Post-Silicon Validation of Multiprocessor Memory Consistency

Biruk W. Mammo, *Student Member, IEEE*, Valeria Bertacco, *Senior Member, IEEE*, Andrew DeOrio, *Member, IEEE*, and Ilya Wagner

***Abstract*—Shared-memory chip-multiprocessor (CMP) architectures define memory consistency models that establish the ordering rules for memory operations from multiple threads. Validating the correctness of a CMP's implementation of its memory consistency model requires extensive monitoring and analysis of memory accesses while multiple threads are executing on the CMP. In this paper, we present a low overhead solution for observing, recording and analyzing shared-memory interactions for use in an emulation and/or post-silicon validation environment. Our approach leverages portions of the CMP's own data caches, augmented only by a small amount of hardware logic, to log information relevant to memory accesses. After transferring this information to a central memory location, we deploy our own analysis algorithm to detect any possible memory consistency violations. We build on the property that a violation corresponds to a cycle in an appropriately defined graph representing memory interactions. The solution we propose allows a designer to choose where to run the analysis algorithm: 1) on the CMP itself; 2) on a separate processor residing on the validation platform; or 3) off-line on a separate host machine. Our experimental results show an 83% bug detection rate, in our testbed CMP, over three distinct memory consistency models, namely: relaxed-memory order, total-store order, and sequential consistency. Finally, note that our solution can be disabled in the final product, leading to zero performance overhead and a per-core area overhead that is smaller than the size of a physical integer register file in a modern processor.**

***Index Terms*—Cache memory, emulation, memory architecture, multiprocessor interconnection, post-silicon validation.**

I. Introduction

RECENT trends in processor design show an ever increasing number of cores being integrated on a single chip. All of today's mainstream processors, from high-end servers to mobile devices, are chip-multiprocessors (CMPs) whose individual cores communicate with each other through a shared-memory subsystem. For correct operation, a shared-memory implementation must preserve system-level properties as defined by its memory consistency model [1], i.e., read and write operations to shared-memory by multiple cores must obey the ordering requirements of the consistency model.

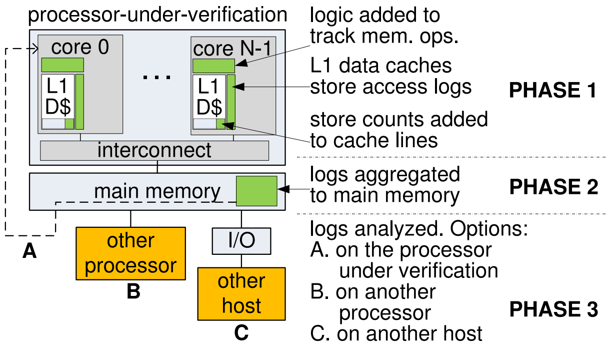
processor-under-verification
core 0
L1
D$
. . .
core N-1
L1
D$
interconnect
main memory
A
other processor
B
I/O
other host
C
logic added to track mem. ops.
L1 data caches store access logs
store counts added to cache lines
PHASE 1
logs aggregated to main memory
PHASE 2
logs analyzed. Options:
A. on the processor under verification
B. on another processor
C. on another host
PHASE 3

Fig. 1. System overview. Each core is modified to track memory accesses. A portion of the L1 caches are temporarily reserved for logging memory accesses. A store counter is attached to each cache line to track the order of writes to the line. Finally, the analysis of the logged data is performed in software, which can be carried out on a processor/host available for the task.

Unfortunately, with increasing core count, verification of shared-memory CMPs is becoming increasingly difficult, mainly due to the growing complexity of the shared-memory interactions. State-of-the-art out-of-order cores rely on aggressive performance optimizations that reorder instructions, resulting in subtle corner cases for shared-memory operations. The shared-memory subsystem is typically realized through a hierarchy of connected memory elements, which themselves might reorder memory operations for increased performance. Moreover, a system-level view of key events across a large number of cycles and over multiple components is required to validate complex memory interactions. Presilicon verification techniques are usually too slow to provide such capabilities. Emulation and post-silicon validation, on the other hand, are orders of magnitude faster, allowing for the execution of complex programs that can provide high validation coverage.

We propose a post-silicon/emulation solution for detecting memory consistency bugs that offers high bug detection capability, debugging support, low area overhead, and can be deployed on a wide range of systems, spanning multiple memory consistency models. When using our solution, a portion of the first-level data caches in the processor-under-verification are claimed to record memory interactions during test program execution. These memory access logs are then aggregated into main memory and analyzed in software. This approach results in a significantly smaller silicon area overhead than similar

Manuscript received June 21, 2014; revised October 20, 2014; accepted January 6, 2015. Date of publication February 10, 2015; date of current version May 20, 2015. This work was supported in part by the National Science Foundation under Grant 1217764, and in part by C-FAR, one of the six SRC STARnet Centers, sponsored by MARCO and DARPA. This paper was recommended by Associate Editor S. Patil.

B. W. Mammo, V. Bertacco, and A. DeOrio are with the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109 USA (e-mail: birukw@umich.edu).

I. Wagner is with the System Validation and Engineering Group, Intel Corporation, Hillsboro, OR 97124 USA.

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2015.2402171

solutions proposed in the literature. Furthermore, our solution is transparent to the end user as logging and analysis is disabled and cache space is released upon product shipment. We demonstrate our approach on a CMP with out-of-order cores, each with private L1 caches.

II. System Overview

Our solution partitions test program execution into multiple epochs, each comprising three distinct phases, as illustrated in Fig. 1. In the first phase, program execution progresses while memory accesses are tracked in the background and logged into a reserved portion of the processor's L1 data caches. A small amount of additional logic is required to perform this task, but note that the additional hardware is off the critical computation paths. When the system exhausts logging resources, program execution is temporarily suspended and program state is saved, to be restored later at the start of a new epoch. The system then transitions into the second phase, where the memory access logs from all the caches are aggregated into main memory. In the third phase, these logs are analyzed by software that checks whether memory consistency violations have occurred during program execution.

Our analysis algorithm builds a specialized directed graph using the information in the memory access logs. Each vertex in the graph represents a memory access, while directed edges are generated between memory accesses based on the observed order of the memory operations and on the requirements of the memory consistency model. The presence of a cycle in this graph indicates that a memory consistency violation has occurred [2], [3]. The information in the memory access logs and the architectural state at the end of each epoch provide insight into the activity of the CMP during an epoch, which can be used to find the root cause of a consistency violation. This analysis can be carried out on- or off-chip, based on available resources and validation strategy.

In Fig. 1, we outline the hardware modifications required to implement our solution. Each core is augmented with a small amount of logic to enable memory access tracking. All cache lines are also augmented with a store counter for tracking the order of writes to a cache line. When a cache line is transferred to a different core, its store counter is also transferred with it. In addition, the L1 cache controllers are modified to temporarily reserve and utilize portions of the cores' L1 data caches for storing memory access logs. All of these modifications applied to the processor-under-verification can be completely disabled after validation, resulting in virtually no performance and energy overheads to the customer.

Our solution allows the log analysis to be performed in a range of time interleavings, as illustrated in Fig. 2. For instance, if the only goal for a test is the verification of memory interactions, then it is reasonable to analyze the recorded logs right after each execution phase and terminate the test if a violation is detected. Here, the analysis may be serialized as in Fig. 2(a) or overlapped with the next epoch's execution as in Fig. 2(b). The scenario in Fig. 2(a) would allow for the analysis software to run on the same processor under verification. In cases where logs have to be transferred off the validation platform for analysis on another host, it may be more efficient to adopt the scenario in Fig. 2(c) to amortize log transfer overheads. This latter setup is especially useful for emulation-based validation flows, where it may be impractical to execute the analysis software on the processor being emulated due to performance limitations.

