On-the-fly Verification of Memory Consistency with Concurrent Relaxed Scoreboards

Leandro S. Freitas, Eberle A. Rambo and Luiz C. V. dos Santos
Computer Science Department, Federal University of Santa Catarina, Brazil
Email: {freitass, eberle18, santos}@inf.ufsc.br

***Abstract*—Parallel programming requires the definition of shared-memory semantics by means of a consistency model, which affects how the parallel hardware is designed. Therefore, verifying the hardware compliance with a consistency model is a relevant problem, whose complexity depends on the observability of memory events. Post-silicon checkers analyze a single sequence of events per core and so do most pre-silicon checkers, although one reported method samples two sequences per core. Besides, most are post-mortem checkers requiring the whole sequence of events to be available prior to verification. On the contrary, this paper describes a novel on-the-fly technique for verifying memory consistency from an executable representation of a multicore system. To increase efficiency without hampering verification guarantees, three points are monitored per core. The sampling points are selected to be largely independent from the core's microarchitecture. The technique relies on concurrent relaxed scoreboards to check for consistency violations in each core. To check for global violations, it employs a linear order of events induced by a given test case. We prove that the technique neither indicates false negatives nor false positives when the test case exposes an error that affects the sampled sequences, making it the first on-the-fly checker with full guarantees. We compare our technique with two post-mortem checkers under 2400 scenarios for platforms with 2 to 8 cores. The results show that our technique is at least 100 times faster than a checker sampling a single sequence per processor and it needs approximately 1/4 to 3/4 of the overall verification effort required by a post-mortem checker sampling two sequences per processor.**

I. Introduction

Perhaps the most primitive question we can ask about the behavior of a memory system is: *what* is the value returned by a load instruction? And yet the answer is not trivial for multiprocessors, because the "last" store instruction writing to the same address is not precisely specified by program order. Therefore, shared-memory parallel programming requires the definition of memory semantics by means of a *memory consistency model* (MCM) [1], which essentially specifies *when* a stored value must be seen by a load instruction. The simplest way to precisely define memory semantics is to enforce a sequential order of all operations, but this disallows compiler and hardware optimizations. That is why relaxed MCMs are required, such as Weak Ordering (ARM), Total Store Order [2] (SPARC), and the ones adopted by Alpha [3] and PowerPC.

This work was partially supported by the Brazilian Council for Scientific and Technological Development (CNPq) through grants 556757/2009-2 (PNM), 559882/2010-6 (CTINFO-PDI), and 306654/2009-1 (PQ).

An MCM not only affects how parallel programs are written, but also how parallel hardware is designed (e.g. multiple out-of-order load/store units, store queues with read bypassing, multiple memory modules, lock-up free caches, cache-coherence protocols, etc.). The verification of the hardware's compliance with an MCM is a relevant problem whose complexity depends on the observability of memory events.

This work exploits the extended observability of an executable representation to speed up the verification of consistency, instead of reusing – at design time – methods originally developed for post-silicon testing. We exploit the extra observability by oversampling memory events. Instead of a single sequence [2], [4]–[9] or two [10], we sample three sequences per processor. Instead of trying to infer order relations between memory operations [2], [5]–[9] or relying on bipartite graph matching [10], we verify the equivalence of expected and observed sequences on the fly. Although conventional scoreboards are more efficient than postmortem checkers [2], [4]–[10], they cannot handle the multiple behaviors that an MCM allows for the same sequence of stimuli. That is why the use of a *relaxed* scoreboard was proposed for MCM verification [4]. Unfortunately, such use of a *single* relaxed scoreboard offers *limited* verification guarantees. For this reason, we propose an entirely new algorithm for a relaxed scoreboard that leads to *full* verification guarantees when *multiple* instances of the novel scoreboard class are employed to concurrently check consistency at each processor.

Section II formulates the target problem. Section III briefly reviews related work. Section IV formalizes the algorithms underlying the proposed technique. Section V provides the theoretical guarantees. Section VI reports experimental results by comparing the technique with two post-mortem checkers in terms of effectiveness and efficiency. Finally, in Section VII, we put theoretical and experimental results in perspective.

II. The Target Verification Problem

To clearly establish the links to related work, we borrow the notations used in [2] and [10] with a few adaptations.

An MCM is specified by two types of axioms. *Order* axioms define the degree of relaxation w.r.t. program order. A *value* axiom restricts the values that a load can return. Formal descriptions of such axioms can be found in the literature (e.g. [2], [10]) for distinct MCMs. Their verification requires the observation of memory traces, as formalized below.

Definition 1: A *trace* is a sequence $(\tau_1, \tau_2, \cdots \tau_j, \cdots \tau_m)$, where $\tau_j = (op, a, v)$ is a memory event such that $op \in \{Load, Store, Swap\}$, a is an address, and v is the value read from or written to memory at address a.

