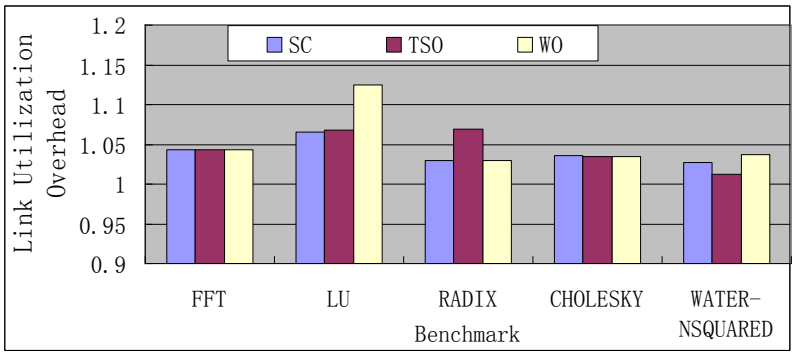


Figure 16: Link Utilization Overhead

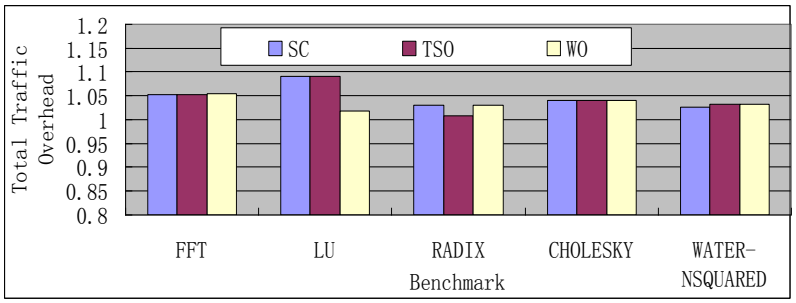


Figure 17: Total Traffic Overhead

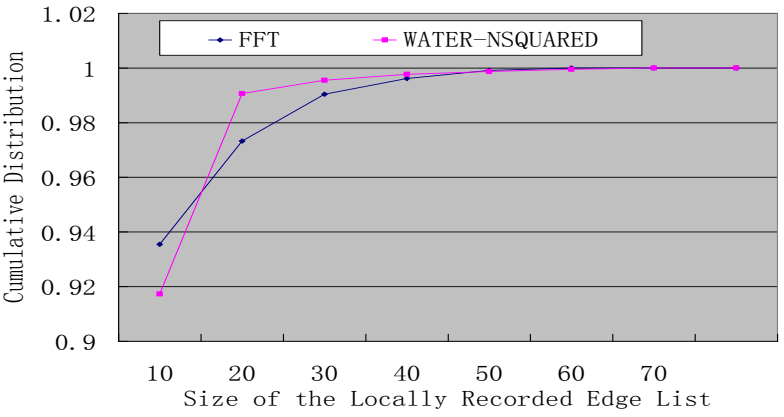


Figure 18: Cumulative Distribution of Locally Recorded Edge List Size

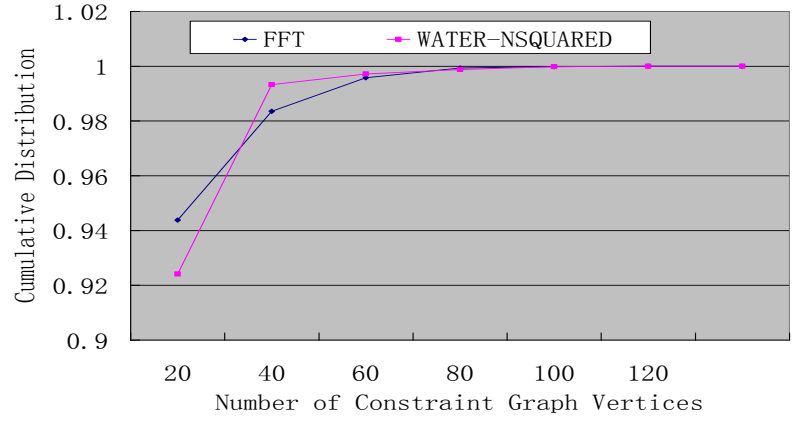


Figure 19: Cumulative Distribution of the Number of Graph Vertices

access records has 256 entries, where each entry has 12 bytes (including the address and instruction ID). In the central graph checker, the size of the input buffer is 2KB, and the internal graph structure has 256 vertices. As shown by the results in constraint graph evaluation, the proposed techniques enable us to effectively perform graph reduction, and the selected hardware size can hold all required verification information in our experiment without incurring resource overflow. In general, if there is a resource overflow, we can 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 stricter execution mode to effectively perform graph slicing.