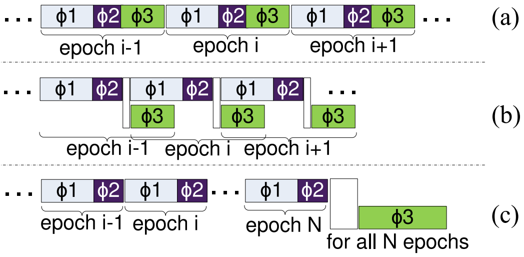

Fig. 2. Timeline scenarios. When logging resources are exhausted, test program execution is suspended and logs are aggregated for analysis. The analysis may then be done (a) before beginning the execution phase of the next epoch, (b) overlapped with the execution phase of the next epoch, and (c) at the end of the test program execution.

III. Background

The memory subsystem in a typical modern CMP consists of two or three levels of caches and a main memory that service memory requests from load-store units in out-of-order cores. From the perspective of the programs running on the processor, the memory subsystem has to appear as if it were one monolithic structure that preserves the ordering properties specified by the memory consistency model. Several consistency models have been proposed and adopted over the years [1], [4], mainly driven by the need to allow high performance CMP implementations. A consistency model for a particular architecture specifies the acceptable orderings of memory accesses that multithreaded shared-memory programs should expect. Below is a brief description of the ordering requirements for three representative models.

Sequential Consistency (SC) [5]: All memory operations from each core must be performed in their respective program order. In addition, there should exist a unique serialization of all memory operations from all cores.

Total-Store Order (TSO) [6]: All memory operations from each core must be performed in their respective program order, except for when a store operation to a memory location is followed by a load operation from another location. In such cases, a core is allowed to execute the load early, before the store completes. Finally, all cores should observe the same unique serialization of all store operations in the system.

Relaxed-Memory Order (RMO) [6]: No implicit ordering requirements are enforced amongst memory operations to distinct memory locations. The programmer can explicitly enforce ordering by inserting fence (barrier) instructions.

Weak memory consistency models, such as RMO, rely on fence instructions to explicitly order memory operations. These fence instructions include a field to specify the ordering constraint being enforced by the instruction. For instance, the SPARC V9 ISA [6] includes a `MEMBAR` instruction with two mask fields—`mmask` (4 bits) and `cmask` (3 bits).

Each mmask bit represents the possible combination of memory operations that cannot be reordered across the instruction (bit 0: load→load, bit 1: store→load, bit 2: load→store, bit 3: store→store)—if all four mmask bits are set, then no memory operation reordering is allowed across the MEMBAR. The cmask bits are used to specify ordering requirements between memory operations and other preceding instructions. Here, we limit the scope of this paper to consider ordering only with respect to other memory operations. Note that several consistency models can be modeled using RMO with an appropriately masked fence instruction inserted after each memory operation. For example, we could model SC by inserting MEMBARs, with all mmask bits set, after each memory operation. In addition, the most commonly used synchronization instructions (test-and-set, exchange, compare-and-swap, load-linked, and store-conditional) can be modeled as sequences of reads, writes and fences.

Previous work has shown that bugs in an implementation of a memory consistency model can be detected by constructing a directed graph from observed memory operations and then searching for cycles in the graph [2], [7]–[11]. The graph should be acyclic if the execution satisfies the ordering rules given by the consistency model. We present an example of such a graph in Fig. 3 for a multithreaded execution on a system based on the RMO consistency model. Fig. 3(a) shows snippets of memory operations from three different cores for a multithreaded program. Our mechanism tracks the ordering requirements of the fence instructions and the data dependencies between memory operations. Consider a case where each core performs its memory accesses in program order and accesses from multiple cores are arbitrarily interleaved. Fig. 3(b) reports a graph obtained from such an execution. The vertices in the graph represent memory accesses, the red dashed edges are generated from the ordering requirements of the fence instructions, and the solid black edges are generated from the data dependencies observed during execution. A different execution may result in different data dependencies, and hence a different graph. Assume core 0 executes LD A (load from address A) before the preceding memory accesses complete, violating the explicit ordering requirements of the two fence instructions in its instruction stream. In this case, core 0 would load the value that is set in its store queue, written by the ST A (store to address A) in its instruction stream, instead of the value from core 1. This results in a different graph as reported in Fig. 3(c). Note that violating the ordering requirements of the fence instructions introduced a cycle in the graph.

The main focus of this paper is the detection of memory consistency bugs that may occur when multiple cores execute multiple threads. We assume that: 1) the system is a cache-coherent CMP that enforces a single writer to a cache line at a time, while allowing multiple readers; 2) the L1 caches implement a write-allocate (fetch-on-write) policy for store misses; 3) the threads executing on the CMP interact only through shared memory accesses; and 4) intrathread data dependencies are handled correctly or there is an orthogonal mechanism addressing this type of issues, possibly among those listed in [12]. In developing our solution, we also make use of the following observations.

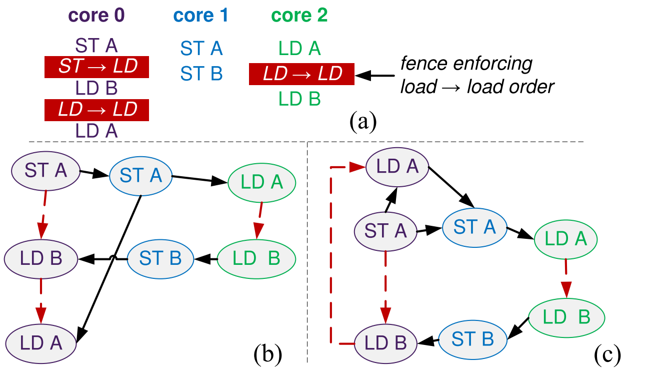

Fig. 3. Memory access graph examples. (a) Sequence of memory operations and fence instructions from three processor cores. (b) Memory access graph for an execution that abides RMO rules. Solid black edges represent data dependencies, while dashed red edges represent fence-enforced orderings. (c) Core 0 executes LD A out of order, in violation of the fences in its instruction stream. This violation manifests as a cycle in the resulting memory access graph.

1) If a thread does not share data with other threads, the core executing it can reorder memory operations compatibly with the correct enforcement of data dependencies. In this case, the thread's execution abides the constraints of any consistency model.
2) If a thread shares data with other threads, it can reorder its memory operations as long as no other thread observes memory access orderings that violate the system's consistency model. If violations do occur, they manifest as cycles in the corresponding memory access graph, involving both interthread edges and edges derived from the rules of the consistency model.
3) For a cache-coherent CMP, as we assume, a unique serialization must exist for all stores to a cache line.
4) The interthread edges in the memory access graph can only result from interthread data dependencies.

Observation 2 dictates that we collect information required to construct the required edges to detect violations. In order to construct the edges imposed by the consistency model, our solution collects information about the relative ordering of memory accesses as dictated by the memory consistency model. To construct the interthread edges, we collect information about data dependencies, utilizing the property in observation 4. Observation 3 provides us with a mechanism to keep track of data dependencies; by uniquely identifying each store to a cache line, not only are we able to determine the order of write accesses to a cache line, but also which store operation produced a value for a subsequent load. We can capture read-after-write (RAW), write-after-write (WAW), and write-after-read (WAR) data dependency edges with this mechanism. The following sections describe in detail how this information is collected and then used to construct memory access graphs for the subsequent analysis.