Let n be the *number of memory operations* of a parallel program and p be the *number of processors*. The verification of memory consistency can be formalized as follows:

Problem 1: Given a collection of traces $T_1, T_2, \cdots T_p$, is there a global trace T satisfying all the axioms of an MCM?

III. Related work

Let us review how Problem 1 is addressed by many authors, as summarized in Table I. For each checker, the table shows whether it can be used for design-time verification (*pre-silicon*) or prototype testing (*post-silicon*), if the analysis is done during simulation (*on-the-fly*) or after it (*post-mortem*), and whether it offers or not full guarantees of finding an error exposed by a given test case. Finally, it shows the required observability and its impact on worst-case time complexity. The table's last row contrasts the checker proposed in the next section with the related works.

Most checkers require all traces to be available before verification can start and they rely on directed acyclic graphs (DAGs) to encode order relations inferred from the traces. The detection of a cycle in a directed graph is a proof of memory inconsistency. The fact that no cycles are detected, however, is not a proof of consistency, because the relation between some operations might not have been inferred. To rule out the false negatives that may result from the limitations of the inference mechanism, a few checkers rely on backtracking [7]–[9], leading to large runtimes, especially when the number of processors increases. One pre-silicon method [10] offers similar guarantees without the need for backtracking. It relies on sampling two sequences of memory events per core. Since it reused "as is" the matching algorithm proposed in [11] to solve a *more general* problem, it unnecessarily inherited a high worst-case complexity, which could be reduced by tailoring the matching algorithm to the actual target instance. Experimental evidence show, however, that the average computational effort is much smaller when using random instruction tests.

A recent method [9] claims that MCM verification can be performed in linear time for a fixed number of processors. However, since it adopts backtracking, its long-term scalability is limited by an exponential factor (C^p, where C is a constant).

As Table I points out, on-the-fly checkers are barely used for MCM verification. This is due to the fact that conventional scoreboards, which are very efficient checkers, cannot directly handle a subsystem that does not preserve at its output the order corresponding to its input stimuli.

That is why the use of a *relaxed* scoreboard was proposed for MCM verification [4]. Instead of recording a single expected value for each input stimulus, as does a conventional scoreboard, the relaxed scoreboard keeps multiple expected values as long as a single value cannot be deterministically identified. It relies on an update rule that records new values after each store and dynamically removes from the scoreboard the values that become invalid after each load. As a result, the number of expected values for each stimulus is progressively narrowed down. Although it represents a very efficient solution for replacing inference-based checkers [2], [5]–[9], the authors acknowledge that it may induce false negatives [4], because the relaxed scoreboard never revisits a previous decision.

Since a *single* relaxed scoreboard has limited verification guarantees, we devised a method relying on *multiple* relaxed scoreboards, which *concurrently* check consistency from the perspective of each processor, as described in the next section.

IV. The proposed on-the-fly checker

As conventional and relaxed scoreboards may induce false negatives, and the full guarantees of a pre-silicon post-mortem checker come at the expense of higher verification effort, the proposed checker is built upon the following foundations:

1) *Sampling in program order*: memory operations are monitored at the output of each processor's commit unit;
2) *Sampling in execution order*: memory operations are monitored at each processor's interface with its private cache;
3) *Checking for uncommitted operations*: The reorder buffer is monitored for identifying uncommitted operations.

The first two monitors avoid that the limited observability of memory events may lead to false negatives [10]. The third monitor precludes that implementation artifacts (such as speculative operations) may lead to false positives.

Note that the points to be monitored were judiciously chosen to be largely independent from the specificities of a microarchitecture handling out-of-order execution. From now on, let i^+, i^-, and i^* represent monitors placed at the commit unit of processor i, at the interface between processor i and its private cache, and at its reorder buffer, respectively.

We decomposed the resolution of Problem 1 in two subproblems addressed by two types of complementary checkers: one type is used to verify consistency from the point of view of each processor, the other is used for checking consistency from a global perspective. It should be noted that both checkers address only operations visible by all processors. Operations that never reach the memory interface (e.g. read-on-write-early [1]) are addressed separately. We employ a single global checker and p *independent* local checkers, each relying on a distinct relaxed scoreboard. As soon as a scoreboard detects a mismatch, an error is raised. If the monitored sequences are locally consistent for all processors, then a global checker further verifies if the value returned by each load is unique from a global perspective and indicates an error if not.

To build the global checker, we adopted the algorithm global-behavior-ok proposed in [10], because it employs a linear order induced by the execution of a *given* test case, instead of inferring valid orderings via backtracking [6]–[9], a mechanism that becomes impractical with processor upscaling.

