On ESL Verification of Memory Consistency for System-on-Chip Multiprocessing

Eberle A. Rambo, Olav P. Henschel, Luiz C. V. dos Santos
Computer Sciences Department, Federal University of Santa Catarina
Florianopolis, Brazil
{eberle18, olav.ph, santos}@inf.ufsc.br

***Abstract*—Chip multiprocessing is key to Mobile and high-end Embedded Computing. It requires sophisticated multilevel hierarchies where private and shared caches coexist. It relies on hardware support to implicitly manage relaxed program order and write atomicity so as to provide well-defined shared-memory semantics (captured by the axioms of a memory consistency model) at the hardware-software interface. This paper addresses the problem of checking if an executable representation of the memory system complies with a specified consistency model. Conventional verification techniques encode the axioms as edges of a single directed graph, infer extra edges from memory traces, and indicate an error when a cycle is detected. Unlike them, we propose a novel technique that decomposes the verification problem into multiple instances of an extended bipartite graph matching problem. Since the decomposition was judiciously designed to induce independent instances, the target problem can be solved by a parallel verification algorithm. Our technique, which is proven to be complete for several memory consistency models, outperformed a conventional checker for a suite of 2400 randomly-generated use cases. On average, it found a higher percentage of faults (90%) as compared to that checker (69%) and did it, on average, 272 times faster.**

I. Introduction

In Mobile and high-end Embedded Computing, *chip multiprocessing* (CMP) became the key to energy-efficient systems-on-chip, such as those supported by ARM Cortex-A and MIPS 1074K families. CMP requires sophisticated multilevel memory hierarchies where private and shared caches coexist. Parallel programs are affected by side effects of the memory system. For instance, two successive memory accesses from different threads to the same location will lead to a *data race* if at least one of them writes to that location. Since data races would induce distinct behaviors for the same parallel program, they must be ruled out by means of a *synchronization mechanism* (e.g. a lock). If the order of memory accesses to distinct locations is relaxed, synchronization may not work as intended [1]. If the program order is enforced, performance is limited. Therefore, the memory subsystem must provide proper hardware support for order relaxation. For enabling parallel programs to exploit relaxation without impairing synchronization, it is crucial to define consistent shared-memory semantics at the hardware-software interface. This leads to the notion of *memory consistency model* (MCM), which defines when the writes of a processor are observed by other processors. Several MCMs are reported in the literature [1], such as Sequential Consistency (SC), Release Consistency (RC), Total Store Order (TSO), Alpha Relaxed Order (ARO), Relaxed Memory Order (RMO), Weak Ordering (WO), etc.

An MCM is formally defined by means of axioms. The problem of verifying if the shared-memory hardware complies with an MCM is crucial to parallel programming. Several techniques have been proposed to solve that problem [2] [3] [4]. They are all based on observing local traces and checking if there is a global trace that satisfies the axioms of a given MCM. Due to the limited observability of the hardware, such a black-box verification problem is rather complex [5].

We propose a novel technique operating at the *Electronic System Level* (ESL). It exploits the extended observability of an *executable representation* of the shared-memory subsystem to reduce the computational effort of verifying, at early phases of the design flow, if it actually implements a given MCM. It decomposes the verification problem into multiple instances of an extended bipartite graph matching problem. Since the decomposition was judiciously designed to induce independent instances, the target problem can be solved by a parallel verification algorithm. For the MCMs not requiring total store ordering (ARO, RMO, RC, WO, etc.), the proposed technique is provenly *complete* when analyzing the behavior induced by a given test case: it neither overlooks actual errors nor flags apparent errors. The technique was experimentally validated with 2400 use cases and was compared to a conventional checker. On average, our technique found a higher percentage of faults (90%) as compared to that checker (69%) and did it, on average, 272 times faster.

The remaining of this paper is organized as follows. Section II describes a template that can accommodate a diversity of CMP architectures. The target verification problem is formalized in Section III. In Section IV, we show how conventional approaches address that problem. Section V proposes a problem decomposition that is largely architecture independent. It also presents formal proofs to support our claim to completeness. Section VI experimentally compares the novel technique to a conventional checker. In Section VII, we draw our overall conclusions and sketch future work directions.

This work was partially supported by a grant from CAPES (a Brazilian Research Agency) and by the Brazilian Council for Scientific and Technological Development (CNPq) through grants 573738/2008-4 (INCT NAMITEC), 306654/2009-1 (PQ), and 559882/2010-6 (CTINFO-PDI).

978-3-9810801-8-6/DATE12/©2012 EDAA

II. An Architecture Template

To pinpoint relevant observation points, Figure 1 shows a microarchitecture template for a generic processing element of a CMP architecture (adapted from [6]). Remind that, to support speculation and to keep precise exceptions, most processors commit instructions in the same order as they were issued [6]. That is why we assume that, given a representation of a CMP architecture, events can be monitored in program order at point i^+. When observed at point i^-, events are monitored in the order they were effectively performed on shared memory.