IV. Tracking Memory Access Orderings

Our solution tracks two types of ordering information: 1) implicit or explicit ordering enforced by the consistency model on memory accesses from the same thread (henceforth referred to as consistency order) and 2) ordering observed during execution due to data dependencies between memory accesses (henceforth referred to as dependence order). We utilize some of the processor's L1 data cache space to temporarily store this information. We present the details of the information collection mechanism below.

A. Capturing Consistency Order

Our solution tags each memory access with a sequence identifier that marks its position in consistency order, relative to other memory accesses from the same thread. The generation of sequence identifiers is dependent on the memory consistency model under consideration. SC, for instance, requires program order to be enforced between all memory accesses (consistency order is the same as program order). Therefore, unique and monotonically-increasing sequence identifiers are required to capture the ordering requirements. These sequence identifiers can be generated by using a simple per-thread counter that is incremented on every memory access. On the other hand, RMO does not impose any implicit ordering constraints between memory accesses to different addresses. A programmer can explicitly enforce ordering through the `MEMBAR` fence instruction. Depending on the `mmask` field of the instruction, loads and/or stores before the `MEMBAR` are required to be ordered before loads and/or stores after. For such a case, a sequence identifier needs to identify the relative position of memory accesses with respect to fence instructions. Since not all memory accesses are affected by a particular fence instruction, the sequence identifier must also encode the `mmask` fields of the `MEMBAR` instructions. The sequence identifier for a memory access can thus be constructed from a {count, mask} tuple where the count is the number of `MEMBAR`s encountered before the memory access instruction and the mask is the value in the `mmask` field of the last `MEMBAR` preceding the memory access instruction.

We observe that a generic solution based on sequence identifiers for RMO can be extended for use with any other consistency model. For instance, for SC and TSO, the count field of the sequence identifier tuple is incremented after every memory instruction, while the mask field is kept at a constant value to reflect the restrictions on the types of memory accesses that can be reordered—no reordering is allowed for SC (i.e., `seq_id.mask = 0xF`) and loads are allowed to be reordered with respect to previous stores for TSO, while every other reordering is restricted (i.e., `seq_id.mask = 0xD`). We can then model SC and TSO using RMO by assuming the existence of `MEMBAR`s with `mmask` `0xF` and `0xD`, respectively, after each memory access instruction. We will use this generic model to discuss our solution without delving into the particulars of any memory consistency model.

We add a special sequence identifier register, with count and mask fields, to each core. When a fence instruction is dispatched, the count is incremented and the ordering constraints imposed by the fence instruction are stored in the mask field. We also add a "retirement sequence identifier register" which is updated in a similar fashion when a fence instruction is retired. The precise (nonspeculative) value in this register is copied into the actual sequence identifier register when the pipeline is flushed. Upon dispatch, all memory accesses are tagged with the value in the sequence identifier register. Fig. 4 illustrates an example of the sequence identifier generation process. The first `ST A` access is tagged with a {count, mask} tuple of {`0,0x0`}, from the 0-initialized sequence identifier register. The fence instruction following the store causes the sequence identifier register to be updated; the count is incremented and the fence instruction's mask is copied to the mask field (`mask = 0x2`, i.e., loads can not be reordered with respect to previous stores). All subsequent memory instructions are then tagged with the new sequence identifier, until a new fence instruction updates the sequence identifier register.

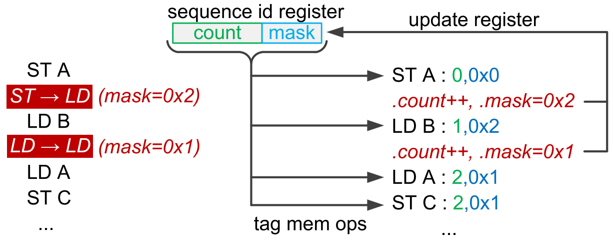

Fig. 4. Memory access tagging. Upon dispatch, each memory operation is tagged with the value currently stored in the sequence identifier register. When a fence instruction is encountered, the count field in the sequence identifier register is incremented and the fence mask is copied to the mask field.

In an Intel P6-like microarchitecture, the sequence identifiers for all in-flight memory instructions can be appended to entries in the load and store queues. When a memory instruction is finally performed, the logging mechanism reads its sequence identifier and stores it in the portion of the cache temporarily reserved for our solution's logging.

B. Capturing Dependence Order

A unique serialization of all stores to a cache line exists for a program executing on a cache-coherent CMP. Our solution leverages this property to track data dependencies. We attach a store counter to each cache line, which is incremented upon each write to an address in that line. To hold the store counter, we repurpose the error-correcting code (ECC) bits of the cache line. Note that the counter value is transferred along with the cache data of the line to the cores and through the memory hierarchy. The ECC update mechanisms for all memory elements in the hierarchy are repurposed to preserve the transferred ECC bits. Each memory access is tagged with the most recent store counter value for the cache line it accesses. This allows our solution to infer the relative order among shared-memory operations to a given cache line, and thus to derive RAW, WAR, and WAW dependency edges in the memory access graph. In addition, these counters enable a straightforward mechanism for verifying the single-writer-at-a-time requirement of a cache-coherent CMP—each write access to a cache line must be tagged with a unique count. A memory operation that accesses multiple cache lines is split into multiple log entries to preserve the store counter values for all the accessed lines. During analysis, such accesses will be merged and all the relative orders inferred from the multiple store counters will be included.

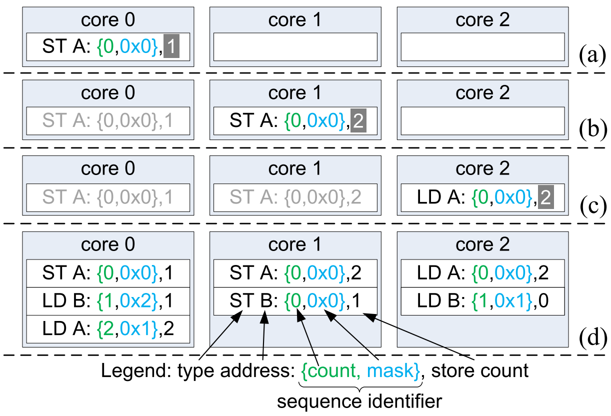

Fig. 5. Memory access logging example for the program snippet in Fig. 3(a). (a) Core 0 performs its first store to line A, updating the store count for A to 1 and taking a snapshot. (b) Core 1 performs a store to A invalidating core 0's cache line. A is transferred to core 1 with an incremented store count of 2. (c) Core 2 loads A, which is transported along with the store count of 2 from core 1. (d) Final contents of the logs that generate the graph of Fig. 3(b).

Fig. 5 illustrates the store counting mechanism and memory access logging for 3 cores executing the program snippet shown in Fig. 3(a). Assume that the cores perform one memory access at a time in the following order: core 0, core 1, core 2, core 2, core 1, core 0. This would result in the memory access graph shown in Fig. 3(b). Note that all memory accesses are tagged with appropriate sequence identifiers as discussed in Section IV-A. When core 0's `ST A` writes to core 0's cache, the store count for the written cache line is incremented to 1. Fig. 5(a) shows the snapshot of this store count, the sequence identifier tuple, the type of memory access and the address logged in core 0's log storage space. Core 1's `ST A` instruction causes core 0's corresponding cache line to be invalidated and the store count to be shipped to core 1. The store count is then incremented and its snapshot is stored in core 1's log storage space, along with the rest of the log entry, as shown in Fig. 5(b). This updated store count is shipped to core 2, along with the data for the corresponding cache line, when core 2's `LD A` is executed. The log entry for core 2's `LD A` access then contains a store count of 2 as shown in Fig. 5(c). Fig. 5(d) shows the final state of the memory access logs after all memory accesses have been performed.