However, to build the local checkers, we designed a *novel* technique that instantiates one relaxed scoreboard per processor. Each scoreboard instance waits for events at the monitors i^+, i^-, and i^* of each processor i so as to continuously update the *sets of monitored events* V_i^+, V_i^-, and V_i^*.

TABLE I: A qualitative comparison between distinct classes of consistency checkers

Technique	Usage	Type	Key idea	Guarantees	Monitors	Worst-case complexity
[2]	(Pre-) Post-silicon	Post-mortem	DAG-based inference		1	$O(n^5)$
[5]	(Pre-) Post-silicon	Post-mortem	DAG-based inference		1	$O(pn^3)$
[7]	(Pre-) Post-silicon	Post-mortem	DAG-based inference	full	1	$O((n/p)^p pn^3)$
[6]	(Pre-) Post-silicon	Post-mortem	DAG-based inference		1	$O(n^4)$
[8]	(Pre-) Post-silicon	Post-mortem	DAG-based inference	 full	1	$O(p^3n)$ $O(p^2n^2)$
[9]	(Pre-) Post-silicon	Post-mortem	DAG-based inference	full	1	$O(p^3n)$
[4]	Pre-silicon	On-the-fly	Single relaxed scoreboard		1	$O(p^2n^2)$
[10]	Pre-silicon	Post-mortem	Bipartite graph matching	full	2	$O(n^6/p^5)$
This work	Pre-silicon	On-the-Fly	Concurrent relaxed scoreboards	full	3	$O(n^2/p)$

Let $\leq$ be a partial order specified by the (order) axioms of a given MCM[1]. Given a processor i, a local checker verifies if the sequences monitored at points i^+ and i^- are consistent. Two sequences are considered consistent when every committed event is equivalent to an event observed at the memory interface, an observed event is committed only once, and the observed events that were committed satisfy the partial order $\leq$ specified by the axioms of a given MCM. This notion is formalized below.

Definition 2: Given a partial order $\leq$ on the set $V^+ = \{v_1^+, v_2^+, \cdots, v_N^+\}$ and an equivalence relation $R = \{(j,m) \in \{1,\cdots,N\} \times \{1,\cdots,M\} : v_j^+ \equiv v_m^-\}$, we say that the sequences $(v_1^+, v_2^+, \cdots, v_N^+)$ and $(v_1^-, v_2^-, \cdots, v_M^-)$ are *consistent* iff there is a mapping $\mu : \{1,\cdots,N\} \mapsto \{1,\cdots,M\}$ such that:

1) μ is a function such that $\mu \subseteq R$;
2) μ is an injection;
3) $\forall k,j \in \{1,\cdots,N\} : (v_k^+ \leq v_j^+) \wedge (\mu(k) = t) \wedge (\mu(j) = m) \Rightarrow (t < m)$.

An event seen at i^- is either a committed event observed at i^+ or an uncommitted event seen at i^*, i.e. $|V_i^-| = |V_i^+| + |V_i^*|$, as formalized below:

Property 1: Let $\sigma : V_i^- \mapsto V_i^*$ be an injective function such that $\sigma \subseteq \{(v_m^-, v_s^*) \in V_i^- \times V_i^* : v_m^- \equiv v_s^*\}$. For every $v_j^- \in V_i^-$, only one of the following conditions hold:

1) $(\exists m \in \{1,\cdots,M\} : \mu(j) = m) \wedge (\nexists v_s^* \in V_i^* : \sigma(v_m^-) = v_s^*)$
2) $(\nexists m \in \{1,\cdots,M\} : \mu(j) = m) \wedge (\exists v_s^* \in V_i^* : \sigma(v_m^-) = v_s^*)$

Algorithm 1 continuously monitors events until the simulation ends. It returns false as soon as a mismatch is detected. After simulation, it returns true if all committed events were matched and all events observed at the memory interface were either committed or uncommitted. It returns false if some committed event was left unmatched or Property 1 was violated. At line 5, it invokes the function match(i, v_m^-), where lies the new checking mechanism described by Algorithm 2.

Algorithm 2 returns false if no committed event was found to be equivalent to the observed event v_m^- or if its matching to an equivalent event would violate the pre-specified order of events ($\leq$). First, it finds the committed events that are equivalent to a given v_m^- (line 2). If any is found (line 3), it selects the first match v_j^+ (line 4). Then it finds the set of dominators of v_j^+ with respect to the order $\leq$ (line 5). If no equivalent event is found ($Q = \emptyset$), this would mean that a committed event was not observed at the memory interface or v_m^- was not committed (it was "squashed"). Similarly, if among the committed events still unmatched, there is some dominator for v_j^+ ($D \neq \emptyset$), the matching of v_j^+ with v_m^- would violate the order $\leq$ unless v_m^- was a squashed event. That is why, in both scenarios (line 6), the algorithm first looks for squashed events that match v_m^- (line 7). Since the non-existence of some squashed event (line 8) is a proof of inconsistency, the algorithm returns false. Otherwise, it arbitrarily selects for v_m^- an equivalent squashed event v_x^* (line 10) and removes it from the scoreboard (line 11). In this case,

