2021 International Symposium on Theoretical Aspects of Software Engineering (TASE) | 978-1-6654-4163-6/21/$31.00 ©2021 IEEE | DOI: 10.1109/TASE52547.2021.00023

Dynamic Verification of C11 Concurrency over Multi Copy Atomics

Sanjana Singh
Department of CSE
Indian Institute of Technology Delhi
sanjana.singh@cse.iitd.ac.in

Divyanjali Sharma
Department of CSE
Indian Institute of Technology Delhi
divyanjali@cse.iitd.ac.in

Subodh Sharma
Department of CSE
Indian Institute of Technology Delhi
svs@cse.iitd.ac.in

***Abstract*—We investigate the problem of runtime analysis of concurrent *C11* programs under *Multi-Copy-Atomic* semantics (*MCA*). Under *MCA*, one can analyze program outcomes solely through interleaving and reordering of thread events. As a result, obtaining intuitive explanations of program outcomes becomes straightforward. Newer versions of ARM (ARMv8 and later), Alpha, and Intel's x-86 support *MCA*. Our tests reveal that state-of-the-art dynamic verification techniques that analyze program executions under the *C11* memory model report safety property violations that can be interpreted as false alarms under *MCA* semantics. Sorting the true from false violations puts an undesirable burden on the user.**

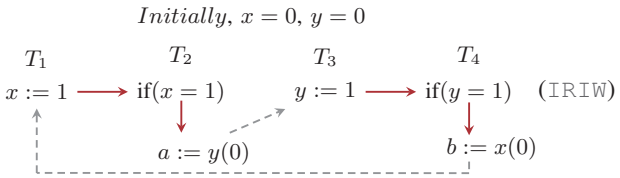
In this work, we provide a dynamic verification technique (`MoCA`) to analyze *C11* program executions which are permitted under the *MCA* model. We restrict *C11* happens-before relation and propose coherence rules to capture precisely those *C11* program executions which are allowed under the *MCA* model. `MoCA`'s exploration of the state-space is based on the state-of-the-art dynamic verification algorithm, source-DPOR. Our experiments validate that `MoCA` captures all coherent *C11* program executions, and is precise for the *MCA* model.

***Keywords*-*C11*; *MCA*; stateless model checking;**

I. Introduction

The relaxed memory orderings over concurrent memory accesses introduced in the C/C++ 2011 ISO standard (*C11*) [1] has been the object of intense study in the past decade The axiomatic specification of *C11* [2] defines relaxed program behaviors as relations over memory accesses along with constraints on stores that can possibly affect each load. The semantics are known to be complex and for a fragment of the standard – *release-acquire* – the state-reachability problem is shown to be undecidable [3]. More notable is that many of the feasible program behaviors under *C11* semantics may never manifest on the underlying architectures.

Therefore, an important desideratum is to engineer analysis tools/techniques to analyze *C11* programs under restricted hardware models with *precision*. In this work, we investigate the problem of precise dynamic analysis of *C11* programs under *Multi-Copy-Atomic* model (*MCA*) – a popular hardware memory model, which is claimed to be supported in Intel's x-86 TSO, newer versions of ARM[1](v8 and later) and Alpha with varying degrees of permitted reorderings. A noteworthy aspect of the *MCA* model is the assumption of a single abstract view of shared memory between processing elements (or threads), leading to the observation that permitted program



behaviors under this model can be explained solely through interleaving and reordering of thread events. As a result, one can use existing dynamic analyses [4][5][6][7] (with suitable adaptations) developed for memory consistency models where program behaviors can be explained by program event interleavings alone.

Several past works on designing efficient dynamic verifiers (also referred to as stateless model checkers) for *C11* concurrency (or its variants) exist such as CDSChecker [10] and GenMC [11]. On the one hand, notwithstanding the sophistication of their algorithms, some of the valid *C11* outcomes flagged by these tools, would not occur when the input program is executed on an *MCA* architecture. Consider the program (`IRIW`). The initial value of x and y is 0 and the value in brackets (*i.e.* $y(0)$ and $x(0)$) indicate the value read in local variables a and b. No two statements in the program can reorder due to the control dependence between the read statements of threads T_2 and T_4. The execution shown in the program with $a{=}0 \wedge b{=}0$ is a valid *C11* outcome if not all statements in the program are `sc` (sequential consistency) ordered. It is so because the dependencies shown via gray dashed arrows may not hold under the *C11* model, thus leading to an acyclic dependence relation (explained further in Section §IV). However, such an outcome cannot be explained through any *acyclic interleaving* of events; therefore, is invalid under the *MCA* model. Hence, if a property violation is detected by a *C11* verification technique it must be scrutinized further for viability under the given hardware memory model. The laborious task of sorting the feasible from infeasible violations is a burden on the user. On the other hand, solutions for *MCA* hardware memory models either ignore the *C11* relaxation directives provided in the program such as [6][7] for x-86 TSO or have been designed for a memory model [12] proposed as a superset of existing hardware models (including

[1]ARMv8 calls its model *other-MCA* [8]. The difference with *MCA* is of terminology, not semantics (as clarified by [9]).

978-1-6654-4163-6/21/$31.00 ©2021 IEEE
DOI 10.1109/TASE52547.2021.00023

ARMv8), namely HMC [13].

Our contribution sits in this cross-section of *C11* program execution on *MCA* architectures. We design a sound and precise dynamic verification technique called MoCA (pronounced as moh·kaa), addressing the problem of *C11* program verification under *MCA*. The key contribution of our work is in restricting *C11* program behaviors to only those that are permitted under the *MCA* model. To accomplish this, we present *happens-before* and *coherence* rules (see §V). Another central contribution of our work is to simulate the reordering of events of a thread (either by the compiler or the hardware) through a new event type, *viz.*, *shadow-writes*. It updates the shared memory for the store instructions in the program.

Our proposed contributions have several merits: (M1) our formalization makes source-DPOR [4] (a powerful stateless model checking algorithm) eminently usable without any modification; (M2) the use of shadow-writes simulates reordering through interleaving and thus allows scheduling of execution sequences that reflect the effects of reordering with simply interleaving of program events.

The remaining paper is organized as follows: §II introduces assumptions and notations used; §III describes the segments of *MCA* operational semantics proposed by Colvin and Smith [14] that are relevant to this work; §IV briefly introduces the *C11* model [1][2]; we present our proposed technique in §V and §VI, where we formally introduce shadow-writes and present MoCA's happens-before relation along with the coherence rules for soundness in §V; in §VI we explain the working of MoCA technique; and, present our experimental observations in §VII. To summarize, in this work we make the following contributions:

- We introduce shadow-writes to precisely capture the reordering semantics of *MCA* through interleaving.
- We propose a restriction of *C11* happens-before relation to disallow non-*MCA* behaviors, and establish coherence rules to ensure exploration of only coherent *C11* execution sequences.
- We establish the correctness and precision of our proposed relations and rules with supporting theorems (see Theorem 1, §V and Theorem 3, §VI).
- We present the MoCA technique that executes source-DPOR utilizing our proposed happens-before restriction (see §VI).
- We present a prototype implementation to validate our technique and perform experiments to evaluate our claims empirically (see §VII).