Note that if core 0's `LD A` had received its value forwarded from core 0's store queue, the store count associated with this access would have been 1 and as a result, the memory access graph shown in Fig. 3(c) would have been generated.

C. Logging Mechanism

We reserve some of the ways in a multiway, set-associative L1 data cache for log storage. A single log entry holds information about the type of memory access, the memory address, the sequence identifier and a snapshot of the store count for the cache line. For the type information, one bit is used to indicate if the access was a load or a store and a second bit is used to indicate whether it should be merged with the preceding access during analysis. Merging is needed for log entries of memory access that cross cache line boundaries. The data arrays in the reserved cache ways are repurposed to hold log entries, with each entry aligned to byte boundaries.

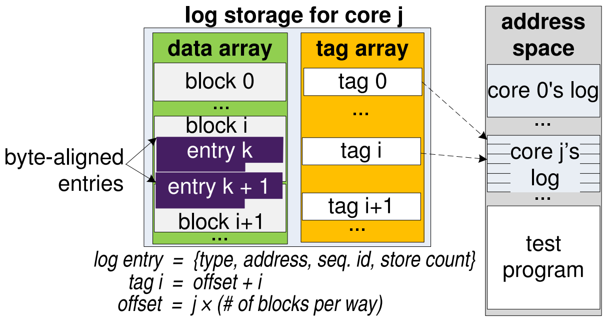

Fig. 6. Logging mechanism: the data arrays in the reserved cache ways are used to store log entries. The tag arrays store address bits to direct the log entries to their allocated space in memory during log aggregation. In the figure, we report the tag computation for the case when only one way of each cache is reserved for log storage.

After the execution phase of an epoch, the log entries from all cores must be aggregated for analysis. A section of the address space is allocated for storing the aggregated logs. We utilize the existing cache line eviction mechanism to move the log entries down the memory hierarchy. To enable this simple transfer mechanism, the tag arrays in the cache ways reserved for log storage are populated with the appropriate address bits to direct the logs to the proper memory locations. Fig. 6 shows the details of the logging mechanism.

D. Log-Delay Buffer

The logging of a memory access needs to be delayed for certain special cases. First, obtaining the store count values might not be straightforward for load operations that obtain their values forwarded from stores in a load-store queue or a store buffer. For such loads, the store count that needs to be associated with the corresponding log entry can not be obtained until the forwarding store accesses the cache and updates the store count. Second, when multiple stores to a cache line are coalesced in the store buffer, the store count updates corresponding to each store must be applied, even though the cache line is accessed only once. Third, a speculative memory access should not be logged until the instruction that issued the access has retired. Lastly, all available L1 write ports maybe in use when our mechanism has an entry to write into the portion of the L1 cache reserved for log storage. To handle these cases, we incorporate a log-delay buffer for saving log entries waiting to be written to the log storage space. Our logging mechanism ensures that entries update their information and leave the log-delay buffer whenever the event they are waiting for occurs.

Logging is performed in the background during program execution until log resources are exhausted. A store count reaching the maximum value, a data block in the reserved log storage filling up, or the log-delay buffer running out of space signal the end of an epoch. The log-delay buffer is designed to handle at least as many entries as the combined sizes of the store buffer and the load-store queue, to reduce the probability of filling up. The fence counter field in the sequence identifier is designed to count at least up to the maximum number of memory operations that can be in flight at any given time. This allows our analysis software to easily detect when the

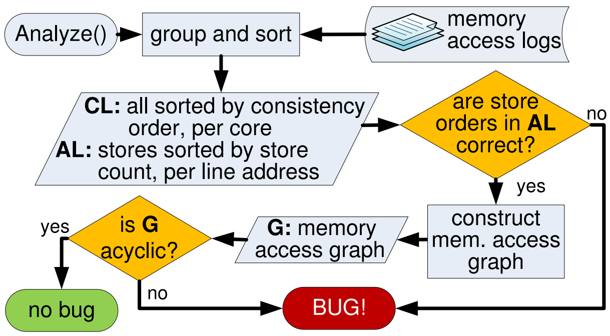
Analyze()
group and sort
memory access logs
CL: all sorted by consistency order, per core
AL: stores sorted by store count, per line address
are store orders in AL correct?
no
yes
construct mem. access graph
G: memory access graph
is G acyclic?
yes
no
no bug
BUG!

Fig. 7. Log analysis algorithm. The algorithm first creates sorted data structures. The sorted memory accesses for each line address are checked for valid store ordering. The graph construction algorithm infers all ordering edges from the sorted memory access lists. The resulting graph is topologically sorted to detect cycles.

counter has wrapped around for the log entries so that it updates all entries thereafter with correctly increasing count values.

V. Log Aggregation and Analysis

At the end of the execution phase of an epoch, our system makes a transition from normal program execution mode to log aggregation mode by: 1) saving the architected state of the processor to memory, much like how it is done during a conventional context switch; 2) disabling the lower level caches and portions of the L1 caches that were used for normal program data; and 3) running software that triggers evictions for the cache lines holding the logs. At the end of the log aggregation phase, the logs of memory interactions and the final architected state of the test program reside in main memory. The analysis phase can then resume in one of three ways as described below.

1) *Analysis Leveraging the Processor Under Verification:* The tracking and logging mechanism is temporarily disabled and the analysis software is loaded and executed.
2) *Analysis on a Separate Processor Connected to the Same Memory:* The analysis software is executed on the separate processor.
3) *Analysis Using a Separate Host:* The logs are first transferred from the memory connected to the processor under verification to the memory of the separate host machine. The analysis software is then executed on the host.

Fig. 7 shows a high-level flowchart of our analysis algorithm. This algorithm first creates two data structures that provide easy access to the memory access logs—the core list data structure (`CL`), which keeps each core's memory accesses in a list sorted by consistency order, and the address list data structure (`AL`), which keeps each store to a cache line in a list sorted by store count. These data structures, generated by the "group and sort" process in the flowchart of Fig. 7, enable efficient implementations of the store order checker, the graph constructor and the cycle detector, which are discussed below.

A. Checking Order of Writes to Cache Line

For a cache-coherent CMP, there must exist a strictly increasing sequence of store counts, per cache line, if only one writer is allowed to a cache line at a time. Fig. 8 shows an example for a cache coherence bug that allows two stores to write to a line simultaneously. Core 0 fails to downgrade its permission (transition from modified to invalid, for a simple MSI protocol) to cache line A before core 1 upgrades its permission. This violates the single-writer-at-a-time invariant, enabling both cores to write to A at the same time. This violation manifests as two consecutive stores with the same store count in the address list for line address A (`AL[A]`).

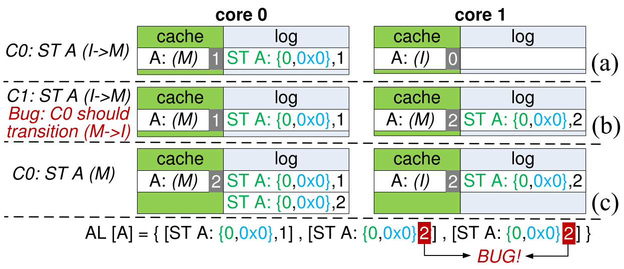

Fig. 8. Coherence bug example. (a) Core 0 performs a store to line A. (b) Core 1 gets line A from core 0 to perform a store, however, core 0's cache fails to downgrade its permission to line A. (c) Core 0's store to line A updates the store count, leading to a conflict detected by the store order checker.