Algorithm 1 LOCAL-BEHAVIOR-OK(i)

```
1:  m ← 1
2:  let T_i^- be an empty sequence of events
3:  repeat
4:      v_m^- ←monitor(i^-)
5:      if ¬ match(i, v_m^-) then
6:          return (false)
7:      m ← m + 1
8:      concatenate v_m^- to T_i^-
9:  until end-of-simulation
10: if (V_i^+ = ∅) ∧ (V_i^- = ∅) then
11:     return (true)
12: else
13:     return (false)
```

Algorithm 2 match(i, v_m^-)

```
1:  D ← ∅
2:  Q ← {v_q^+ ∈ V_i^+ : v_q^+ ≡ v_m^-}
3:  if Q ≠ ∅ then
4:      j ← min{1 ≤ q ≤ |V_i^+| : v_q^+ ∈ Q}
5:      D ← {v_k^+ ∈ V_i^+ : v_k^+ ≤ v_j^+}
6:  if Q = ∅ ∨ D ≠ ∅ then
7:      S ← {v_s^* ∈ V_i^* : v_s^* ≡ v_m^-}
8:      if S = ∅ then
9:          return (false)
10:     x ← arbitrary element of {1 ≤ s ≤ |V_i^*| : v_s^* ∈ S}
11:     V_i^* ← V_i^* − {v_x^*}
12: else
13:     V_i^+ ← V_i^+ − {v_j^+}
14: V_i^- ← V_i^- − {v_m^-}
15: return (true)
```

[1]Including the Membar axiom [2]. A Membar is a barrier that prevents operations from being reordered through it. Although monitored by i^+, Membars are not seen by i^- (they do not reach the memory system). For consistency verification, when their effect is captured by $\leq$, the analysis can be reduced to sequences free of Membars without loss of generality.

v_j^+ is kept in the scoreboard (since it may match a later event). On the contrary, when an equivalent v_j^+ was found and it has no dominators, v_j^+ is considered the good match for v_m^- and is removed from the scoreboard (line 13). Finally, whether v_m^- was matched to a committed event or to a squashed event, it is removed from the scoreboard (line 14).

Algorithm 3 behavior-ok()

```
1: for i ← 1 to p do in parallel
2:     if ¬LOCAL-BEHAVIOR-OK(i) then
3:         return false
4: return global-behavior-ok(T_1^-, T_2^-, ···, T_p^-)
```

Algorithm 3 integrates local and global verification of memory consistency. The novelty of our checker lies exactly in the use of multiple instances of a new class of relaxed scoreboard to *concurrently* evaluate LOCAL-BEHAVIOR-OK(i) for every i (line 2). After consistency is locally checked for all p processors, a global checking is required (line 4). As already justified, we reuse the algorithm global-behavior-ok, which is formally described in [10]. Essentially, that algorithm builds a global trace from the local ones, according to a linear ordering induced by a given test case, and checks for consistent value consumption.

Given a perfectly-balanced parallel program serving as a test case, each processor executes exactly n/p operations, i.e. $|V_i^+| = n/p$, and the worst-case scenario corresponds to a sequence of n/p operations such that the execution of each one causes the squashing of the whole reorder buffer, i.e. $|V_i^*| \leq Cn/p$, where C is the (constant) size of the reorder buffer. Therefore, Algorithm 2 takes $O(n/p)$. Since $|V_i^-| = |V_i^+| + |V_i^*|$, Algorithm 1 invokes Algorithm 2 at most $n/p + Cn/p$ times. Thus, Algorithm 1 takes $O(n^2/p^2)$. Considering that global-behavior-ok takes $O(n \log p)$ [10], Algorithm 3 takes $O(n^2/p)$ under the pessimistic assumption that its parallel loop performs all p iterations sequentially.

V. Theoretical guarantees

To establish guarantees for the proposed technique, we partially rely on proofs from related work [10], [11]. Since we deliberately reused the global checker from the post-mortem technique proposed in [10], but we replaced each local checker with an on-the-fly version (LOCAL-BEHAVIOR-OK) of the original post-mortem algorithm (local-behavior-ok), we can inherit the guarantees provided by that work [10] if we can prove that the former (which is based on a relaxed scoreboard) is equivalent to the latter (which relies on extended bipartite graph matching).

Lemma 1: Every invocation of Algorithm 2 such that $Q \neq \emptyset \wedge D = \emptyset$ removes exactly one element from each of the sets V_i^+ and V_i^-, and returns true.