II. Preliminaries

Concurrent program model: We consider an acyclic multi-threaded program, P, as a finite set of program threads. A thread i in P, uniquely identified by tid_i, has deterministic computation and terminating executions. The threads in P access a fixed set of memory locations called objects (denoted by $\mathcal{O}$). Each thread executes a sequence of *events* that, in essence, are *actions* on these objects. The set of actions (denoted by $\mathcal{A}$) include: *write*, *read*, *rmw* (or read-modify-write), and *fence*. We extend this set with a special action called *shadow-write*. In our model of computation, each *write* or *rmw* action is now associated with a corresponding *shadow-write* action. This action updates the shared memory with the value of its corresponding write or rmw.

Let Σ be the set of global states of P with a given initial state $s_0 \in \Sigma$. We assume the standard definition of a state, *i.e.*, valuation of all shared and local objects of P, and the program counter of all threads. In a global state $\sigma \in \Sigma$, **shr** σ denotes the shared component comprising of (shared object, value) pairs, and **lcl** σ denotes the local component comprising of (local object, value) pairs.

Program execution and events. The set of all events of a program P is denoted by $\mathcal{E}$. An execution of P is, therefore, a sequence of events $\tau{=}e_1.e_2.\ldots.e_n$ s.t. $e_i \in \mathcal{E}$, for $i \in \{1, 2, \ldots, n\}$. The sequence of events of a thread tid_i is denoted by $tid_i{:}\tau$. The system upon execution of e_i (which is a sequence of internal operations on local objects followed by a single operation on a shared object) transitions from a state s_{i-1} to the next state s_i (denoted by $s_{i-1} \xrightarrow{e_i} s_i$). Note that the event e_i must be *enabled* in state s_{i-1} for the transition to take place. Let $s_{[\tau]}$ represent the state reached after exploring the execution sequence τ and $\texttt{prefix}_{[\tau]}(e)$ represent the prefix of τ up to (but not including) e. Two events $e', e \in \mathcal{E}_\tau$ (where $\mathcal{E}_\tau$ denotes the set of events in τ) are related by a total order $<_\tau$. For instance, $e'{<_\tau}e$ denotes that e' occurs before e in τ. An empty sequence is represented as $\langle\rangle$.

An event from thread thr at index idx in an execution τ is a tuple $\langle thr, act, obj, ord, idx\rangle$, where $act \in \mathcal{A}$ represents the action on a set of objects $obj \subseteq \mathcal{O}$ under the memory ordering constraint $ord \in \mathcal{M}$. Observe that obj can potentially be a non-singleton for rmw events and empty set for fences. The projections $thr(e)$, $act(e)$, $obj(e)$, $ord(e)$, and $idx(e)$ return the respective tuple elements of e. With the exception of $obj(e)$ when e is an rmw event, all other projections are straightforward to interpret. When an rmw event e is a read of o_1 and write of o_2, then $obj(e)$ returns o_1 when e is analysed as a read event, and o_2 otherwise.

Memory ordering constraints. Under the *C11* model each event has an associated *memory order*. A memory order specifies the ordering possibilities of an (atomic or non-atomic) event around an atomic event; thereby restricting the freedom available to compilers and underlying systems to reorder events. Let $\mathcal{M}$ = {na, rlx, rel, acq, acq-rel, sc} represent the set of memory orders – relaxed (rlx), release (rel), acquire/consume (acq), acquire-release (acq-rel) and sequentially-consistent (sc) – provided for atomic events and na representing a non-atomic access. Let $\sqsubset \subseteq \mathcal{M}\times\mathcal{M}$ be a relation on memory orders such that $m_1 \sqsubset m_2$ denotes that m_2 is stricter than m_1; thus, m_2 may restrict certain program outcomes that are otherwise possible with m_1. The memory orders in $\mathcal{M}$ are related as na $\sqsubset$ rlx $\sqsubset$ {acq, rel} $\sqsubset$ acq-rel $\sqsubset$ sc. Accordingly, $\sqsubseteq$ represents strict or stricter ordering. For instance, rel $\sqsubseteq$ rel. We overload the operators $\sqsubset$, $\sqsubseteq$ as unary operators that return the set of memory orders

that are weaker. For instance, $\sqsubseteq$rel = {na, rlx, rel }. The set of events pertaining to a memory order m is represented as $\mathcal{E}^{(m)}$. We also use $\mathcal{E}^{(\sqsubset m)}$ (or $\mathcal{E}^{(\sqsubseteq m)}$) to represent the set of $\mathcal{E}^{(m_1)} \cup ... \cup \mathcal{E}^{(m_N)}$ where $m_i \sqsubset m$ (or $m_i \sqsubseteq m$), eg $\mathcal{E}^{(\sqsubseteq \texttt{rlx})} = \mathcal{E}^{(\texttt{na})} \cup \mathcal{E}^{(rlx)}$. Similarly, $\mathcal{E}^{(\sqsupset m)}$ (or $\mathcal{E}^{(\sqsupseteq m)}$) can be interpreted as union of sets of events with ordering annotations stricter (or at least as strict) than (as) m.

A note on event categories. For ease of explanation, we categorise the set of events in $\mathcal{E}$ into: (i) set of writes ($\mathcal{E}^{\mathbb{W}}$) that issue a write (*i.e.*, write events and rmws), (ii) the set of modifiers ($\mathcal{E}^{\mathbb{M}}$) that update the shared memory for an issued write (ie, shadow-write events), (iii) the set of reads ($\mathcal{E}^{\mathbb{R}}$) (*i.e.*, read events and rmws), and (iv) the set of program fences ($\mathcal{E}^{\mathbb{F}}$). Note that the shared-read events and shared-write events (including corresponding shadow-writes) can either be atomic or non-atomic in nature. We use $\mathcal{E}^{\mathbb{W}}_\tau, \mathcal{E}^{\mathbb{R}}_\tau, \mathcal{E}^{\mathbb{M}}_\tau$ and $\mathcal{E}^{\mathbb{F}}_\tau$ for the respective categories of the events in an execution τ. Similarly, we write $\mathcal{E}^{\mathbb{W}(m)}_\tau, \mathcal{E}^{\mathbb{R}(m)}_\tau, \mathcal{E}^{\mathbb{M}(m)}_\tau$ and $\mathcal{E}^{\mathbb{F}(m)}_\tau$ to denote the set of events with associated memory order m of the respective action categories in the execution sequence τ.

Traces and equivalence relation: Given τ, if reversing the order of execution of two co-enabled events $e', e \in \mathcal{E}_\tau$ does not change the outcome of the program then e', e are called *independent* events [15]. All sequences that differ only in the order of independent events are called *equivalent* sequences. We use $\tau_1 \sim \tau_2$ to denote that τ_1 and τ_2 are equivalent. A set of equivalent sequences form an equivalence class known as a program *trace* that represents a specific behavior of the program. A sound analysis must explore at least one sequence from each trace of P.