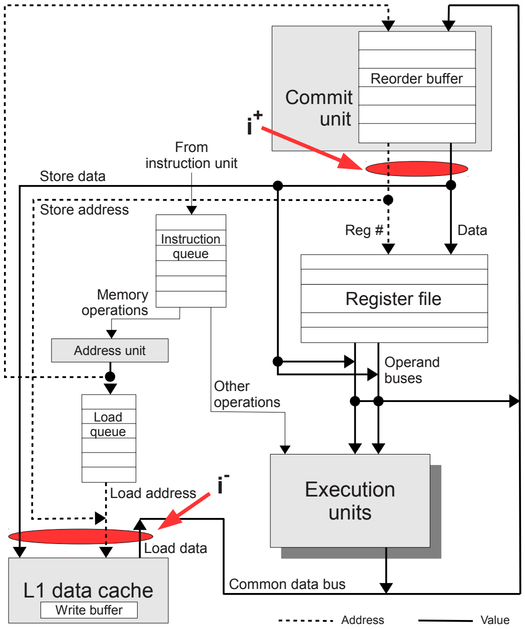

Fig. 1. Microarchitecture of a processing element

III. The Verification Problem

We rely on the following notation. $\mathcal{O}$ is the set of memory operations issued by all processors and $\mathcal{S}$ is the set of all stores. A is the set of all addresses referenced by operations in $\mathcal{O}$. $\mathcal{O}^i \subset \mathcal{O}$ and $\mathcal{S}^i \subset \mathcal{S}$ denote operations issued by processor i. We let p and n denote, respectively, the number of processors and the overall number of operations (i.e. $n = |\mathcal{O}|$). To specify that an arbitrary operation is issued by a processor i and makes a reference to an address $a \in A$, we write $(Op_j)_a^i$. We replace Op by L or S to specify that the operation is either a load or a store. We drop the subscript j in shorthand notation.

Let us first address two orderings of memory events that are *observable* and then an ordering that can only be *inferred*.

Definition 1 - **Local order induced by the program:** We say that $(Op_1)^i$ precedes $(Op_2)^i$ in program order, written $(Op_1)^i <_+ (Op_2)^i$, iff $(Op_1)^i$ reaches the head of processor i's instruction queue before $(Op_2)^i$ does.

Definition 2 - **Local order induced by the execution:** We say that $(Op_1)^i$ precedes $(Op_2)^i$ in execution order, written $(Op_1)^i <_- (Op_2)^i$, iff the former's completion is acknowledged to the private data-cache controller of processor i before the latter's.

Definition 3 - **Global order:** We say that $(Op_1)^i$ precedes $(Op_2)^k$ in global order, written $(Op_1)^i \leq (Op_2)^k$, iff $(Op_1)^i$ completes its writing or reading on the shared memory subsystem before $(Op_2)^k$ does.

The order $<_+$ ($<_-$) can be monitored at point i^+ (i^-). The order $\leq$ is *specified* by the axioms of a given MCM.

Several MCMs, such as ARO, RMO, SC, TSO and WO [1] require the serialization of stores to the same location.

Property 1 - **Cache coherence constraint:** $\forall S_a^i, S_a^k \in \mathcal{S} : (S_a^i \leq S_a^k) \vee (S_a^k \leq S_a^i)$.

MCMs are distinguished by their degree of relaxation of program order and by their write atomicity requirements. For simplicity, we illustrate the *specific* requirements of only two MCMs: TSO [2] and ARO [7]. TSO states that the execution order in a given processor must be the program order, except that loads may overtake stores if they make references to distinct locations. ARO allows the order of all local operations to be relaxed, except for operations to the same address.

Axiom 1 - **Relaxation of program order:**
$(L_a^i <_+ Op_b^i \Rightarrow L_a^i \leq Op_b^i) \vee (S_a^i <_+ S_b^i \Rightarrow S_a^i \leq S_b^i)$. (TSO)
$(Op_1)_a^i <_+ (Op_2)_a^i \Rightarrow (Op_1)_a^i \leq (Op_2)_a^i$. (ARO)

TSO specifies that all processors must observe the same linear order of stores. In contrast, ARO only requires the stores to the same location to be linearly ordered (Property 1).

Axiom 2 - **Write atomicity constraint:**
$\forall S_a^i, S_b^k \in \mathcal{S} : (S_a^i \leq S_b^k) \vee (S_b^k \leq S_a^i)$. (TSO)
$\forall S_a^i, S_a^k \in \mathcal{S} : (S_a^i \leq S_a^k) \vee (S_a^k \leq S_a^i)$. (ARO)

Most MCMs (SC, TSO, WO, RC, ARO, RMO, etc.) capture a processor's inner property:

Property 2 - **Read own write early:** The store queue of each processor i is allowed to bypass data from an outstanding store S_a^i to a load L_a^i, before S_a^i becomes globally visible.

Let us formulate the impact of Property 2 on a load L_a^i.

Definition 4 - **Local producer:** the operation $S_a^i \in \{S_a^i | S_a^i <_+ L_a^i\}$ such that $\neg(\exists {S'}_a^i | S_a^i <_+ {S'}_a^i <_+ L_a^i)$, written $Max_{<_+}[\{S_a^i | S_a^i <_+ L_a^i\}]$.