Proof: Let match(i, v_m^-) denote an invocation of Algorithm 2 such that $Q \neq \emptyset \wedge D = \emptyset$. Since $(Q \neq \emptyset) \wedge (D = \emptyset)$ holds, one element (v_j^+) is removed from (V_i^+) (line 13), one element (v_m^-) is removed from V_i^- (line 14), and the algorithm returns true. ∎

Lemma 2: If the sequences $(v_1^+, v_2^+, \cdots, v_N^+)$ and $(v_1^-, v_2^-, \cdots, v_M^-)$ are consistent, then every invocation of Algorithm 2 such that $Q = \emptyset$ removes exactly one element from V_i^-, none from V_i^+, and returns true.

Proof: By hypothesis, all clauses from Definition 2 hold. Let match(i, v_m^-) denote an invocation of Algorithm 2 such that $Q = \emptyset$. Since Clauses 1 and 2 hold and $Q = \emptyset$, we conclude that $\forall j \in \{1, \cdots, N\} : \mu(j) \neq m \Rightarrow \forall v_j^+ \in V_i^+ : v_j^+ \not\equiv v_m^-$. Therefore, from Property 1 (Condition 2), we conclude that $\exists v_s^* \in V_i^* : v_m^- \equiv v_s^*$ for a given v_m^-. Thus, since $S \neq \emptyset$, v_m^- is removed from V_i^-, v_j^+ is not removed from V_i^+, and the algorithm returns true. ∎

Theorem 1: Algorithm 1 returns true iff the sequences $(v_1^+, v_2^+, \cdots, v_N^+)$ and $(v_1^-, v_2^-, \cdots, v_M^-)$ are consistent.

Proof: Assumption 1 – Consistent sequences.

(1.1) $N = M$: Since Algorithm 1 calls Algorithm 2 M times under condition $Q \neq \emptyset \wedge D = \emptyset$ and as $M = N$, we conclude that N elements are removed from V_i^+ and M elements from V_i^- (Lemma 1). Therefore, on exit to the M-th call, $(V_i^+ = \emptyset) \wedge (V_i^- = \emptyset)$. Thus, Algorithm 1 returns true.

(1.2) $N < M$: Let $|V_i^*| = U$. Algorithm 1 calls Algorithm 2 N times under the condition $Q \neq \emptyset \wedge D = \emptyset$. We conclude that N elements are removed from V_i^+ and N elements from V_i^- (Lemma 1). As it calls Algorithm 2 U times under the condition $Q = \emptyset \vee D \neq \emptyset$, U elements are removed from V_i^- (Lemma 2). Since $|V_i^-| = M$, $|V_i^+| = N$, and $M = N + U$, we conclude that, after the M-th call, $(V_i^+ = \emptyset) \wedge (V_i^- = \emptyset)$. Thus, Algorithm 1 returns true.

Assumption 2 – Inconsistent sequences.

(2.1) μ is not a function.

(2.1.1) Unmapped domain element: For some $u \in \{1, 2, \cdots, N\}$, the following holds: $\forall m \in \{1, 2, \cdots, M\}$: $v_u^+ \not\equiv v_m^-$. For any invocation of match(i, v_m^-), we have $v_u^+ \notin Q$. When $(Q = \emptyset) \vee (D \neq \emptyset)$ holds, v_u^+ is not removed from V_i^+. When $Q \neq \emptyset \wedge D = \emptyset$ holds, v_j^+ is removed from V_i^+ but $v_j^+ \neq v_u^+$. Since v_u^+ is never removed from V_i^+, Algorithm 1 returns false since $V_i^+ \neq \emptyset$.

(2.1.2) Multiple mappings from same domain element: For some $j \in \{1, 2, \cdots, N\}$, the following holds: $\exists m, t, \cdots, z \in \{1, 2, \cdots, M\} : v_j^+ \equiv v_m^-, v_j^+ \equiv v_t^-, \cdots, v_j^+ \equiv v_z^-$ and $\forall y \in \{j+1, \cdots, N\} : v_y^+ \not\equiv v_t^-, \cdots, v_y^+ \not\equiv v_z^-$. Therefore, since v_j^+ is removed from V_i^+, the events $v_t^-, \cdots, v_z^-$ can never be removed from V_i^- in later calls to Algorithm 2, leading to $V_i^- \neq \emptyset$. Thus, Algorithm 1 returns false (line 13).

(2.2) μ is a non-injective function: For some $m \in \{1, 2, \cdots, M\}$, the following holds: $\exists j, k, \cdots, x \in \{1, 2, \cdots, N\} : \mu(j) = m \wedge \mu(k) = m \wedge \cdots \wedge \mu(x) = m$. Since μ is a function and $\mu(k) = m$, we conclude that $\forall y \neq m : \mu(k) \neq y, \cdots, \mu(x) \neq y$. Since Algorithm 2 removes v_m^- from V_i^- at its m-th invocation, we conclude that $v_k^+ \notin Q, \cdots, v_x^+ \notin Q$ for any arbitrary invocation such that $y \in \{m, \cdots, M\}$. Therefore, the events $v_k^+, \cdots, v_x^+$ can never be removed from V_i^+. Thus, $V_i^+ \neq \emptyset$ on exit to the last call to Algorithm 2, and Algorithm 1 returns false (line 13).