III. Multi Copy Atomic model

Under the *MCA* model if a write to object o from tid_i is *observed* by a different thread, tid_j, then the write is *coherently* observable to all other threads that access the object o. It is, however, permitted for a thread to observe its own writes prior to making them visible to other threads in the system. The term *observed* refers to tid_j becoming aware of a write from tid_i, either directly when a read of tid_j reads-from the write or indirectly through a chain of intra- and inter-thread dependencies [8]. We formally revisit the term *observed* and *coherent* access in §V after formally defining event interactions.

We adapt the *MCA* model formalized by Colvin and Smith [14] as a basis for our technique that relies on *shadow-writes*. The model allows for a sequence of events $tid_i{:}\tau$ of thread tid_i to be reordered to a sequence $tid_i{:}\tau'$. Consider two events, $e', e \in \mathcal{E}_{tid_i:\tau}$. Assume that e originally occurred *after* e'. The reordering such that e now occurs *before* e' is represented by $e' \overset{R}{\Leftarrow} e$. The reordering $e' \overset{R}{\Leftarrow} e$ can take place only if e', e have distinct local and shared variables. To ensure a semantic preserving reordering, the following must hold:

- spr1 $\mathcal{E}_{tid_i:\tau'} = \mathcal{E}_{tid_i:\tau}$ (the event sets are the same)
- spr2 each read event of $tid_i{:}\tau$ must have the same set of writes to read from in $tid_i{:}\tau$ as well as $tid_i{:}\tau'$ (we call it *thread semantics*)

$$\frac{tid_i{:}e.\tau' \xrightarrow{e} tid_i{:}\tau \quad e' \overset{R}{\Leftarrow} e}{tid_i{:}e'.e.\tau' \xrightarrow{e} tid_i{:}e'.\tau} \quad \texttt{(reordering)}$$

$$\frac{p_1 \xrightarrow{e} p_1'}{p_1 \| p_2 \xrightarrow{e} p_1' \| p_2} \quad \frac{p_2 \xrightarrow{e} p_2'}{p_1 \| p_2 \xrightarrow{e} p_1 \| p_2'} \quad \texttt{(parcom)}$$

$$\frac{\tau' \xrightarrow{r:=x} \tau \quad \mathbf{shr}\ \sigma(x) = v}{(\mathbf{lcl}\ \sigma{\cdot}\tau') \xrightarrow{[x=v]} (\mathbf{lcl}\ \sigma_{[r:=v]}{\cdot}\tau)} \quad \texttt{(r-shared)}$$

$$\frac{\tau' \xrightarrow{x:=r} \tau \quad \sigma(r) = v}{(\mathbf{lcl}\ \sigma{\cdot}\tau') \xrightarrow{x:=v} (\mathbf{lcl}\ \sigma{\cdot}\tau)} \quad \texttt{(w-issue)}$$

$$\frac{p \xrightarrow{tid_i::x:=v} p'}{(\mathbf{shr}\ \sigma{\cdot}\ p) \xrightarrow{*} (\mathbf{shr}\ \sigma_{[x:=v]}{\cdot}\ p')} \quad \texttt{(w-update)}$$

Fig. 1: Semantics of *MCA* model [14]

- spr3 $tid_i{:}\tau'$ preserves the order of updates and accesses of each shared variable with respect to $tid_i{:}\tau$ (we call it *coherence-per-location*).

The following language represents an *MCA* model [14].

processing element, $p := (\mathbf{tid}_N\ \mathbf{lcl}\ \sigma \cdot tid_N{:}\tau) \mid p1 \parallel p2$
system, $s := (\mathbf{shr}\ \sigma \cdot p)$

The key element of the language is a *processing element*. It is identified by a unique identifier ($\mathbf{tid}_N$), the local state ($\mathbf{lcl}\ \sigma$), and a sequence of events to be executed ($tid_N{:}\tau$). The entity *system* is identified with a shared state ($\mathbf{shr}\ \sigma$) and a parallel composition of processing elements. Thus, a system term is of the form $(\mathbf{shr}\ \sigma{\cdot}(\mathbf{tid}_0\ \mathbf{lcl}\ \sigma{\cdot}tid_0{:}\tau_0) \parallel (\mathbf{tid}_1\ \mathbf{lcl}\ \sigma{\cdot}tid_1{:}\tau_1) \parallel \ldots)$. The operational semantics for a program P under *MCA* is listed in Fig. 1.

***MCA* semantics.** The `(reordering)` rule states that if an event e can reorder before another event e' (where $e' <_{tid_i:\tau} e$) then the processing element tid_i can execute e before e' and suitably update the remaining sequence to be executed later. The rule `(parcom)` shows the parallel composition of the processing elements. It states that one step of the system is taken by one processing element at a time. The `(r-shared)` rule captures the read of a shared variable from the shared storage into a local variable. The `(w-issue)` rule shows that a processing element initiates a write operation of value v to a shared variable x, and moves to the next event. A write initiated by tid_i is updated to the shared storage by the system as shown in rule `(w-update)`. Notably, this rule captures the effects of *shadow-write* events.

IV. *C11* Memory Model Semantics

In *C11* memory model the behavior of an execution, τ, is usually defined through an acyclic and irreflexive happens-before relation ($\rightarrow_\tau^{[c]::\mathbf{hb}} \subseteq \mathcal{E}_\tau \times \mathcal{E}_\tau$). The *C11* model defines its happens-before relation $\rightarrow^{[c]::\mathbf{hb}} = \rightarrow_\tau^{[c]::\mathbf{sb}} \cup \rightarrow_\tau^{[c]::\mathbf{ithb}}$, where: (i) $\rightarrow_\tau^{[c]::\mathbf{sb}}$ (*Sequenced-before*) is the intra-thread order on events, and (ii) $\rightarrow_\tau^{[c]::\mathbf{ithb}}$ (*Inter-thread hb*) is the relation between events of different threads (say tid_i, tid_j) formed by a transitive closure of $\rightarrow_\tau^{[c]::\mathbf{sb}} \cup$ synchronizations (that occur

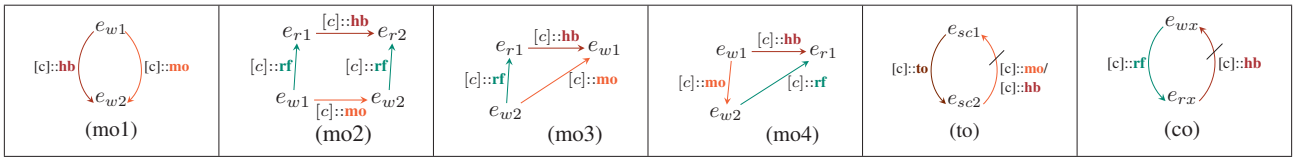


Fig. 2: *C11* coherence rules

when a read of tid_i with an ordering $\sqsupseteq$acq reads from a write event in a *release sequence* of some event e in tid_j) [1].