Let us define the impact of global order on a load L_a^i:

Definition 5 - **Global producer:** the operation $S_a^k \in \{S_a^k | S_a^k \leq L_a^i\}$ such that $\neg(\exists S_a^j | S_a^k \leq S_a^j \leq L_a^i)$, written $Max_{\leq}[\{S_a^k | S_a^k \leq L_a^i\}]$.

Let $Val[L_a^i]$ denote the value returned by a load and let $Val[S_a^i]$ be the value written by a store. For every address, an MCM requires that, among the potential producers for a load, either in global order or program order, the value returned by that load is the value written by the latest store [2]:

Axiom 3 - **Uniqueness of returned value:**

$$\forall a \in A : Val[L_a^i] = \begin{cases} Val[Max_{<_+}[\{S_a^i | S_a^i <_+ L_a^i\}] & \text{if Prop. 2} \\ Val[Max_{\leq}[\{S_a^k | S_a^k \leq L_a^i\}] & \text{otherwise} \end{cases}$$

Supplementary axioms address atomic swaps (X) and memory barriers (M) [8]. We assume (without loss of generality) that such supplementary axioms are merged, for verification purposes, into Axioms 1, 2, and 3.

The following necessary condition for memory consistency (despite being obvious) is a useful property for early fault detection.

Property 3 - **Global visibility:** Except for the loads satisfying Property 2, which never reach the memory, all other operations in the program must be observed between their issuing processor and its interface with the memory system.

Let us formalize the key notion that enables MCM checking:

Definition 6 - **Trace:** a sequence $(\tau_1, \tau_2, \cdots \tau_j, \cdots \tau_m)$, where $\tau_j = (op, a, v)$ is a memory event such that $op \in \{L, S, X, M\}$, $a \in A$ and $v = Val[op_a]$ when $op \neq M$.

Since a memory barrier does not produce nor consume a value, we define $Val[M] = NIL$.

Finally, the target problem can be formulated:

Problem 1 - **MCM Verification:** Given a collection of traces $T_1, T_2, \cdots T_p$, is there a global trace T satisfying all the MCM's order and value axioms?

IV. Related Work

Most verification approaches [2] [9] [4] rely on automatically generating multithreaded random instruction tests (RITs). Such synthetic programs have one thread per processor executing a sequence of operations that make references to shared addresses. Both the sequence and the references are generated pseudo-randomly as a way to provide proper coverage.

Conventional checkers [2]–[4], [8], [9] solve instances of Problem 1 by encoding axioms into a *constraint graph*. They make successive inferences on the order of operations by inserting edges in the constraint graph until a cycle is detected or no more inferences can be made. Since only a directed *acyclic* graph (DAG) can represent an order relation, the detection of a cycle in the constraint graph is a proof of memory inconsistency. However, an acyclic constraint graph does not necessarily prove memory consistency, because some existing order relation between operations might not have been inferred [8]. Therefore, a checker relying on such an incomplete graph may raise false negative outcomes. The reader should bear in mind that, in the context of checkers stimulated by RITs, *completeness* is usually defined from the perspective of a given test-case [4], [8].

TSOTool [2] enforces RIT generation so that unique values are written to different addresses. The analysis takes $O(n^5)$ in the worst case. A refinement based on more efficient heuristics [3] reduced that worst-case complexity to $O(pn^3)$. However, the algorithms proposed in [2] and [3] are incomplete, i.e. they may induce false negatives. That is why a backtracking algorithm was later proposed [8]. As a price to pay for completeness, it takes $O((n/p)^p n^3)$ in the worst case.

A more comprehensive technique [9] generalized the key ideas from [2] to handle processors implementing several MCMs. The technique's execution time was improved by using incremental graph closure and parallelization. The final algorithm takes time $O(n^4)$ in the worst case.

LCHECK [4] relies on execution intervals of instructions to add timing constraints to the analysis. The technique requires store atomicity and extra observability for verification and testing. Since the extra observability and additional constraints allow for local checking, the complexity can be reduced to $O(C^p p^2 n^2)$ for complete verification, where C is a constant.

Extended bipartite graph matching (E-matching) [10] addresses the functional verification of IPs in ESL design. It relies on a bipartite graph whose partitions represent events monitored at equivalent points of distinct executable representations and whose edges represent the compatibility of the observed values. As a consequence, the nonexistence of a matching indicates a fault in the device under verification. However, as opposed to conventional matching, the vertices within an E-matching must also satisfy a vertex-ordering constraint specified by a relation R.

Despite being originally targeted to IP verification, we envisioned E-matching [10] as a promising mechanism for MCM checking for the following reasons. Unlike conventional techniques, E-matching does not depend on inferences (which are not always successful [8]). Instead of relying on pairwise analysis of value production and consumption as a way to detect an ordered pair, E-matching analyzes the overall consequences of out-of-order execution on all-pairs of values. Those facts motivated us to design a decomposition of Problem 1 leading to p *independent* E-matching instances.