(2.3) Partial order $\leq$ is violated: $\exists k, j \in \{1, \cdots, N\} : (v_k^+ \leq v_j^+) \wedge (\mu(j) = m) \wedge (\mu(k) = t) \wedge (t \geq m)$ and

TABLE II: Error characterization

ID	Description	Location
e1	Outstanding store overlooked by load to same address	Store queue bypass
e2	Stores to same address committed out of program order	Reorder buffer
e3	Incorrect value from outstanding store forwarded to requesting load	Store queue bypass
e4	Incorrect value from cache sent to requesting load	Data cache interface
e5	Violation of memory barrier	Execution unit control
e6	Obsolete block read due to invalidation-mechanism malfunction	Data cache bus interface
e7	Corrupted block read due to invalidation-mechanism malfunction	Data cache bus interface
e8	Reset of validity bit precluded	Cache control logic
e9	Correct value written by swap but returned value corrupted	Data cache interface
e10	Obsolete value read due to precluded setting of dirty bit	Cache control logic

$\forall y \in \{t, \cdots, M\} : v_j^+ \not\equiv v_y^-$. Let match$(i, v_y^-)$ be a call to Algorithm 2. Since $\mu(j) = m$ holds, v_j^+ was not removed from V_i^+ for y such $1 \leq y < m$. Since $(v_k^+ \leq v_j^+) \wedge (\mu(k) = t) \wedge (t \geq m)$, we conclude that $D \neq \emptyset$ for $m \leq y \leq t$. Therefore, v_j^+ was not removed from V_i^+ for such invocations. Since $\forall y \in \{t, \cdots, M\} : v_j^+ \not\equiv v_y^-$ holds, we conclude that $v_j^+ \notin Q$ for y such $t < y \leq M$. Therefore, v_j^+ was not removed from V_i^+ for such invocations. Since v_j^+ is not removed after M invocations of Algorithm 2, we have $V_i^+ \neq \emptyset$, and Algorithm 1 returns false.

Thus, Algorithm 1 always returns true for consistent sequences and false for inconsistent sequences. ■

Lemma 3: Algorithm local-behavior-ok, which is described in [10], returns true iff the sequences $(v_1^+, v_2^+, \cdots, v_N^+)$ and $(v_1^-, v_2^-, \cdots, v_M^-)$ are consistent.

Proof: Algorithm local-behavior-ok finds a *proper matching* $\mathcal{M}$ [11] on a bipartite graph (V, E) with $V = V_i^+ \cup V_i^-$, $V_i^+ \cap V_i^- = \emptyset$, and $E = \{(v^+, v^-) \in V_i^+ \times V_i^- : v^+ \equiv v^-\}$ subject to a partial order $\leq$ on the set V_i^+, such that the following holds: 1) $\mathcal{M} \subseteq E$ is a matching; 2) $|\mathcal{M}| = |V_i^+|$; 3) $\mathcal{M} = \{(v^+, v^-) \in E : v^+ \equiv v^-\}$; and 4) $\forall (v_j^+, v_m^-), (v_k^+, v_t^-) \in \mathcal{M} : ((v_j^+ \leq v_k^+) \wedge (m < t)) \vee ((v_k^+ \leq v_j^+) \wedge (t < m))$. Conditions 2 and 3 ensure that $\mathcal{M}$ induces a mapping $\mu : \{1, 2, \cdots, N\} \mapsto \{1, 2, \cdots, M\}$, which is a function $\mu \subseteq R$, i.e Clause 1 from Definition 2 holds. Condition 1 ensures that μ is an injection, i.e. Clause 2 from Definition 2 holds. Condition 4 guarantees that $\forall v_k^+, v_j^+ \in V^+ : (v_k^+ \leq v_j^+) \wedge (\mu(k) = t) \wedge (\mu(j) = m) \Rightarrow (t < m)$, i.e. Clause 3 from Definition 2 holds. ■

Theorem 2: For any cache-coherent memory system and for any MCM not requiring total store ordering[2], Algorithm 3 returns true iff all the MCM's axioms hold for the traces induced by a given test case.

Proof: This theorem is proved in [10] for an algorithm that is the same as Algorithm 3, except that we replace every invocation of local-behavior-ok by an invocation of LOCAL-BEHAVIOR-OK. Since, from Theorem 1 and Lemma 3, those algorithms are indistinguishable for the same sequences, the proof provided in [10] serves as a proof for this theorem. ■

Informally, Theorem 2 means that, for largely relaxed models, when analyzing the behavior induced by a given test case, our technique never overlooks actual errors nor raises apparent errors. After establishing its verification guarantees, we experimentally compared our checker with two post-mortem checkers, as reported in the next section.