In addition, all write events of an object in a sequence τ must be related by a total order called *modification-order* ($\rightarrow_\tau^{[c]::\textbf{mo}} \subseteq \mathcal{E}_\tau^{\mathbb{W}} \times \mathcal{E}_\tau^{\mathbb{W}}$). The $\rightarrow_\tau^{[c]::\textbf{mo}}$ relation is fundamentally involved in specifying a set of sufficient conditions which ensure that a *C11* program execution is *coherent*. For ease of understanding, we present diagrammatic representations of these conditions in Fig. 2 (the rules are formally presented in the extended version [16]). The conditions are as follows:

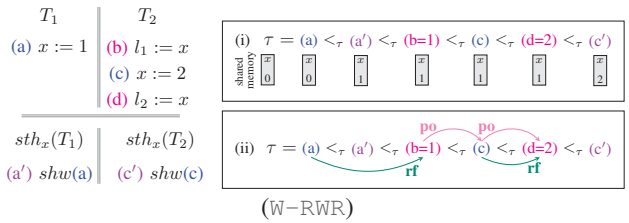
- (mo1): hb-ordered writes are also mo-ordered;
- (mo2): a read e_{r2} hb-ordered after another read e_{r1}, either reads-from the same source as e_{r1}'s or from a source mo-ordered after the source of e_{r1};
- (mo3): a read hb-ordered before a write reads-from a write mo-ordered before that write;
- (mo4): a read hb-ordered after a write either reads-from the write or from a write mo-ordered after that write;
- (to): all sc ordered events must form a total-order ($\rightarrow_\tau^{[c]::\textbf{to}}$) *wrt* $\rightarrow_\tau^{[c]::\textbf{mo}}$ and $\rightarrow_\tau^{[c]::\textbf{hb}}$; and,
- (co): a read must take its data from a write event occurring in the trace. The (co) rule ensures that a read does not take a value from thin-air *i.e.* a value not generated in the program execution.

Notice that shadow-write events are a particular constuct of our technique; naturally, definitions of *C11* relations do not contain them. In §V, we shall present one of our main contributions of redefining the above-mentioned relations (keeping shadow-write events in consideration) which admit only the *MCA* behaviors of a *C11* program.

Reordering restrictions. For any two events $e', e \in \mathcal{E}_\tau$ s.t. $thr(e) = thr(e') \wedge e' <_\tau e$, we have the following: if $e \in \mathcal{E}^{\mathbb{W}(\sqsupseteq \texttt{rel})}$, then it restricts events such as e' from reordering after it. We denote this downward reordering restriction on e' by e as $\nmid(e, e')$. Similarly, if $e' \in \mathcal{E}^{\mathbb{R}(\sqsupseteq \texttt{acq})}$, it restricts a *later* event e from reordering before it. We denote this upward reordering restriction on e by e' as $\nmid(e', e)$. Furthermore, *C11* also disallows reordering of events e', e that share program dependence (such as data, address and control), which we represent by $\texttt{dep}(e', e)$.

V. *MCA* RESTRICTION FOR *C11*

We (i) re-formulate *C11*'s $\rightarrow_\tau^{[c]::\textbf{hb}}$ relation and (ii) define trace coherence rules to accurately recognize *C11* program traces admissible under *MCA*. A principal contribution of our technique is the introduction of a new event type called *shadow-writes* that simulate reordering through interleaving as explained below. Shadow-writes break a write operation into two (not necessarily consecutive) events: (i) the write event from the program that is visible only to events of the same thread and (ii) the shadow-write event that updates the shared memory with the write event's value at a later timestamp; thus, completing the write operation and making it visible to all threads. In order to issue shadow-write events, we introduce the notion of *shadow-threads*. We maintain a separate shadow-thread per program thread per object. For an event $e \in \mathcal{E}^{\mathbb{W}}$, we use $shw(e)$ to represent the shadow-write event associated with e. Similarly, $prw(e')$ denotes the write event corresponding to the shadow-write $e' \in \mathcal{E}^{\mathbb{M}}$. The set of shadow-threads associated with $thr(e)$ is denoted by $sth(e)$. Note that shadow-writes of the threads in $sth(e)$ can interleave with the events of $thr(e)$ thereby enabling reordering through interleaving.



(W−RWR)

Consider the example (W−RWR). Events labeled (a′) and (c′) are the shadow-writes corresponding to the writes labeled (a) and (c), respectively. The shadow-threads $sth_x(T_1)$ and $sth_x(T_2)$ execute the shadow-write events. W−RWR(i) shows an execution sequence τ where updates to the memory by shadow-writes are illustrated.

As our second central contribution to realize the *MCA* restriction of *C11* model, we define the relation $\rightarrow_\tau^{\textbf{rf}}$ based on shadow-writes. Let $\texttt{LastW}_{[\tau]}(o)$ represent the write corresponding to the latest shadow-write of o in τ.

Definition 1. **($\rightarrow_\tau^{\textbf{rf}}$ relation)** For $e_r \in \mathcal{E}_\tau^{\mathbb{R}}$, its $\rightarrow_\tau^{\textbf{rf}}$ relation is now defined as:
(general case) $\texttt{LastW}_{[\tau']}(o) \rightarrow_\tau^{\textbf{rf}} e_r$; unless
(special case) $\exists e_w \in \mathcal{E}_\tau^{\mathbb{W}}$, which is the latest write from $thr(e_r)$ of object $obj(e_r)$ s.t. $shw(\texttt{LastW}_{[\tau']}(o)) <_\tau e_w <_\tau e_r$ then $e_w \rightarrow_\tau^{\textbf{rf}} e_r$.

Intuitively, a read event takes the data from the last write whose shadow-write *updated* the shared memory, unless there is a later write from the same thread.

Consider the execution sequence τ in example W−RWR(ii). The event (b) reads from (a) (since $\texttt{LastW}_{[\texttt{a.a'}]}(x)$ = (a)); however, for event (d) $\texttt{LastW}_{[\texttt{a.a'.b.c}]}(x)$ = (a) but write (c) from same thread occurs after (a), thus, (d) reads from (c).