V. The Proposed Decomposition and Casting

Figure 2 illustrates our technique as compared to a conventional checker. Suppose that we want to verify if a system with two processors (P1 and P2) complies with the ARO model, by analyzing their traces. Assume that traces capture execution sequences that were affected by a fault in the coherence engine (a value written to a local cache was not propagated to another). The arrows labeled as $<_t$ indicate the order in which the events were monitored. For each processor, our technique builds a bipartite verification graph (BVG), whose vertices represent monitored events and whose edges represent event equivalence. The dotted arrows (which are *not* BVG edges but elements of an order relation R) represent the ordering constraints specified by Axioms 1 and 2. Note that edge crossings in a BVG indicate that operations observed at the interface with the memory system (i^-) are out of program order (i^+). In Step 1, each BVG is submitted to the E-matching algorithm, which tries to find a complete matching that does not induce edge crossings if the involved operations are ordered by the MCM specification. If such a *proper* matching [10] is not found, this indicates that either the order relation is not satisfied or that operations are missing at the memory interface. Note that, for P1, there is a single matching (which turns out to be proper). For P2, however, there are two possible matchings. Notice that the order inversion depicted by the dashed edges violates ARO's order constraints and induces an improper matching. That is why the E-matching algorithm *prunes* those edges. It turns out that the remaining edges lead to a proper matching. As a result, no errors were found from the perspective of intra-processor behavior (since the fault is exposed through inter-processor behavior). In Step 2, a global trace (T_G) is built according to the order of timestamps ($<_t$). While scanning the global trace, the current status of each memory location is kept in a table (V), which is updated on every store event and checked on every load event. Note that,

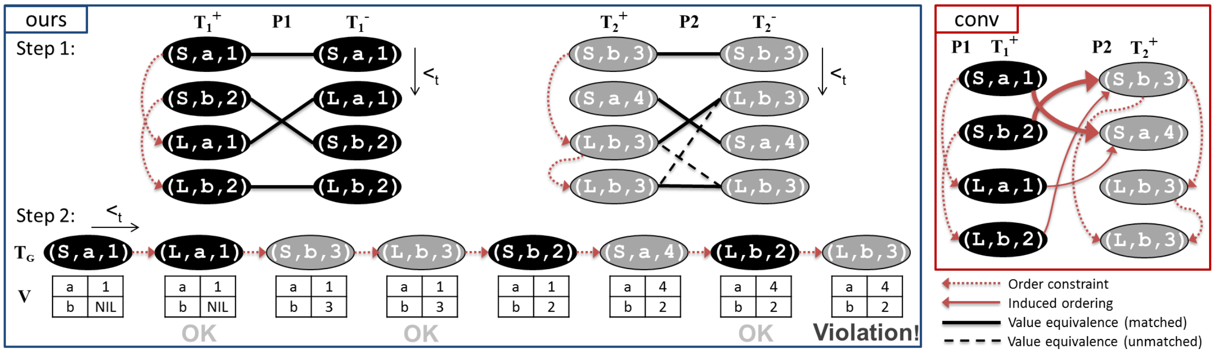

Fig. 2. Illustrative example for the proposed technique as compared to a conventional checker

for the first three loads, the value read by the load matches the value written by the last store to the same address. However, a mismatch occurs for the last load, exposing the wrong inter-processor behavior induced by the coherence fault.

From the initial set of program-order constraints, the conventional checker tries to infer new order relations. One of the mechanisms that enables inference is value consumption. Note that the last load issued by P2 should consume the value produced by the last store issued by P1. However, since the coherence fault prevented value propagation between processors, the inference mechanism fails to detect their relation. To overcome such limitation, some conventional checkers may rely on backtracking [8]. However, even at the price of higher computational effort, a conventional checker is unable to find the fault. It eventually finds an order that validates the execution: for instance, if the order indicated by the thick arrows in Figure 2 is adopted as tie breaker, the order shown by the thin arrows will be inferred. As the resulting graph is a DAG, the conventional checker asserts the correctness of an execution that is indeed incorrect.

A. Step 1: Verifying local behavior

From the traces T_i^+ and T_i^- monitored at points i^+ and i^-, Algorithm 1 builds a *bipartite verification graph* BVG(V, E) and tries to match its partitions V^+ and V^-. It first verifies if Axiom 3 holds for local producers (line 2). When it does, the loads consuming values from local outstanding stores are excluded from V^+ (line 4), since they do not reach the memory system. This guarantees that every event in V^+ has at least one corresponding event in V^-. Then it does the E-matching's casting (lines 4–9). The algorithm proposed by [10] was employed (at line 10) to locally check memory consistency. E-matching takes time $O(|E|^3)$ [10], where $|E|$ is the number of edges in the BVG. When using an RIT generator that enforces a uniform distribution of operations between processors, we have $|E| = O(n^2/p^2)$. Therefore, E-matching takes time $O(n^6/p^6)$ in the worst case to verify the local behavior of a single processor. However, the average complexity tends to be much smaller (see Section VI).

Lemma 1 - Algorithm 1 returns true iff Axioms 1, 2, and 3, as well as Property 3, hold for $\mathcal{O}^i$.