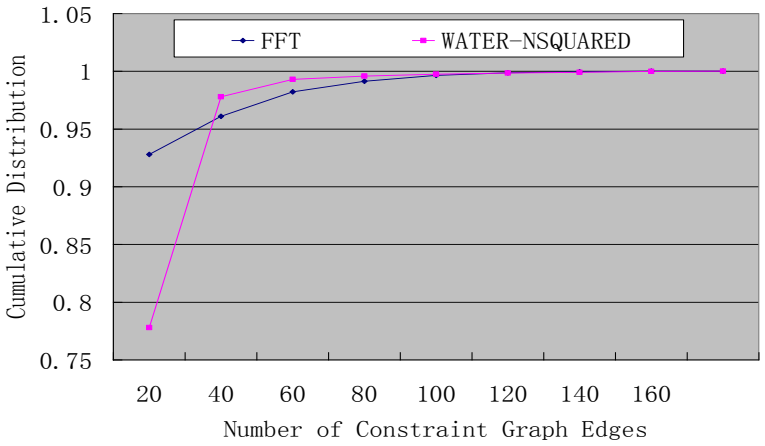


Figure 20: Cumulative Distribution of the Number of Graph Edges

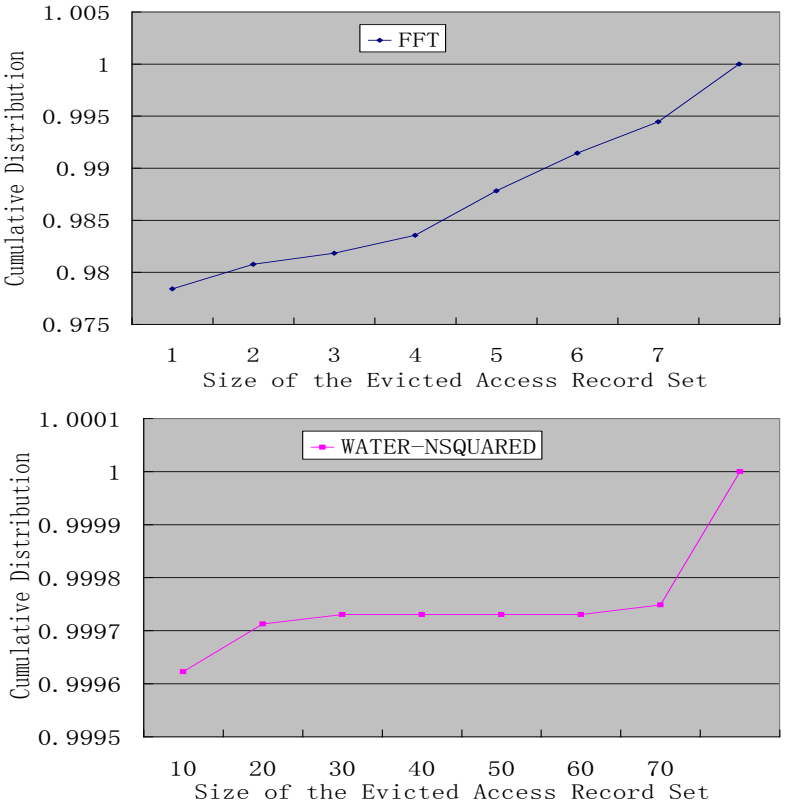


Figure 21: Cumulative Distribution of the Number of Graph Vertices

6. Related Work

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]. Their approach is different from ours in

Table 2. Validation Hardware Overhead

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

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. Since they do not examine the actual global behavior, it is subject to more sources of false positive errors that will incur a performance penalty.

Further, their solution is based on a simplistic assumption. Since they only check the local reordering of memory operations, they rely 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". Based on this, they claim that the global perform order of memory operations can be deduced from their local cache access order. But this claim may not be valid for practical systems. Relying on the system to actually maintain the assumed access timing may be problematic due to sophisticated buffering and split transactions implemented in high-performance commercial systems. In their work, they suggest that the cache coherence can be verified by examining the access history of the same cache block at the central memory controller, but this does not address possible design bugs or runtime errors that lead to illegal ordering of memory accesses at different locations.

In a practical system, 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 consistency 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 practice, it may be impossible to infer the actual global perform order of memory operations by just looking at their local cache access order. 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 should reflect the actual global behavior of the target system. Thus our validation approach is more complete.

7. Conclusions

Validation of memory ordering consistency 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.

References

[1] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," Technical Report WRL-TR 95/7, Digital Western Research Laboratory, September, 1995. This is a more detailed version of an article with the same title in IEEE Computer, vol. 29, pp 66-76, Dec. 1996.,

[2] B. Bailey, “A New Vision for Scalable Verification”, EE Times, March 18, 2004.

[3] S. S. Mukherjee, J. Emer, and S. K. Reinhardt, “The Soft Error Problem: an Architectural Perspective”, International Symposium on High-Performance Computer Architecture, 2005.

[4] T. M. Austin, “DIVA: A Reliable Substrate for Deep-submicron Microarchitecture design”, 32nd Annual International Symposium on Microarchitecture, Nov. 1999.

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

[6] 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.

[7] 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.

[8] S. Hangal, D. Vahia, C. Manovit, J. J. Lu, and Sridhar, “TSOtool: A program for verifying memory systems using the memory consistency model”. 31st International Symposium on Computer Architecture, 2004.

[9] A. Roy, S. Zeisset, C. Fleckenstein, and J. Huang, “Fast and Generalized Polynomial-time Memory Consistency Verification”. International Conference on Computer Aided Verification, 2006

[10] 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”. 29th Annual International Symposium on Computer Architecture, 2002.

[11] M. Prvulovic, Z. Zhang, and J. Torrellas, “ReVive: Cost-Effective Architectural Support for Rollback Recovery in Shared-Memory Multiprocessors,” 29th International Symposium on Computer Architecture, 2002.

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

[13] A. Condon and A. J. Hu. “Automatable verification of sequential consistency.” In Proc. of the 13th Symp. on Parallel Algorithms and Architectures, January 2001.

[14] A. Landin, E. Hagersten, and S. Haridi. “Race-free interconnection networks and multiprocessor consistency.” In Proc. of the 18th Intl. Symp. on Comp. Architecture, 1991.

[15] Arvind and J.-W. Maessen, "Memory Model = Instruction Reordering + Store Atomicity", Proceedings of the 33rd International Symposium on Computer Architecture”, 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): 13-25 (1990).

[17] J. F. Cantin, M. H. Lipasti and J. E. Smith, “Dynamic verification of cache coherence protocols”. Workshop on Memory Performance Issues, 2001.

[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, 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”, 22nd. International Symposium on Computer Architecture, June 1995.