B. Graph Construction and Cycle Detection

The graph construction subroutine constructs a directed graph from the observed memory accesses, their data dependencies, and the ordering constraints imposed by the memory consistency model. A vertex in the resulting graph represents a memory access and an edge between two vertices represents the order between the two accesses. Dependence order edges are constructed using the store count snapshots. Every memory operation is placed in between two stores—one occurring right before it and the other occurring right after, in dependence order. The sequence identifier information attached with each log entry is used to construct the consistency order edges.

Fig. 9 details our graph construction algorithm using the example introduced in Fig. 3. Note that the logs shown in Fig. 9(a) are from an execution where core 0's `LD A` violates the ordering requirements and obtains its value early from the `ST A` in its instruction stream. The memory access logs are first organized into two lists as presented in Fig. 9(b). For each core *i*, accesses are placed in `CL[i]` in increasing order of their sequence identifiers. Stores to each cache line *x* are placed in `AL[x]` in increasing order of their store count. The graph construction algorithm walks through each access in `CL[i]` and attempts to construct directed edges to and from the access. First, dependence order edges are constructed by identifying preceding and succeeding stores in dependence order. This is easily achieved by directly indexing `AL` using indices derived from the address and the store count of the memory access. Next, for each preceding access in `CL[i]`, the algorithm checks if the effective ordering requirement warrants an edge to the current access. The effective ordering requirement is obtained as a bitwise AND of all masks in the sequence identifiers of the memory accesses appearing between the two memory accesses in `CL[i]`. The edges

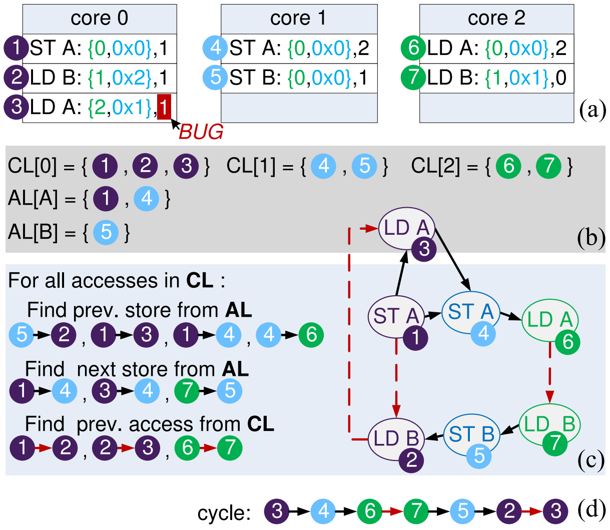

Fig. 9. Graph construction example. (a) Memory access logs from the example introduced in Section III. (b) Contents of CL and AL derived from the logs. (c) Graph construction algorithm generates edges utilizing the entries in CL and AL. (d) Cycle is identified in the resulting memory access graph.

derived by our algorithm and the resulting memory access graph are shown in Fig. 9(c). The cycle due to the violation of the ordering requirements is presented in Fig. 9(d).

VI. Strengths and Limitations

A. Debugging Support

The memory access logs, preserved architectural state and the state of the memory at the end of each epoch can be leveraged to identify the root cause of an error. We can further enhance the debugging capability that our solution provides by adding a slow debug mode where additional useful information can be collected.

B. Low-Overhead Store Tracking

In [7], we found that using access vectors to track the order of stores to an individual cache line resulted in an efficient analysis algorithm. However, such a mechanism for storing and transferring access vectors with each cache line cannot scale well with increasing core count. Our single store count can be piggy-backed on the ECC bits for a cache line, obviating the need for extra storage and bandwidth. In addition, our improved graph construction algorithm eliminates the inefficiencies that required the use of access vectors.

C. Configurable Granularity

Our solution can be easily configured to infer memory access ordering at multiple granularities, trading off log storage and analysis complexity for accuracy. Inferring at a cache-line granularity is the natural choice, since we track store counts per cache line. It also lets us handle multibyte reads and writes that do not cross cache lines. This approach might introduce extra dependence edges, potentially resulting in false positives but no false negatives. Even while tracking store counts per cache line, we can infer memory access ordering at a byte granularity by storing byte addresses in the access logs along with the sizes of the accesses. The analysis algorithm will then infer dependence edges only between overlapping memory accesses.

D. Multithreaded Cores and Multilevel Cache Hierarchy

Our solution can be easily extended to handle multithreaded cores by maintaining per-thread sequence identifiers in each core and attaching a thread-identifier with each memory access log entry. If there are multiple levels of private caches per core, any of the caches can be utilized for logging.

E. No-Write-Allocate L1 Caches

Our solution can be extended with some effort to support L1 caches with a no-write-allocate policy. For such caches, current store counts will not be available at the L1 for stores that miss in the L1. We can work around this issue by moving the store count update mechanism to the L2. This solution will, however, require a specialized, implementation-specific approach to handle loads that forward their values from previous stores that missed in the L1.

F. Test Quality

The detection capability of our solution depends on the quality of the stimulus; hence, our solution can only discover bugs exposed by the test programs running on the CMP. Even though, we use constrained random test cases designed to stress the memory subsystem and expose bugs, we do not address the issue of test generation in this paper.

G. Out-of-Order Core Correctness

Even though a memory consistency model allows a core to reorder memory operations from a single thread, the core must enforce data dependence ordering between memory operations. These may be direct dependencies between memory operations to the same address or indirect dependencies between memory operations to different addresses through other instructions in the thread. We notice that this requirement is a subset of the fundamental requirement for an out-of-order core to preserve the illusion of sequential execution and, therefore, we believe its verification is a well researched problem, orthogonal to the verification of shared-memory multiprocessor interactions. Our solution does not attempt to detect violations in these requirements.

H. Execution Perturbation

Theoretically, the search for consistency violations should be performed on a memory access graph constructed from the entire execution of a program. Our solution, however, breaks program execution into multiple epochs due to limited logging resources. The cores in the system drain their pipelines and complete all outstanding memory requests in between epochs. Therefore, RAW data dependencies between an epoch and all subsequent epochs cannot exist. i.e., a load in epoch i can never read a value written in epoch $i+1$. This perturbation of program execution may hinder the sensitization of certain functional errors.

I. Graph Size

The analysis phase is dominated by the topological sort algorithm for detecting cycles. The worst-case time complexity of this algorithm is $O(|V| + |E|)$, where $|V|$ represents the number of vertices and $|E|$ the number of edges [13]. Previous works have demonstrated techniques that reduce memory access graph size [2], [7], by recording only those accesses that create interthread dependence edges and inferring the transitive closure of intrathread edges. We can also utilize these optimizations to reduce graph size.

J. Verifying Coherence

While our solution can identify coherence violations that manifest as improper store serializations to a single cache line or cycles in a memory access graph, it is not a complete cache coherence verification solution. However, the mechanisms used in our solution are similar to those utilized by the CoSMa coherence verification solution [14]. Therefore, an enhanced hybrid technique that also logs cache line states and incorporates the CoSMa checking algorithm can provide a more robust memory coherence verification solution, albeit with some degradation in performance.