HB relation. Based on $\rightarrow_\tau^{\textbf{rf}}$ and *C11*'s *Release Sequence* [1] we tweak $\rightarrow_\tau^{[c]::\textbf{hb}}$, and its constituent relations [1] to

$e' \rightarrow_\tau^{po} e$ if $e' \rightarrow_\tau^{[c]::sb} e \vee act(e') = act(e)$ = shadow-write $\wedge\ thr(e') = thr(e) \wedge idx(e') < idx(e)$	$e' \rightarrow_\tau^{ithb} e$ if $e' \rightarrow_\tau^{sw} e \vee e' \rightarrow_\tau^{dob} e \vee \exists e''$ s.t. $(e' \rightarrow_\tau^{sw} e'' \wedge e'' \rightarrow_\tau^{po} e) \vee (e' \rightarrow_\tau^{po} e'' \wedge e'' \rightarrow_\tau^{ithb} e) \vee (e' \rightarrow_\tau^{ithb} e'' \wedge e'' \rightarrow_\tau^{ithb} e)$
$e' \rightarrow_\tau^{sw} e$ if $e' \in \mathcal{E}_\tau^{\mathbb{W}(\sqsupseteq \mathtt{rel})}, e \in \mathcal{E}_\tau^{\mathbb{R}(\sqsupseteq \mathtt{acq})} \wedge e' \rightarrow_\tau^{rf} e$	$e' \rightarrow_\tau^{hb} e$ if $e' \rightarrow_\tau^{po} e \vee e' \rightarrow_\tau^{ithb} e$
$e' \rightarrow_\tau^{dob} e$ if $e' \in \mathcal{E}_\tau^{\mathbb{W}(\sqsupseteq \mathtt{rel})}, e \in \mathcal{E}_\tau^{\mathbb{R}(\sqsupseteq \mathtt{acq})} \wedge \exists e'' \in \mathcal{E}_{\mathrm{RS}_\tau(e')}$ s.t. $e'' \rightarrow_\tau^{rf} e$	$e' \twoheadrightarrow_\tau e$ if $e' \rightarrow_\tau^{hb} e \wedge \neg\ (e' \rightarrow_\tau^{sw} e \vee e' \rightarrow_\tau^{dob} e)$

Fig. 3: $\rightarrow_\tau^{hb}$ relation

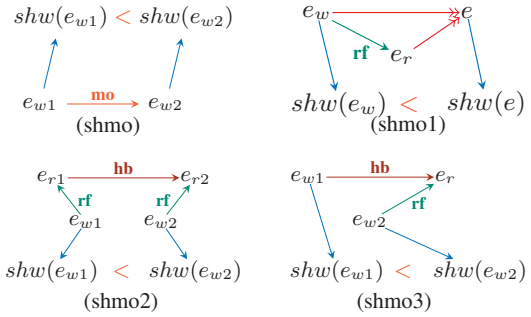


Fig. 4: $\rightarrow_\tau^{rf}$ based $\rightarrow_\tau^{mo}$ in MoCA

define a happens-before relation ($\rightarrow_\tau^{hb}$) for MoCA. Note that $\rightarrow_\tau^{hb} \subseteq <_\tau$. For the purpose of recognizing *C11* behaviors relevant to *MCA*, MoCA defines the following relations (refer Fig. 3): program-order ($\rightarrow_\tau^{po}$)[2], dependency-ordered-before ($\rightarrow_\tau^{dob}$), synchronizes-with ($\rightarrow_\tau^{sw}$) and inter-thread-happens-before ($\rightarrow_\tau^{ithb}$). The relation *non-racing*-hb ($\twoheadrightarrow_\tau$) relates hb-ordered events that do not race to access an object either due to $\rightarrow_\tau^{po}$ ordering or due to synchronization between their respective threads through events related by $\rightarrow_\tau^{sw}$ or $\rightarrow_\tau^{dob}$. The relations $\rightarrow_\tau^{sw}$ and $\rightarrow_\tau^{dob}$ are also formed between $\mathcal{E}^{\mathbb{F}}$, similar to *C11*'s relations. For brevity we skip the details in the paper, please refer the extended version [16] for additional details.

Coherence rules. Read value coherence for MoCA is a derivative of $\rightarrow_\tau^{rf}$ and is interpreted as:

- (shco) a read event must take its value either from (i) a valid write event whose shadow-write does not occur after the read, or (ii) from a valid write event of the same thread that does not occur after it (refer Fig. 5 for formal definition).

The (shco) rule disallows the discussed behavior in IRIW (§I). We introduce a set of MoCA-*mo rules* based on which MoCA determines the order of occurrence of shadow-writes of an object and help determine $\rightarrow_\tau^{mo}$; they are formally defined in Fig. 5 and represented diagrammatically in Fig. 4.

- (shmo): writes e_{w1}, e_{w2} are mo-ordered if $shw(e_{w1})$ occurs before $shw(e_{w2})$.
- (shmo1): if a write e_w is $\twoheadrightarrow_\tau$ or $\rightarrow_\tau^{rf}$ ordered with event e from another thread, then either $shw(e_w)$ must occur before $shw(e)$ (if $e \in \mathcal{E}^{\mathbb{W}}$) or before e (if $e \notin \mathcal{E}^{\mathbb{W}}$);
- (shmo2): if a read e_{r1} is hb-ordered before another read e_{r2}, then the shadow-write of e_{r1}'s source must occur before the shadow-write of e_{r2}'s source;
- (shmo3): shadow-write of a read's source must occur after shadow-writes of all writes hb-ordered before the read.
- (shrmo): To ensures atomicity, each rmw event must read-from the *immediately* ordered before event in the modification order.

The above rules assist MoCA in constructing coherent *C11* sequences. Notably MoCA also maintains a total order relation $\rightarrow_\tau^{to}$ on sc events. It does so in the following way: (i) all sc events in $\rightarrow_\tau^{po}$ are also in $\rightarrow_\tau^{to}$ relation, and (ii) sc ordered events from different threads are in $\rightarrow_\tau^{to}$ by their occurrence order, except write events that are in $\rightarrow_\tau^{to}$ by the occurrence order of their shadow-write. Based on $\rightarrow_\tau^{to}$ coherence on sc events is maintained by rule (shto): all sc ordered events must form a total-order ($\rightarrow_\tau^{to}$) *wrt* $\rightarrow_\tau^{mo}$ and $\rightarrow_\tau^{hb}$.

A maximal sequence, τ, and the associated $\rightarrow_\tau^{hb}$ are coherent and represent a MoCA trace if (shmo), (shmo1), (shmo2), (shmo3), (shrmo), (shco) and (shto) are satisfied by τ. Finally, through Theorem 1, we demonstrate that traces generated by MoCA are indeed coherent under *C11*.

Theorem 1. $\forall \tau,\ \rightarrow_\tau^{hb} \subseteq \rightarrow_\tau^{[c]::hb}$

Proof: The coherence rules of MoCA satisfy the coherence rules of *C11* [1]. Thus, MoCA traces are coherent *C11* traces. See the extended version [16] for formal proof.

VI. *C11-MCA* AWARE SOURCE-DPOR

MoCA explores all relevant program behaviors for detecting safety assertion violations as well as non-atomic (na) data races. Central to MoCA is source-DPOR (Algorithm 1 of [4]), which is a near-optimal improvement over DPOR [5]. It is noteworthy that source-DPOR algorithm used in MoCA is *as is*, *i.e.*, without any modification. This was feasible because of several reasons:

- our design of a valid happens-before relation (Theorem 1) for restricting *C11* under *MCA* is directly pluggable in source-DPOR,
- our proposal of shadow-threads and shadow-writes makes it possible to avoid reordering instructions from a thread during exploration and rely on interleaving model of computation alone, and
- the parallel composition rule (parcom)(§III) satisfies the requirement of source-DPOR that only one thread executes at a time.