Proof: Since `read-own-write-ok` holds when Algorithm 1 returns true, we conclude that Axiom 3 certainly holds for local producers. Given two operations x and y, let $ax_1(x, y)$ and $ax_2(x, y)$ be the predicates specified by Axioms 1 and 2 for a generic MCM. Let $R_1^i = \{(x, y) \in \mathcal{O}^i \times \mathcal{O}^i | ax_1(x, y)\}$ and $R_2^i = \{(x, y) \in \mathcal{S}^i \times \mathcal{S}^i | ax_2(x, y)\}$, i.e. the sets of ordered pairs satisfying Axioms 1 and 2, respectively. Algorithm 1 (at line 9) assigns to R^i all the elements in $R_1^i \cup R_2^i$. Besides, all the equivalent events in traces T_i^+ and T_i^- are assigned to the relation E (at line 5). Two theorems in [10] guarantee that, when E is an equivalence relation, `proper-matching` returns true iff two conditions hold simultaneously: 1) R^i is satisfied for all events monitored at i^-; 2) a one-to-one mapping $M \subseteq E$ is found. Since E is indeed an equivalence relation, the first condition proves that Axioms 1 and 2 hold for the operations issued by processor i. The second condition proves that Property 3 holds for processor i. ■

Algorithm 1: `local-behavior-OK(MCM,` T_i^+, T_i^-`)`

```
1  let ≤ be the order specified by the MCM;
2  if ¬ read-own-write-ok($T_i^+$) then
3      return false;
4  $V^+ = T_i^+ -$ local-consumers($T_i^+$);
5  $V^- = T_i^-$;
6  $V = V^+ \cup V^-$;
7  $E = \{(v^+, v^-) \in V^+ \times V^- | \ v^+ = v^-\}$;
8  $\mathcal{O}^i = \{op \ | \ (op, a, v) \in V^+\}$;
9  $R^i = \{(op_1, op_2) \in \mathcal{O}^i \times \mathcal{O}^i \ | \ op_1 \leq op_2\}$;
10 return proper-matching($R^i, V, E$);
```

B. Step 2: Verifying global behavior

We assume that a timestamp $t(\tau_j)$ is assigned to every event τ_j when it is monitored. As a result, when the simulation completes, all the events have a timestamp. An operation x captured by an event τ_j has timestamp $t(x) = t(\tau_j)$.

Definition 7 - **Linear order induced by timestamping:** Let $\ll$ be an arbitrary linear ordering on the set $\mathcal{O}$. The order $<_t$ is defined as follows:

$$\forall x, y \in \mathcal{O} : x <_t y \Leftrightarrow (t(x) < t(y)) \vee (t(x) = t(y) \wedge x \ll y).$$

A global trace is built by sorting events in the order $<_t$. It takes time $O(n \log p)$ in the worst case, by using a heap and limiting to p the number of simultaneously enqueued events.

Algorithm 2 checks for consistent value consumption from a global trace. It keeps a hash table (lines 1–2) that records the observed values. After building a global trace (line 3), it evaluates the set of addresses referenced by the events in that trace (line 4). Then it visits events in the order $<_t$ (lines 7–13). When a store is visited (line 9), its value is recorded in the hash table (line 10). When a load is visited (line 11), the algorithm checks if the value it consumes matches the last value produced by a store to the same address (Axiom 3). It returns false as soon as a value mismatch is detected. As it does $O(n)$ accesses to the hash table and each access takes $O(1)$, Algorithm 2 is dominated by the building of a global trace, i.e. it takes time $O(n \log p)$ in the worst case.

Algorithm 2: global-behavior-ok(T_1^-, T_2^-, $\cdots$, T_p^-)

```
1  let $V$ be a hash table;
2  let $h : A \rightarrow \{1, 2, \cdots, |A|\}$ be a hash function;
3  $T_G$ = $(\tau_1, \tau_2, \cdots \tau_j, \cdots \tau_n)$ = build-global-trace($T_1^-$, $T_2^-$, $\cdots$, $T_p^-$);
4  $A$ = $\{a \mid \tau_j = (op, a, v) \wedge 1 \leq j \leq n\}$;
5  for $i$ = 1 to $|A|$ do
6      V[i] = NIL;
7  for $j$ = 1 to $n$ do
8      let $\tau_j$ be (op, a, v);
9      if $op = S$ then
10         V[h(a)] = v;
11     if $op = L \wedge v \neq V[h(a)]$ then
12         return false;
13 return true;
```

Lemma 2 - Axiom 3 holds for global producers iff Property 1 holds.

Proof: Let σ_a^i be a shorthand notation for $\{S_a^k | S_a^k \leq L_a^i\}$ and let $\sigma_a^i(j) \subset \sigma_a^i$ denote stores issued by processor j. *Necessity*: Property 1 ensures that $\forall S_a^i, S_a^k \in \mathcal{S} : (S_a^i \leq S_a^k) \vee (S_a^k \leq S_a^i)$. Since $\sigma_a^i \subset \mathcal{S}$, we have: $\forall S_a^i, S_a^k \in \sigma_a^i : (S_a^i \leq S_a^k) \vee (S_a^k \leq S_a^i)$. As $\leq$ is a linear order on σ_a^i, the statement $\exists s \in \sigma_a^i : s = Max_{\leq}[\sigma_a^i]$ holds for every L_a^i. When L_a^i executes, it must consume the last value stored at address a; therefore, $Val[L_a^i] = Val[\sigma_a^i]$ holds for every $a \in A$.