[2]E.g. Alpha's and PowerPC's Relaxed Order, Weak Ordering, etc.

VI. Experimental impact

We used the framework GEM5 [12] to build platform instances implementing Alpha's MCM [3]. They were built with distinct numbers of processors ($p \in \{2, 4, 8\}$) for a configuration where (L1) instruction/data caches are private, the (L2) unified cache is shared, and snooping is used for coherence. We generated 240 random-instruction test cases by combining distinct numbers of operations ($n \in \{2K, 4K, 8K, 16K\}$), shared addresses (2, 4, 8, 16, 32), and instruction mixes (4).

Then we modeled ten distinct errors, which are described in Table II. From the correct platform, we derived ten faulty instances, each with a distinct error. Each test case was run on every faulty platform, leading to 2400 use-case scenarios. Each scenario was submitted to our checker and to two post-mortem checkers: a conventional *inference-based checker* (INF) similar to the one described in [2] and a *checker based on extended matching* (EXM), developed in our previous work [10].

As an estimate for error *coverage*, we measured the percentage of the use cases for which an error was detected. As an estimate for verification *time*, we measured the average test case runtime. Since ours is an on-the-fly checker, this time already captures the whole verification effort. However, for the post-mortem checkers, we distinguish verification time from *effort*, which also includes the time to generate the traces.

Fig. 1 shows the impact of test case size for a quad-core system. On average, both ours and EXM found 92% of the errors while INF found 77%. EXM is faster than INF by two orders of magnitude. Our checker reaches the same coverage as EXM's but it requires approximately 1/4 of EXM's average verification time. Even when averaging only the test cases that actually found an error (as if we could optimistically assume full coverage in practice), EXM is still faster than INF by one order of magnitude and ours is 3.5 times faster than EXM.

Fig. 2 shows the impact of processor upscaling for test cases of fixed size ($n = 16K$). Notice that, within the observed range, the effectiveness of our checker, as well as EXM's, is above 90%. Observe that EXM's verification time decreases with the number of processors, since its complexity decreases with p for test cases of same size n (see Table I). Of course, larger number of processors are expected to require larger test cases to keep acceptable coverage, thereby requiring higher verification time. Although the complexity of our checker also decreases with p, a slight increase in verification time is observed. This is due to the fact that, for a given test case size, simulation takes longer for a larger number of processors, because a larger number of threads must

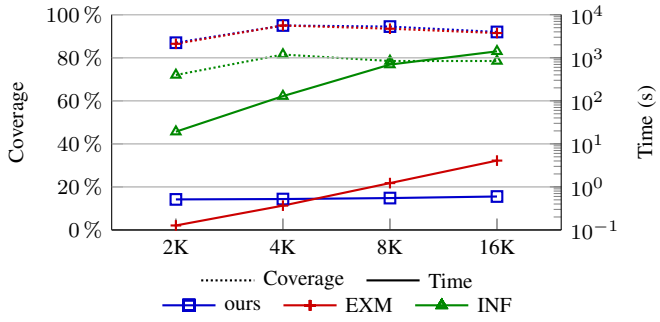

Fig. 1: Impact of increasing test case sizes

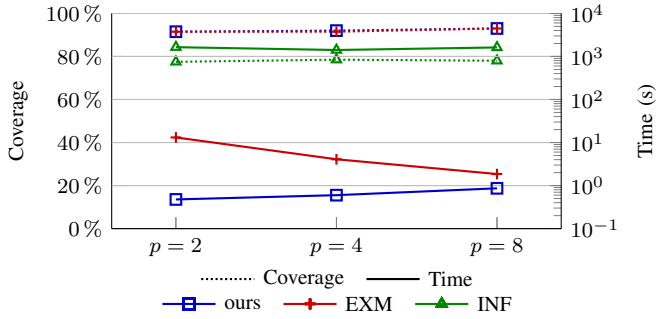

Fig. 2: Impact of processor upscaling

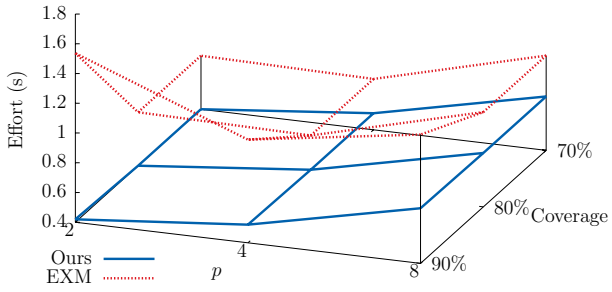

Fig. 3: Impact of coverage and scalability on verification effort

be initialized. Besides, the simulation of the hardware design representation also takes longer. We observed that, for each new processor included in the platform, an overhead of 10% is added to the simulation time.