VII. Experimental Evaluation

A. Experimental Framework

Our experimental framework is built upon the Ruby memory system simulator in the Wisconsin Multifacet GEMS toolset [15]. We configured Ruby to simulate the memory hierarchy for a 16 core CMP using the MOESI directory-based cache coherence protocol with 32 kB, 8-way private L1 data caches, a single shared 8 MB L2 cache, and a 4 × 4 on-chip mesh interconnect. Cache lines are 64 bytes in size. We augmented the existing trace-based memory operation request driver with support for fences and intracore memory operation reordering windows. This gives us the ability to investigate the effects of memory instruction reordering, fence instructions and a range of memory consistency models. We implemented a reordering window of 16 memory operations and a constrained random reordering policy that respects the requirements of the memory consistency model. We configured our solution to track memory accesses at a cache-line granularity.

We developed a constrained-random test suite of ten multithreaded traces consisting of memory requests and fence instructions. Each test in our suite contained 1.6 million load, store and fence instructions (100 000 per thread) tuned to exhibit various data sharing characteristics as summarized in Table I. We generate special synchronization sequences that maximize sharing between the threads with the probability shown in the sync column. When such a sequence is not generated, load, store and fence instructions are produced with the percentages in the %ld, %st and %fc columns. We also control the size of our address pool and introduce false sharing in some of our tests. In addition, we modeled and probabilistically injected a total of 10 bug manifestations, described in Table II, to investigate the ability of our solution to detect bugs. The local bugs were injected in the instruction scheduling logic while the global bugs were injected in the memory subsystem.

TABLE I
Characteristics of the Evaluation Test Suite.

test	sync	%ld	%st	%fc	addr. pool	false sharing
low-sharing	0	50	50	0	100,000	none
few-writes	0.5	60	20	20	10,000	none
few-reads	0.5	20	60	20	10,000	none
synch40	0.4	60	40	0	1,000	none
false-sharing	1	-	-	-	1,000	high
fence40	0	30	30	40	10,000	none
mixed-medium	0.3	40	40	20	10,000	medium
mixed-low	0.2	30	30	40	100,000	low
synch100	1	-	-	-	1,000	none
high-sharing	1	-	-	-	10	none

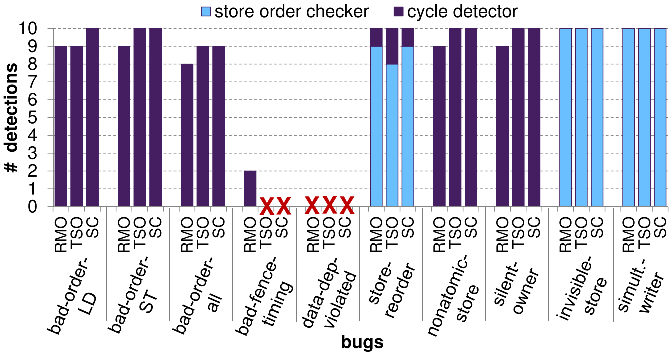

Fig. 10. Bug detections. We configured CMPs with RMO, TSO, and SC consistency models. We ran our experiments by injecting one bug at a time in each configuration and running our ten tests. We report the number of tests where the injected bug was detected. The store order checker detects bugs that manifest as incorrect store serializations. The cycle detector constructs a memory access graph and detects bugs that manifest as cycles.

TABLE II
Injected Bugs.

id	type	description
bad-order-LD	local	some intra-thread load re-ordering restrictions are violated
bad-order-ST	local	some intra-thread store re-ordering restrictions are violated
bad-order-all	local	some intra-thread load and store re-ordering restrictions are violated
bad-fence-timing	local	fences have no effect on memory operations dispatched on the same cycle
data-dep-violated	local	ordering that violates local data dependency
store-reorder	local	stores reordered in store buffer, regardless of fences
nonatomic-store	global	a store is not visible to all cores at the same time
silent-owner	global	owner of a cache line doesn't respond to GET requests
invisible-store	global	store invisible to other cores
simult.-writer	global	multiple writers to a cache line

B. Evaluating Bug Detection Capability

We investigated the capability of our mechanism to detect a sample set of local and global ordering violations. In Fig. 10, we show a summary of bug detection results for all the bugs

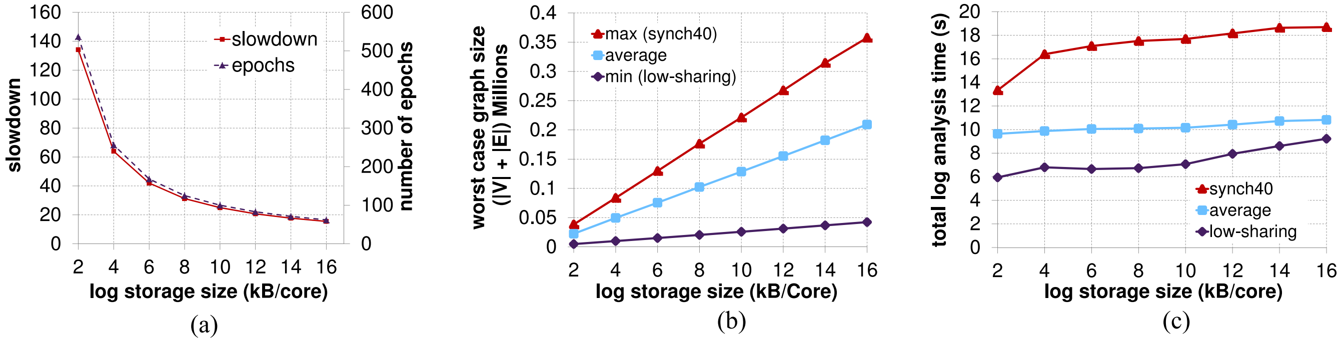

Fig. 11. Effect of increasing log storage size. (a) Number of epochs required to complete execution decreases with increasing storage. This also reduces the total execution slowdown caused by the reduced cache capacity and the phase switching overheads. (b) Sizes of the constructed access graphs appear to increase linearly with increasing log storage size. (c) Total graph analysis times (sums of analysis times for each epoch) are minimally affected by increasing log size.

and memory consistency models investigated. In these experiments, our solution was configured to use 16 kB per core for log storage. Each bar in the graph indicates the number of tests, out of a total of 10, that exposed a bug manifestation which was detected either as an illegal store ordering (shown in light blue) or as a cycle in the memory access graph (shown in purple). Red cross marks are used for bugs that were not detected by our solution. Note that our solution constructs a graph and performs cycle detection only if a bug is not detected by the store order checker, as illustrated in Fig. 7.

We observe that, with the exception of two bugs, our solution detects the injected bugs for more than 80% of the test cases. For the bad-fence-timing bug to manifest, a memory operation must be dispatched on the same cycle as a fence instruction restricting its ordering. This bug is a rarely occurring event in our RMO configurations. Our TSO and SC configurations are not affected, since ordering rules are implicitly enforced without the need for fence instructions. Note that the bad-order-* bugs affect both the implicit ordering constraints in TSO and SC, and the explicit fence constraints in RMO. As discussed in Section VI, our solution does not address local data dependency violations. Therefore, the data-dep-violated bug, which affects intrathread data dependencies, is never detected by our system. For the remaining cases, the bug manifestation did not create a cycle in the memory access graph for some of the test cases. This situation arises when none of the other cores execute memory instructions that allow them to observe the wrong ordering in the affected core.

Our store order checker is able to detect global ordering violations that result in illegal serialization of stores to a single cache line. It, therefore, misses the nonatomic-store and silent-owner bugs that do not manifest as incorrect store serializations. However, these bugs still produce cycles in the memory access graph.