Sufficiency: Given a pair of stores (x, y), let $p_1(x, y)$ be the predicate specified by Property 1. Axiom 3 ensures that, for every L_a^i, we have $\exists s \in \sigma_a^i : s = Max_{\leq}[\sigma_a^i] = Max_{\leq}[\sigma_a^i(1) \cup \cdots \cup \sigma_a^i(j) \cup \cdots \cup \sigma_a^i(p)] = Max_{\leq}[\{Max_{\leq}[\sigma_a^i(1)], \cdots, Max_{\leq}[\sigma_a^i(j)], \cdots, Max_{\leq}[\sigma_a^i(p)]\}]$.

The existence of local maxima proves that all stores in $\sigma_a^i(j)$ are linearly ordered by $\leq$, i.e. that $\forall (x, y) \in \sigma_a^i(j) \times \sigma_a^i(j) : p_1(x, y)$ holds for every L_a^i or, equivalently, that the statement $\forall (x, y) \in \mathcal{S}^j \times \mathcal{S}^j : p_1(x, y)$ holds, as the value produced by each store is consumed by at least one load. For simplicity, let M_a^i be a shorthand notation for the set $\{Max_{\leq}[\sigma_a^i(1)], \cdots, Max_{\leq}[\sigma_a^i(j)], \cdots, Max_{\leq}[\sigma_a^i(p)]\}$. The existence of a global maximum proves that all store operations in the set M_a^i are linearly ordered by $\leq$. Therefore the statement $\forall (x, y) \in M_a^i \times M_a^i| : p_1(x, y)$ holds for each L_a^i.

Since the value produced by every store is consumed by at least one load L_a^i, we conclude that the set $\cup_{i,a} M_a^i$ is the set of all value producers, i.e. $\cup_{i,a} M_a^i = \mathcal{S}$. As $\mathcal{S} = \{\mathcal{S}^1 \cup \cdots \cup \mathcal{S}^j \cup \cdots \cup \mathcal{S}^k \cdots \cup \mathcal{S}^p\}$ is a partition and each store in M_a^i was issued by a distinct processor, then every store in M_a^i belongs to exactly one component of the partition. Since, for each pair $(x, y) \in M_a^i$, the store operations $x = Max_{\leq}[\sigma_a^i(j)]$ and $y = Max_{\leq}[\sigma_a^i(k)])$ were issued by distinct processors j and k, we conclude that the statement $\forall (x, y) \in \mathcal{S}_j \times \mathcal{S}_k| : p_1(x, y)$ holds for $j \neq k$. As a similar statement was proved for $j = k$, we conclude that Property 1 holds. ■

Lemma 3 - For any cache-coherent memory system, the order $\leq$ is indistinguishable from the order $<_t$.

Proof: We want to prove that $\forall S_a^i, S_a^k \in \mathcal{S} : (S_a^i \leq S_a^k) \Leftrightarrow (S_a^i <_t S_a^k)$. *Necessity.* For a correct execution of the the memory system's representation, two operations specified as ordered must be observed in successive times, i.e. $x \leq y \Rightarrow t(x) < t(y)$. Thus, for $x = S_a^i$ and $y = S_a^k$, we have $\forall a \in A : S_a^i \leq S_a^k \Rightarrow t(S_a^i) < t(S_a^k)$.

Sufficiency. Since $\leq$ is a partial order, one of the following scenarios must be observed: $x \leq y \Rightarrow t(x) < t(y)$, $y \leq x \Rightarrow t(x) > t(y)$, or $\neg(x \leq y) \wedge \neg(y \leq x) \Rightarrow t(x)$ and $t(y)$ are arbitrary. Therefore, $t(x) < t(y) \Rightarrow (x \leq y) \vee \neg((x \leq y) \vee (y \leq x))$. Assuming $x = S_a^i$ and $y = S_a^k$, we have: $t(S_a^i) < t(S_a^k) \Rightarrow (S_a^i \leq S_a^k) \vee \neg((S_a^i \leq S_a^k) \vee (S_a^k \leq S_a^i))$ for every $a \in A$. Since Property 1 must hold for any cache-coherent memory system, the second clause of the disjunction is false. Thus, $t(S_a^i) < t(S_a^k) \Rightarrow (S_a^i \leq S_a^k)$ holds.

Hence, $(S_a^i \leq S_a^k) \Leftrightarrow t(S_a^i) < t(S_a^k)$, i.e. $(S_a^i \leq S_a^k) \Leftrightarrow (S_a^i <_t S_a^k)$ for every $a \in A$ (Definition 7). ■

Lemma 4 - Algorithm 2 returns true iff Axiom 3 holds.