We also evaluated how efficient the checkers are for reaching the *same coverage* level. We excluded INF from the evaluation, as its effort is orders of magnitude higher than EXM's and ours. For platforms with distinct number of processors, we determined the verification effort to reach the following coverage levels: 70%, 80%, and 90%. Fig. 3 displays the required verification effort as a function of coverage and number of processors (p). Our checker needs approximately 1/4 to 3/4 of the overall verification effort required by EXM to reach the same coverage level when the number of processors increases from 2 to 8. Observe that, for a given p, our checker's verification effort is less sensitive to the coverage level than EXM's. Note that the verification effort of both techniques increases with processor upscaling for two reasons: 1) for a fixed test case size, it takes longer to initialize a larger number of threads and longer to simulate the hardware; 2) *larger test case sizes* are required to reach the same coverage level. The latter explains why the rate of growth is higher in Fig. 3 than it is in Fig. 2 (where test case size is fixed).

These experimental results, combined with the theoretical guarantees, allows us to draw a big picture in the next section.

VII. Conclusions and Future Work

Although conventional checkers based on inferences are crucial to post-silicon testing (due to observability limitations), our experiments showed that their reuse is inadequate for the pre-silicon verification of relaxed MCMs. We showed that the tailoring of consistency checkers to pre-silicon verification pays off, since it leads to speed-ups of 1 or 2 orders of magnitude, as compared to a conventional checker without backtracking [2]. This also allows us to conclude that backtracking does not pay off for pre-silicon verification, since it would lead to even larger runtimes for essentially the same verification guarantees provided by the checker proposed in [10] and by the one proposed in this paper. Besides, backtracking limits the long-term scalability of inference–based checkers to handle largely relaxed MCMs. The proposed technique needs approximately 1/4 to 3/4 of the overall verification effort required by a post-mortem checker with the same scalability.

Although our technique provenly offers superior verification guarantees as compared to the approach proposed in [4], as future work, we intend to compare their verification efforts.

References

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

[2] S. Hangal, D. Vahia, C. Manovit, J.-Y. Lu, and S. Narayanan, "TSOTool: a program for verifying memory systems using the memory consistency model," in *31st ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2004, pp. 114–123.

[3] R. L. Sites and R. T. Witek, *Alpha AXP architecture reference manual (2nd ed.)*. Newton, MA, USA: Digital Press, 1995.

[4] O. Shacham, M. Wachs, A. Solomatnikov, A. Firoozshahian, S. Richardson, and M. Horowitz, "Verification of chip multiprocessor memory systems using a relaxed scoreboard," in *41st ACM/IEEE International Symposium on Microarchitecture (MICRO)*, 2008, pp. 294–305.

[5] C. Manovit and S. Hangal, "Efficient algorithms for verifying memory consistency," in *17th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2005, pp. 245–252.

[6] A. Roy, S. Zeisset, C. Fleckenstein, and J. Huang, "Fast and generalized polynomial time memory consistency verification," in *Computer Aided Verification (CAV)*, ser. Lecture Notes in Computer Science. Springer, 2006, vol. 4144, pp. 503–516.

[7] C. Manovit and S. Hangal, "Completely verifying memory consistency of test program executions," in *IEEE Symposium on High-Performance Computer Architecture (HPCA)*, 2006, pp. 166–175.

[8] Y. Chen, Y. Lv, W. Hu, T. Chen, H. Shen, P. Wang, and H. Pan, "Fast complete memory consistency verification," in *IEEE 15th International Symposium on High Performance Computer Architecture (HPCA)*, 2009, pp. 381–392.

[9] W. Hu, Y. Chen, T. Chen, C. Qian, and L. Li, "Linear time memory consistency verification," *IEEE Transactions on Computers*, vol. 61, no. 4, pp. 502–516, 2012.

[10] E. A. Rambo, O. P. Henschel, and L. C. V. Santos, "On ESL verification of memory consistency for system-on-chip multiprocessing," in *ACM/IEEE Design, Automation, and Test in Europe Conference (DATE)*, 2012, pp. 9–14.

[11] G. Marcilio, L. C. V. Santos, B. Albertini, and S. Rigo, "A novel verification technique to uncover out-of-order DUV behaviors," in *46th ACM/IEEE Design Automation Conference (DAC)*, 2009, pp. 448–453.

[12] N. Binkert, R. Dreslinski, L. Hsu, K. Lim, A. Saidi, and S. Reinhardt, "The M5 simulator: Modeling networked systems," *IEEE Micro*, vol. 26, no. 4, pp. 52–60, 2006.