C. Evaluating Log Storage Sizes

We designed our system so that a single log entry for a memory operation requires 10 bytes (16 bits for the store counter, 20 bits for the sequence identifier, 2 bits for the request type, 42 bits for the physical line address). This configuration was found to be more than sufficient even for our most demanding tests. To study the impact of log storage size, we created 8 L1 data cache configurations, simulating 1–8 ways dedicated for log storage. The data arrays for a single cache way provide us with up to 4 kB storage, allowing us to store up to 409 entries per cache way. For each configuration, test-case and consistency model combination, we ran two experiments allowing us to investigate log sizes ranging from 2 to 16 kB per core. The logs collected were then analyzed by our single-threaded analysis algorithm running on a host with a 2.80 GHz Intel Core i7-930 CPU and a 1066 MHz DDR3 memory.

Fig. 11 summarizes the impacts and computation demands of our solution as a function of log storage size, for our RMO configurations. In Fig. 11(a), we observe the impact of log storage size on the total test execution time, for the test case exhibiting the worst-case execution time. Note that the total execution time here is the sum of the test program execution times for all epochs and does not account for the time required to aggregate and analyze the logs. We present the slowdown in execution time relative to test execution on a configuration where our solution is completely disabled. This slowdown is due to the following reasons: 1) by reserving a portion of the L1 data caches for log storage, we reduce the effective cache capacity that the test program can utilize, potentially resulting in increased L1 capacity misses. We do not observe any significant changes in miss rates for the test programs in our experiments and 2) at the end of each program execution phase in an epoch, the system waits for all pending operations to drain before proceeding to the next phase. This per-epoch drain time is an overhead that is independent of log storage size and thus remains almost constant across all epochs and configurations. The total overhead for an execution is the sum of these drain times and increases with increasing number of epochs. Note that the number of epochs is a function of log storage size—the larger our log storage size, the fewer our epochs.

For each test program, we identified the largest observed graph size under the different log storage configurations; graph size is the sum of the number of vertices and edges. We observed that the worst-case graph size increases roughly linearly with increasing log storage size for all test programs. The synch40 test program exhibits the maximum worst-case graph size for all log storage sizes while the low-sharing test program

TABLE III
MEMORY ACCESS GRAPH CASE STUDY.

core	vertices	local edges	global edges
0	1620	57,007	5019
1	1624	17,086	2421
2	1624	15,541	2232
3	1621	15,864	2225
4	1621	16,342	2340
5	1622	15,331	2220
6	1623	15,505	2270
7	1614	13,935	2087
8	1623	13,719	2099
9	1620	13,023	2025
10	1618	13,786	2062
11	1619	14,373	2135
12	1618	13,974	2145
13	1617	16,332	2282
14	1622	12,764	1975
15	1618	14,035	2117
Total	**26,563**	**278,617**	**37,654**

exhibits the minimum. Fig. 11(b) presents the results for the average of all test cases, synch40 and low-sharing. It is worth noting that the per-epoch graph size is also dependent on the consistency model, the frequency of fence instructions (if any), and the amount of data sharing between the multiple interacting threads. These results are from multiple synthetic tests of varying characteristics running on a configuration with the RMO consistency model. Real-world applications would generate far smaller graphs due to limited interthread interactions and far fewer fence instructions.

Fig. 11(c) summarizes the sum of per-epoch graph analysis times for synch40, low-sharing, and the average of all test programs. Even though a larger log storage size results in larger graphs—hence larger graph analysis times per epoch—there are fewer epochs to analyze. Therefore, the total analysis time changes fairly slowly with increasing log storage size. In fact, the maximum increase we observe for an 800% increase in per-core log storage size (from 2 to 16 kB) is 55% for the low-sharing test program.

Table III provides a detailed look into a memory access graph constructed from one execution epoch of the synch40 test program. The system was configured with the RMO consistency model, 16 kB of per-core log storage and had the bad-order-LD bug injected. The resulting graph had a total of 26 563 vertices and 316 271 edges. The bug manifested as a cycle in this graph spanning only four memory accesses from two cores, reading and writing to two memory locations.

D. Hardware Overheads

Our solution adds a modest amount of hardware overhead. This overhead is mainly due to the extra storage required to track information for in-flight memory operations. A 20 bit sequence identifier register and its associated retirement register are added to each core. Each entry in the load and store queues is increased by 20 bits to hold the sequence identifier for in-flight memory operations. Assuming a size of 32 entries for both the load and store queues, the total overhead due to this augmentation becomes 160 bytes. The log-delay buffer needs to buffer 10 bytes of information per entry. If we design our log-delay buffer to hold as many entries as the load queue, the store queue, and the post-retirement store buffer (assumed to be 16 entries), we have an overhead of 800 bytes. The overall per-core storage overhead of our solution is then 965 bytes, which is less than 72% of the size of the Intel Haswell physical register file [16].

The hardware modifications we introduce are not on the critical computation paths, as all of the control logic we add operates in parallel with the normal processor paths. Each of our additions interferes with normal processor paths in two places: 1) when sampling signals from the normal paths and 2) when driving multiplexers that choose between our control/data and normal processor control/data. For the latter, we would add at most two NAND gates (approximately a 50 ps delay at 22 nm) to the normal processor path, which is hardly sufficient to make a noncritical path critical. The former can increase capacitive loading on the outputs of some gates. However, we can use sleep transistors to disconnect our solution from the power rails, effectively eliminating the capacitive load during normal operation.

VIII. RELATED WORK

When verifying a shared-memory multiprocessor, a precise understanding of its consistency model is required. Lamport formalized SC in [5]. Since then, several consistency models have been proposed with the intention of allowing optimized hardware implementations and/or improving the programmability of shared-memory multiprocessors [4], [17]. Several works attempt to analyze and formalize various memory consistency models implemented by modern shared-memory multiprocessors [10], [18]–[20]. Special programs known as “litmus tests” have been used to systematically compare and contrast between multiple memory consistency models [21], [22].

A directed graph constructed from the dynamic instruction instances in a multithreaded execution can be used to analyze the performance and correctness of executions [3], [9]. Researchers have proposed verification solutions that leverage the property that an ordering violation manifests as a cycle in such a graph. TSOtool [8] is a software-based approach that constructs a graph from program order and data dependencies captured from pseudo-randomly generated test programs. To capture data dependencies, these programs are generated with unique store values and special code to observe loaded values. Roy *et al.* [11] used an approach similar to TSOtool. Chen *et al.* [2] augmented a shared-memory multiprocessor with hardware to capture and validate memory operation ordering. DACOTA [7] is post-silicon solution that captures memory operation ordering in hardware by repurposing existing resources and performs graph construction and analysis using software running on the multiprocessor under verification.

Other researchers have proposed solutions that do not rely on graph analysis to verify the correctness of shared-memory multiprocessors. Lin *et al.* [23] inserted operations in multithreaded test programs that check the correctness of values loaded from memory. Meixner and Sorin [24] conquered the memory consistency verification challenge by identifying three invariants and designing hardware checkers for each.

IX. CONCLUSION

This paper presents a novel shared-memory interaction verification solution for accelerated (emulated) simulation and post-silicon validation. When enabled, our solution tracks information about issued memory operations and fence instructions in hardware, periodically aggregating this information to perform a software-based analysis. The analysis algorithm is implemented purely in software and can be run anywhere, giving the verification team the flexibility of choosing the most efficient alternative. In addition to its effectiveness in detecting subtle shared-memory interaction ordering violations, our solution can enhance the debugging effort through the information it collects. The hardware components of our solution can be completely disabled before shipment, leaving no impact on the end user.