Proof: As events are visited in order of increasing timestamps, the last value assigned by Algorithm 2 to $V[h(a)]$ (at lines 9–10) is $Max_{<_t}[\sigma_a^i]$. As it only returns false if $Val[L_a^i] \neq Val[Max_{<_t}[\sigma_a^i]]$ for some $a \in A$ (lines 11–12), it returns true iff $Val[L_a^i] = Val[Max_{<_t}[\sigma_a^i]]$ for all $a \in A$. To assert that this last statement is equivalent to Axiom 3, we have to prove that $Max_{<_t}[\sigma_a^i] = Max_{\leq}[\sigma_a^i]$, i.e. we need to show that the statement $(S_a^i \leq S_a^k) \Leftrightarrow (S_a^i <_t S_a^k)$ holds for all $S_a^i, S_a^k \in \sigma_a^i$. Indeed, this is guaranteed by Lemma 3, since $\forall a \in A : \sigma_a^i \subseteq \mathcal{S}$ ■

C. Overall complexity and combined theoretical guarantees

Algorithm 3 simply invokes Algorithm 1 for each processor before invoking Algorithm 2. As the E-matching algorithm is invoked p times, Step 1 takes $O(n^6/p^5)$ in the worst case. Since Algorithm 2 is invoked only once, Step 2 takes $O(n \log p)$. As a result, the overall verification effort takes $O(n^6/p^5)$ in the worst case. However, since the p invocations of the E-matching algorithm are fully independent, when the sequential loop (lines 1–3) is converted into a parallel one, the overall complexity is reduced to $O(n^6/p^6)$.

Algorithm 3: behavior-ok(MCM, $T_1^+, \cdots, T_p^+$, $T_1^-, \cdots, T_p^-$)

```
1 for $i$ = 1 to $p$ do
2     if ¬local-behavior-ok(MCM, $T_i^+$, $T_i^-$) then
3         return false;
4 return global-behavior-ok($T_1^-$, $T_2^-$, $\cdots$, $T_p^-$);
```

Theorem 1 - For any cache-coherent memory system and for any MCM not requiring total store ordering, Algorithm 3 returns true iff all the MCM's axioms hold for the observed behavior induced by a given test case.

Proof: Let l^i = local-behavior-OK(MCM, T_i^+, T_i^-) and let g = global-behavior-ok(T_1^-, T_2^-, $\cdots$, T_p^-).

From Lemma 1, we have $l^1 \wedge l^2 \wedge \cdots \wedge l^i \wedge \cdots \wedge l^p \Leftrightarrow$ (Axiom 1 holds) $\wedge$ (Axiom 2 holds for every $(u, v) \in \mathcal{S}^i \times \mathcal{S}^i$) $\wedge$ (Axiom 3 holds for local producers) $\wedge$ (Property 3 holds). From Lemma 4, we have $g \Leftrightarrow$ (Axiom 3 holds for global producers). Since from Lemma 2 we know that, for MCMs not requiring total store ordering, Axioms 2 and 3 are equivalent and Axiom 3 $\Rightarrow$ Property 3, we conclude that $l^1 \wedge l^2 \wedge \cdots \wedge l^i \wedge \cdots \wedge l^p \wedge g \Leftrightarrow$ Axioms 1, 2, and 3 hold. ■

VI. Experimental Results

We employed an RIT generator [11] that accepts four parameters: the number of processors (p), the overall number of operations (n), the number of shared addresses (s), and the instruction mix π = (π_L, π_S, π_X, π_M), where π_t is the probability of occurrence of a memory operation of type $t \in \{L, S, X, M\}$. We generated 240 distinct RITs. We relied on executable representations built upon the GEM5's infrastructure [12]. We customized one of GEM5's templates (private L1 instruction/data caches, shared L2 cache, snooping for coherence, and ARO for consistency). We compared our technique with a conventional checker that implements a version of the algorithm described in [2], which was tailored to handle ARO. First, we ran all 240 test cases on a platform