Source-DPOR is a non-chronological depth-first search of a directed acyclic graph of execution states. Much like the quintessential DPOR [5], source-DPOR maintains set of events that should be explored at each state and a set of *sleeping* threads. However, unlike the classical DPOR, source-DPOR

[2]Some events of a thread are not ordered by $\rightarrow_\tau^{[c]::sb}$ (eg operands of ==). We assume a total order ($\rightarrow_\tau^{po}$) on the events of a thread, similar to [10], [17].

(shco)	$\forall e_r \in \mathcal{E}^{\mathbb{R}}_\tau$, if $\tau' = \texttt{prefix}_{[\tau]}(e_r)$ then, $\exists e_w \in \mathcal{E}^{\mathbb{W}}_{\tau'}$ s.t. $(thr(e_w) = thr(e_r) \ \vee \ shw(e_w) \in \mathcal{E}^{\mathbb{M}}_{\tau'})$, e_r takes its data from $e_w \ \wedge \ e_r \not\rightarrow^{\textbf{hb}}_\tau e_w$
(shmo)	$\forall e', e \in \mathcal{E}^{\mathbb{M}}_\tau$ s.t. $obj(e') = obj(e), \ e' <_\tau e \Rightarrow prw(e') \rightarrow^{\textbf{mo}}_\tau prw(e)$
(shmo1)	$\forall \ e_w \in \mathcal{E}^{\mathbb{W}}_\tau, \ e_r \in \mathcal{E}^{\mathbb{R}}_\tau, \ e \in \mathcal{E}_\tau, \ \ $ s.t. $thr(e) \neq thr(e_w), \ e_w \twoheadrightarrow_\tau e \ \vee \ e_w \rightarrow^{\textbf{rf}}_\tau e_r \twoheadrightarrow_\tau e \ \Rightarrow \ shw(e_w) <_\tau e$ (if $e \notin \mathcal{E}^{\mathbb{W}}_\tau$) $\wedge$ $shw(e_w) <_\tau shw(e)$ (if $e \in \mathcal{E}^{\mathbb{W}}_\tau$)
(shmo2)	$\forall e_{r1}, e_{r2} \in \mathcal{E}^{\mathbb{R}}_\tau$ s.t. $e_{r1} \rightarrow^{\textbf{hb}}_\tau e_{r2}$ if $\exists e_{w1}, e_{w2} \in \mathcal{E}^{\mathbb{W}}_\tau$ s.t. $e_{w1} \rightarrow^{\textbf{rf}}_\tau e_{r1} \wedge e_{w2} \rightarrow^{\textbf{rf}}_\tau e_{r2}$ where $e_{w1} \neq e_{w2}$ then $shw(e_{w1}) <_\tau shw(e_{w2})$
(shmo3)	$\forall e_{w1}, e_{w2} \in \mathcal{E}^{\mathbb{W}}_\tau, e_r \in \mathcal{E}^{\mathbb{R}}_\tau$ s.t. $e_{w1} \rightarrow^{\textbf{hb}}_\tau e_r \wedge e_{w2} \rightarrow^{\textbf{rf}}_\tau e_r$ where $e_{w1} \neq e_{w2}$, $shw(e_{w1}) <_\tau shw(e_{w2})$
(shrmo)	$\forall e \in \mathcal{E}_\tau, act(e) =$ rmw, $\exists e_w \rightarrow^{\textbf{rf}}_\tau e$ s.t. $e_w \rightarrow^{\textbf{mo}}_\tau e \wedge \nexists e'_w$ s.t. $e_w \rightarrow^{\textbf{mo}}_\tau e'_w \rightarrow^{\textbf{mo}}_\tau e$

Fig. 5: Coherence rules for traces in MoCA

computes much compact set of possible starts from a state than *persistent sets* [15]. We invite the reader to refer to [4] for source-DPOR details.

Shadow-threads and shadow-writes: The shadow-threads introduced in our technique are handled in the following way so that source-DPOR algorithm can be used *as is*: whenever an event $e \in \mathcal{E}^{\mathbb{W}}$ of thread tid_i is executed from a state $s_{[\tau]}$, a corresponding shadow-write event $shw(e)$ is generated and added to the shadow-thread tid_{si} corresponding to $obj(e)$. Similar to program threads, execution of an enabled shadow-write $shw(e)$ of a shadow-thread tid_{si} from a state $s_{[\tau]}$ enables the next event of tid_{si} at state $s_{[\tau.shw(e)]}$.

The shadow-writes, as remarked before, enable reordering through interleaving. Note, however, that a shadow-write is created only after a corresponding program write has occurred. As a result shadow-writes simulate the reordering of program writes with *later* events from the same thread. An important question that arises is: how MoCA covers the case of a **program write reordering with an *earlier* program event** from the same thread? MoCA implicitly assumes that the writes are at the earliest location in the program possible (where earlier refers to a lower event index). To meet this requirement MoCA performs a static *early-write* transformation from input program P to $\widehat{P}$.

Early-write: The transformation rules for each thread sequence $tid_i{:}\tau$ of the original program, P, are:

if there exists a corresponding thread sequence $tid_i{:}\tau'$ of the transformed program then,

- ewt1 $\mathcal{E}_{tid_i:\tau} = \mathcal{E}_{tid_i:\tau'}$ (*i.e.*, same event sets but their order of occurrence may vary);
- ewt2 if $\exists e_1, e_2$ s.t. $e_1 <_{tid_i:\tau} e_2 \ \wedge \ e_2 <_{tid_i:\tau'} e_1$, then $e_2 \in \mathcal{E}^{\mathbb{W}} \ \wedge \ (\nexists e_3 \in \mathcal{E}_{tid_i:\tau}$ s.t. $e_1 \leq_{tid_i:\tau} e_3 <_{tid_i:\tau} e_2 \ \wedge \ (\texttt{dep}(e_3, e_2) \ \vee \ \sharp(e_3, e_2)))$ (*i.e.*, e_2 is a write and can reorder above e_1 only if there is no intervening e_3 that either introduces program dependency with e_2 or creates upward reordering restriction).

We show through Theorem 2 that early-write transformation does not alter the semantics of P. As a result, instead of P, the transformed program $\widehat{P}$ is provided to the source-DPOR algorithm as input.

Theorem 2. *Early-write (P to $\widehat{P}$) is semantics preserving.*

Proof: We show that rules ewt1 and ewt2 are compliant with semantic preserving reordering rules spr1, spr2 and spr3 defined for *MCA* model in §III; Refer the extended version [16] for details.

Via Theorem 1, we established that any trace of P that MoCA examines is a valid *C11* trace. However, to show that MoCA explores precisely the *MCA* traces of *C11* program, we present Theorem 3.