REFERENCES

[1] D. Sorin, M. Hill, and D. Wood, "A primer on memory consistency and cache coherence," in *Synthesis Lectures on Computer Architecture*. San Rafael, CA, USA: Morgan Claypool, 2011.

[2] K. Chen, S. Malik, and P. Patra, "Runtime validation of memory ordering using constraint graph checking," in *Proc. Int. Symp. High Perform. Comput. Archit. (HPCA)*, Salt Lake City, UT, USA, 2008, pp. 415–426.

[3] D. Shasha and M. Snir, "Efficient and correct execution of parallel programs that share memory," *ACM Trans. Program. Lang. Syst.*, vol. 10, no. 2, pp. 282–312, Apr. 1988.

[4] S. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *IEEE Comput.*, vol. 29, no. 12, pp. 66–76, Dec. 1996.

[5] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Trans. Comput.*, vol. C-28, no. 9, pp. 690–691, Sep. 1979.

[6] D. L. Weaver and T. Germond, Eds., *The SPARC Architecture Manual (Version 9)*. Upper Saddle River, NJ, USA: Prentice-Hall, 1994.

[7] A. DeOrio, I. Wagner, and V. Bertacco, "DACOTA: Post-silicon validation of the memory subsystem in multi-core designs," in *Proc. Int. Symp. High Perform. Comput. Archit. (HPCA)*, Raleigh, NC, USA, 2009, pp. 405–416.

[8] S. Hangal, D. Vahia, C. Manovit, and J.-Y. Lu, "TSOtool: A program for verifying memory systems using the memory consistency model," in *Proc. Annu. Int. Symp. Comput. Archit. (ISCA)*, Munich, Germany, 2004, pp. 114–123.

[9] H. Cain, M. Lipasti, and R. Nair, "Constraint graph analysis of multithreaded programs," in *Proc. 12th Int. Conf. Parallel Archit. Compil. Technol. (PACT)*, New Orleans, LA, USA, 2003, pp. 4–14.

[10] A. Arvind and J. W. Maessen, "Memory model = instruction reordering + store atomicity," *ACM SIGARCH Comput. Archit. News*, vol. 34, no. 2, pp. 29–40, May 2006.

[11] A. Roy, S. Zeisset, C. J. Fleckenstein, and J. C. Huang, "Fast and generalized polynomial time memory consistency verification," in *Proc. 18th Int. Conf. Comput.-Aided Verif. (CAV)*, Seattle, WA, USA, 2006, pp. 503–516.

[12] R. Kalayappan and S. Sarangi, "A survey of checker architectures," *ACM Comput. Surv.*, vol. 45, no. 4, Aug. 2013, Art. ID 48.

[13] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. Cambridge, MA, USA: MIT Press, 2009.

[14] A. DeOrio, A. Bauserman, and V. Bertacco, "Post-silicon verification for cache coherence," in *Proc. IEEE Int. Conf. Comput. Design (ICCD)*, Lake Tahoe, CA, USA, 2008, pp. 348–355.

[15] M. Martin *et al.*, "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset," *ACM SIGARCH Comput. Archit. News*, vol. 33, no. 4, pp. 92–99, Nov. 2005.

[16] P. Hammarlund *et al.*, "Haswell: The 4th generation Intel core processor," *IEEE Micro*, vol. 34, no. 2, pp. 6–20, Mar./Apr. 2014.

[17] D. Marino, A. Singh, T. Millstein, M. Musuvathi, and S. Narayanasamy, "DRFX: A simple and efficient memory model for concurrent programming languages," in *Proc. Program. Lang. Design Implement. (PLDI)*, Toronto, ON, Canada, 2010, pp. 351–362.

[18] P. Sewell, S. Sarkar, S. Owens, F. Nardelli, and M. Myreen, "X86-TSO: A rigorous and usable programmer's model for x86 multiprocessors," *Commun. ACM*, vol. 53, no. 7, pp. 89–97, Jul. 2010.

[19] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams, "Understanding POWER multiprocessors," in *Proc. Program. Lang. Design Implement. (PLDI)*, San Jose, CA, USA, 2011, pp. 175–186.

[20] J. Alglave *et al.*, "The semantics of power and ARM multiprocessor machine code," in *Proc. Declarat. Aspects Multicore Program. (DAMP)*, Savannah, GA, USA, 2008, pp. 13–24.

[21] S. Mador-Haim, R. Alur, and M. Martin, "Generating litmus tests for contrasting memory consistency models," in *Proc. 22nd Int. Conf. Comput.-Aided Verif. (CAV)*, Edinburgh, Scotland, 2010, pp. 273–287.

[22] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell, "Litmus: Running tests against hardware," in *Proc. 17th Int. Conf. Tools Algorithms Construct. Anal. Syst. (TACAS)*, Saarbrücken, Germany, 2011, pp. 41–44.

[23] D. Lin, T. Hong, F. Fallah, N. Hakim, and S. Mitra, "Quick detection of difficult bugs for effective post-silicon validation," in *Proc. Design Autom. Conf. (DAC)*, San Francisco, CA, USA, 2012, pp. 561–566.

[24] A. Meixner and D. Sorin, "Dynamic verification of memory consistency in cache-coherent multithreaded computer architectures," in *Proc. Int. Conf. Depend. Syst. Netw. (DSN)*, Philadelphia, PA, USA, 2006, pp. 73–82.

Biruk W. Mammo (S'13) received the B.Sc. degree in electrical and computer engineering from Addis Ababa University, Addis Ababa, Ethiopia, in 2007, and the M.S. degree from the University of Michigan, Ann Arbor, MI, USA, in 2012, where he is currently pursing the Ph.D. degree, both in computer science and engineering.

His current research interests include microarchitecture design and validation, fault modeling and analysis, and robust system design.

Valeria Bertacco (S'95–M'03–SM'10) received the Laurea degree in computer engineering from the University of Padua, Padua, Italy, and the M.S. and Ph.D. degrees in electrical engineering from Stanford University, Stanford, CA, USA, in 2003.

She was at Synopsys, Mountain View, CA, USA, for four years. She is currently an Associate Professor of Electrical Engineering and Computer Science with the University of Michigan, Ann Arbor, MI, USA. Her current research interests include complete design validation, digital system reliability, and hardware-security assurance.

Prof. Bertacco was the recipient of the IEEE CEDA Early Career Award and served on the program committees of DAC, DATE, and MICRO.

Andrew DeOrio (S'07–M'12) received the B.S.E. and M.S.E. degrees in electrical engineering, and the Ph.D. degree in computer science and engineering from the University of Michigan, Ann Arbor, MI, USA, in 2006, 2008, and 2012, respectively.

He is currently a Lecturer with the University of Michigan. His current research interests include ensuring the correctness of digital hardware designs, including verification, reliable system design, and post-silicon validation.

Ilya Wagner received the M.S. and Ph.D. degrees in computer engineering from the University of Michigan, Ann Arbor, MI, USA, in 2006 and 2008, respectively.

He was at Intel's Oregon Microarchitecture Laboratory, Hillsboro, OR, USA, research in system-on-chip survivability features and memory error recovery. He is a Research Scientist with the System Validation Engineering Group, Intel Corporation. His current research interests include next-generation silicon and firmware debug tools.

Dr. Wagner served on different capacity on several program committees for the IWLS Symposium from 2011 to 2013.