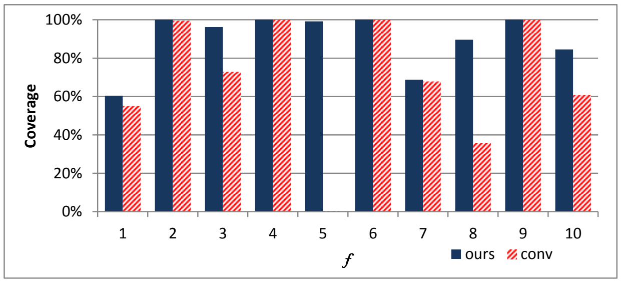

Fig. 3. Fault coverage

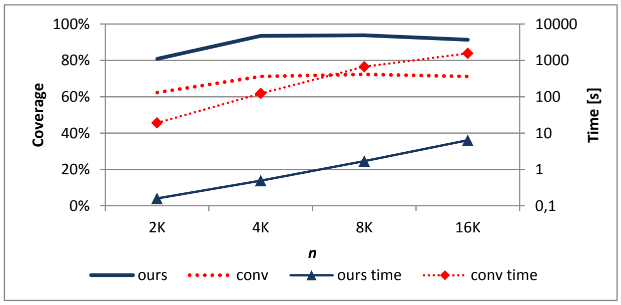

Fig. 4. Overall behavior when varying n

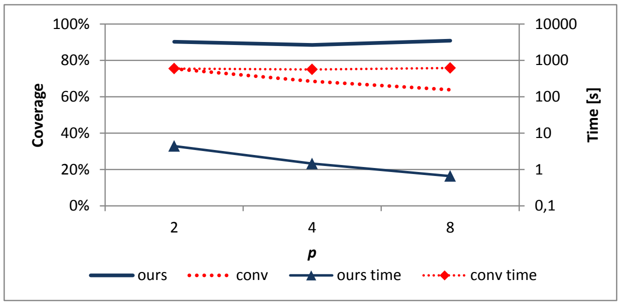

Fig. 5. Overall behavior when varying p

without faults and let it be verified by our checker. From a given reference platform instance, we derived ten faulty platforms, each with a distinct fault type (stuck-at-bit, overlooked memory barrier, lack of coherence, etc.) and location (store queue bypass, reorder buffer, cache and protocol controllers, etc.). Every generated RIT was run on each faulty platform, giving rise to 2400 **use** cases.

Figure 3 compares the coverage of individual faults. Notice that our checker led to higher or equal coverage for all faults as compared to the conventional one. On average, our checker covers 90% of the faults, while the conventional one covers 69% of them. Notice that Fault 5 was almost never found by the conventional checker. Its extremely low coverage (0.42%) can be explained as follows. The MCM under verification is rather relaxed. Therefore, the number of constraints induced by the order axiom is small. Since fault 5 does not induce a violation of the value axiom as a side effect, the number of constraints is insufficient to induce a cycle. Note that, the conventional checker reaches less than half the coverage of our checker for fault 8. Since – for more than 60% of the test cases – it cannot infer that a consumed value is obsolete, a missing relation precludes the closing of a cycle.

Figure 4 compares the overall behavior when the size of the test cases is increased. Notice that our checker reaches higher coverage with smaller tests due to the extended observability and its coverage remains 20% higher than the conventional one. Despite its higher coverage, our checker is two orders of magnitude faster.

Figure 5 shows the behavior of both checkers when the number of processors is scaled up. Note that the coverage obtained with our checker is largely independent from scaling, as opposed to the conventional one, whose coverage actually decreases. Since the complexity of E-matching decreases with increasing p, our checker's verification time decreases. This is an evidence that our checker is suitable for the verification challenges raised by architectural scaling.

VII. Conclusion and Future Work

We showed that our complete technique is faster than an incomplete conventional checker. Although the latter is suitable for both design-time verification and post-silicon testing, the use of an ESL memory verification technique pays off. The augmented observability leads to a speed up of two orders of magnitude and allows the detection of 20% more faults in spite of the fact that only two points must be monitored in each processor's executable representation. Besides, since the selected points are quite generic, the technique is largely independent from microarchitecture choice. It is likely that multiprocessor systems-on-chip will require largely-relaxed MCMs for higher performance. Since the efficiency of conventional checkers is reduced for such MCMs (due to the smaller probability of cycle detection) and since our technique benefits from improved verification guarantees when addressing relaxed MCMs (Theorem 1), our contribution seems to open a promising path.

References

[1] S. V. Adve and K. Gharachorloo, “Shared Memory Consistency Models: A Tutorial,” *IEEE Computer*, 1996.

[2] S. Hangal, D. Vahia, C. Manovit, J. Lu, and S. Narayanan, “TSOtool: A Program for Verifying Memory Systems Using the Memory Consistency Model,” in *International Symposium on Computer Architecture*, 2004.

[3] C. Manovit and S. Hangal, “Efficient Algorithms for Verifying Memory Consistency,” in *ACM Symposium on Parallelism in Algorithms and Architectures*, 2005.

[4] Y. Chen, Y. Lv, W. Hu, T. Chen, H. Shen, P. Wang, and H. Pan, “Fast Complete Memory Consistency Verification,” in *International Symposium on High-Performance Computer Architecture*, 2009.

[5] P. B. Gibbons and E. Korach, “Testing Shared Memories,” *SIAM Journal on Computing*, 1997.

[6] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, 4th ed. Morgan Kaufmann, 2006.

[7] R. L. Sites and R. T. Witek, *Alpha AXP Architecture Reference Manual (2nd ed.)*. Newton, MA, USA: Digital Press, 1995.

[8] C. Manovit and S. Hangal, “Completely Verifying Memory Consistency of Test Program Executions,” in *Symposium on High-Performance Computer Architecture*, 2006.

[9] A. Roy, S. Zeisset, C. Fleckenstein, and J. Huang, “Fast and Generalized Polynomial Time Memory Consistency Verification,” *Lecture Notes in Computer Science*, 2006.

[10] G. Marcilio, L. C. V. Santos, B. Albertini, and S. Rigo, “A Novel Verification Technique to Uncover Out-of-Order DUV Behaviors,” in *ACM/IEEE Design Automation Conference*, 2009.

[11] E. A. Rambo, O. Henschel, and L. C. V. Santos, “Automatic Generation of Memory Consistency Tests for Chip Multiprocessing,” in *IEEE International Conference on Electronics, Circuits, and Systems*, 2011.

[12] N. Binkert, R. Dreslinski, L. Hsu, K. Lim, A. Saidi, and S. Reinhardt, “The M5 simulator: Modeling networked systems,” *IEEE Micro*, 2006.