Theorem 3. *MoCA traces are equivalent to C11 traces valid over MCA*

Proof: *Case* $\leftarrow$: $\forall e', e$ s.t. $\neg(e' \overset{R}{\Leftarrow} e) \Rightarrow dep(e', e) \vee \sharp(e', e) \Rightarrow$ if $e' <_{tid_i:\tau} e$ then $e' <_{tid_i:\tau'} e$. Further, if $e' \in \mathcal{E}^{\mathbb{W}}_\tau$, e can observe its effect. (by construction of $\rightarrow^{\textbf{rf}}_\tau$ and shadow-threads). Thus, reordering restricted by *C11* over *MCA* is also restricted by MoCA. *inf* (i). As early-write is semantic preserving, thus reordering allowed under MoCA is allowed under *MCA*. *inf* (ii). $e_w \in \mathcal{E}^{\mathbb{W}}$ along with $shw(e_w)$ perform (w-issue) and (w-update) while preserving semantics (using (shco)). $e_r \in \mathcal{E}^{\mathbb{R}}$ perform (r-shared). Source-DPOR algorithm ensures (parcom). *inf* (iii). Thus, from *inf* (i), (ii), (iii), traces of MoCA have equivalent *C11 MCA* traces.

Case $\rightarrow$: If $e' <_{tid_i:\tau} e$ and $e' <_{tid_i:\tau'} e$ then $\texttt{dep}(e', e) \vee \sharp(e', e) \Rightarrow \neg(e' \overset{R}{\Leftarrow} e)$ and reordering of e' and e is not supported by *C11*. Hence, reordering restricted by MoCA is also restricted by *C11* under *MCA*. *inf* (iv). $\forall e' \overset{R}{\Leftarrow} e \Rightarrow obj(e') \neq obj(e)$ and thus effect of e', e can interleave under MoCA. Hence, reordering allowed under *MCA* are allowed by MoCA. *inf* (v). (r-shared) is performed by $\mathcal{E}^{\mathbb{R}}$, (w-issue) and (w-update) by $\mathcal{E}^{\mathbb{W}}$ and $\mathcal{E}^{\mathbb{M}}$, (parcom) is ensured by source-DPOR algorithm. *inf* (vi). Thus, from *inf* (iv), (v), (vi) *C11 MCA* traces are valid MoCA traces.

Design complexity: The complexity of source-DPOR algorithm is $O(T^2|\mathcal{E}|^2 S)$, where T is the number of threads and S is the number of sequences explored. The relation $\rightarrow^{\textbf{hb}}_\tau$ has the same computational complexity as in the original source-DPOR work, *i.e.* $O(|\mathcal{E}|^2)$. Addition of shadow-threads, however, makes the worst-case complexity of MoCA $O(|\mathcal{O}|^2 T^2 |\mathcal{E}|^2 S)$.

VII. Experimental Validation

Implementation details. We present a prototype implementation to experimentally validate MoCA technique. The implementation is built on rInspect [7]. MoCA-tool takes a *C11* program as input. The input program P is statically transformed to $\widehat{P}$ by the early-write transformation (§VI). Further, $\widehat{P}$ is instrumented using LLVM to recognize newly enabled events dynamically during execution of a sequence. The events are communicated to MoCA scheduler, which orchestrates the order of execution of events using source-DPOR algorithm. MoCA-tool re-runs $\widehat{P}$ for every maximal sequence explored. After analyzing the input program P, MoCA reports assert

TABLE I: Comparative Results on litmus tests

Test	MoCA			CDSChecker			GenMC			HMC		
	M	N	Time	M	N	Time	M	N	Time	M	N	Time
WRC+addrs(7)	7	0	0.03s	7	1	0.01s	7	1	0.02s	7	1	0.03s
WR-ctrl(4)	7	0	0.03s	4	2	0.01s	4	2	0.02s	4	2	0.02s
Z6+poxxs(4)	18	0	0.12s	14	4	0.01s	4	4	0.03s	4	4	0.03s
IRIW+addrs(15)	15	0	0.07s	15	1	0.01s	15	1	0.02s	15	1	0.02s
WW+RR(15)	96	0	0.53s	15	66	0.02s	15	66	0.02s	15	66	0.02s

M: #*MCA* sequences, N: #non-*MCA* sequences

TABLE II: Comparative Results on benchmarks

Test	MoCA		CDSChecker		GenMC		HMC	
	#Seq	Time	#Seq	Time	#Seq	Time	#Seq	Time
mutex	5	0.02s	2-NVs	0.01s	NV	0.03s	NV	0.02s
peterson	13	0.15s	666-NVs	2.73s	NV	0.02s	NV	0.02s
RW-lock	246	0.52s	193-NVs	0.38s	NV	0.02s	NV	0.04s
spinlock	506	16.98s	TO	-	NV	0.08s	NV	0.15s
fibonacci-2	667	5.57s	TO	-	NV	0.04s	NV	0.03s
fibonacci-3	10628	2m14s	TO	-	NV	0.06s	NV	0.07s
fibonacci-4	92421	56m21s	TO	-	NV	0.13s	NV	0.31s
counter-5	3599	39.78s	25-NVs	0.31s	NV	0.06s	NV	0.03s
counter-10	55927	12m53s	100-NVs	9.21s	NV	0.05s	NV	0.07s
counter-15	TO	-	225-NVs	50.31s	NV	0.11s	NV	0.16s
flipper-5	2489	20.19s	201-NVs	3.26s	NV	0.03s	NV	0.04s
flipper-10	96737	6m12s	TO	-	NV	0.04s	NV	0.02s
flipper-15	TO	-	TO	-	NV	0.03s	NV	0.03s
prod-cons-10	9373	1m23s	TO	-	NV	0.04s	NV	0.04s
prod-cons-15	38593	6m46s	TO	-	NV	0.02s	NV	0.02s
prod-cons-20	109838	20m28s	TO	-	NV	0.02s	NV	0.02s

x-NVs: 'x' number of non-*MCA* violations (if #violations reported)
NV: non-*MCA* violation (if analysis halts at first violation, #NVs not known)
TO: timeout (60m)
Additionally, MoCA detected na races in tests mutex *and* counter.

violations if any. According to *C11* standard if the order of occurrence of a pair of na ordered events can potentially be reversed in a trace, then the behavior of the trace is *undefined*. Such a behavior can defy the coherence specification (shco), (co) and produce invalid values. Thus, MoCA also reports *data races on non-atomic memory accesses*.

Experiment details. We performed tests to validate correctness *wrt MCA* using diy7 family of litmus tests [18] (sample listed in Table III). To test *C11* coherence, we synthesized 56 litmus tests relevant to the *C11* coherence rules (eg. row 1-4, Table IV) and borrowed multi-threaded benchmarks from the SV-Comp benchmark suite [19] (eg. row 5-8, Table IV). We remodeled them for *C11* with the use of atomic data types and *C11* memory orders. We recorded time (column 'Time') and the number of maximal sequences explored (column '#Seq'), which includes *at least* one execution corresponding to each trace and (possibly) few redundant executions owing to the non-optimal nature of the underlying source-DPOR algorithm. Further, if a test contains na races (column 'race?') then we report the number of maximal sequences that contain na race(s) (column '#Rseq').

To demonstrate the effectiveness of MoCA, we used litmus tests and benchmarks from the SV-Comp suite that produce a *strict subset* of *C11* behaviors when restricted to *MCA*. We compared the outcome of such tests on MoCA-tool with state-of-the-art stateless model checking tools for *C11* (and its variants) namely CDSChecker [10] and GenMC [20]; and hardware model checker (HMC) [13].

Results' analysis. Table I shows the results of comparative study on litmus tests that would show additional behaviors

TABLE III: *MCA* tests

Test	#Seq	Time
CoRR(3)	3	0.02s
CO-RSDWI(3)	6	0.02s
R+fn+fn(2)	5	0.02s
RSDWI(6)	22	0.11s
WRR+2W(12)	29	0.12s
Luc17(12)	12	0.07s
Luc10(3)	MV	0.02s
S-popl(3)	MV	0.01s

MV: *MCA* violation(s) detected

TABLE IV: *C11* tests

Test	#Seq	Time	race?	#Rseq
simple-sw(3)	3	0.006s	Y	2
simple-ithb(4)	4	0.034s	Y	2
RS-blk(10)	MV	0.07s	Y	8
CSE-no-blk(8)	12	0.158s	N	-
no-fence-sync(5)	MV	0.054s	Y	5
fib-no-assert	26	0.14	N	-
fmax-cas	31	0.21s	N	-
flipper	1628	9.29s	N	-

MV: *MCA* violation(s) detected
race?: Does the test have a race on na events?
#Rseq: number of na races detected by MoCA

on non-*MCA* model. The table contains small tests with 15 or less traces. The number of valid *C11* traces under *MCA* have been shown in bracket accompanying the name of the test. For instance, 'WRC+addrs(7)', shows that the test 'WRC+addrs' has 7 valid *MCA C11* traces . These numbers have been manually computed. We have reported the number of *MCA* sequences (column 'M') and the number of non-*MCA* sequences (column 'N') for each of the techniques MoCA, CDSChecker, GenMC and HMC, along with the time taken by the techniques for performing their analysis.

For larger benchmarks we have used assert statements to catch non-*MCA* sequences and used 'NV' to indicate assert violation(s) in non-*MCA* sequences. The results are shown in Table II. An 'NV' result implies that the benchmark may have a legitimate violaton under *C11* model but not under *MCA*.

CDSChecker reports all sequences explored including ones with assert violation, thus, for CDSChecker the collected number of non-*MCA* violations have been reported as 'x-NVs', indicating 'x' number of assert violations in non-*MCA* sequences. GenMC and HMC halt at the first detection of violation, thus, no such information is available for reporting. Hence, for GenMC and HMC we have simply written 'NV'. As a consequence, the time reported for GenMC and HMC is the time to encounter the first assert violation and is therefore much lower than the time reported by CDSChecker and MoCA. Due to such difference in tool design, the reported time of analysis is incomparable and has been reported only for reference. The value 'TO' indicates timeout set for 60 minutes.

Finally, we re-emphasize that the techniques CDSChecker and GenMC are designed for *C11* (or its variants) and HMC is for a collection of hardware models subsuming *MCA*. Naturally, these techniques explore a larger set of traces, and the non-*MCA* violation(s) reported by them are indeed true violations under their respective models. However, some of the violations reported by them may never manifest on the underlying architecture. We can observe from Table I and Table II that benchmarks can produce hundreds of assert violations that may not be reproducible on an actual architecture. Thus, a precise technique for *MCA* such as MoCA can be useful.

VIII. Related Work

Stateless model checking: Stateless model checking (SMC) with DPOR [5] has been used for (SC) [4][21][22] and weak memory models (WMM) TSO, PSO [6][7], Power [23] and *C11* [10]. The techniques [17][20][24] have proposed SMC for variants of *C11* and [13] for a superset of architectural

memory models (including *MCA* model).

Symoblic Analyses: Symbolic and predictive trace analysis is investigated in [25][26][27] and has been applied to the verification of MPI programs [28]. Static analysis using thread modular analysis or abstract interpretation have been proposed under SC [29][30] and WMM [31][32]. While these techniques are sound, they may suffer from false alarms. Recent works have also investigated bounded model checking under loop and *view* bounds to analyze WMM [3][33].

IX. Concluding Remarks

We present MoCA, a dynamic verifier to analyze *C11* program traces valid under *MCA* model for assertion violations na data races. The technique is shown to be sound and precise. The empirical results demonstrate the utility of MoCA over existing techniques for *C11*.

Future Work: In future we would like to explore the extensions of our work to reactive systems and include richer program constructs such as locks and memory barriers. Another area of possible investigation would be to combine current work with symbolic trace verification so as to avoid re-runs of the input program.

Acknowledgment

This work is partially supported by the Department of Science and Technology under the grant number DST ECR/2017/003427.

References

[1] ISO/IEC, "Programming languages – c. international standard," www.open-std.org/jtc1/sc22/wg14/www/docs/n1548.pdf, ISO/IEC, 2011.

[2] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber, "Mathematizing c++ concurrency," *ACM SIGPLAN Notices*, vol. 46, no. 1, pp. 55–66, 2011.

[3] P. A. Abdulla, J. Arora, M. F. Atig, and S. Krishna, "Verification of programs under the release-acquire semantics," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 1117–1132.

[4] P. Abdulla, S. Aronis, B. Jonsson, and K. Sagonas, "Optimal dynamic partial order reduction," *ACM SIGPLAN Notices*, vol. 49, no. 1, pp. 373–384, 2014.

[5] C. Flanagan and P. Godefroid, "Dynamic partial-order reduction for model checking software," *ACM Sigplan Notices*, vol. 40, no. 1, pp. 110–121, 2005.

[6] P. A. Abdulla, S. Aronis, M. F. Atig, B. Jonsson, C. Leonardsson, and K. Sagonas, "Stateless model checking for tso and pso," in *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 9035*, 2015.

[7] N. Zhang, M. Kusano, and C. Wang, "Dynamic partial order reduction for relaxed memory models," in *ACM SIGPLAN Notices*, vol. 50, no. 6. ACM, 2015, pp. 250–259.

[8] A. Ltd., "Arm architecture reference manual (armv8, for armv8-a architecture profile)," https://developer.arm.com/documentation/ddi0487/aa, ARM Ltd., 2017.

[9] C. Pulte, S. Flur, W. Deacon, J. French, S. Sarkar, and P. Sewell, "Simplifying arm concurrency: multicopy-atomic axiomatic and operational models for armv8," *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, pp. 1–29, 2017.

[10] B. Norris and B. Demsky, "Cdschecker: checking concurrent data structures written with c/c++ atomics," in *ACM SIGPLAN Notices*, vol. 48, no. 10. ACM, 2013, pp. 131–150.

[11] M. Kokologiannakis, A. Raad, and V. Vafeiadis, "Model checking for weakly consistent libraries," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 96–110.

[12] A. Podkopaev, O. Lahav, and V. Vafeiadis, "Bridging the gap between programming languages and hardware weak memory models," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–31, 2019.

[13] M. Kokologiannakis and V. Vafeiadis, "Hmc: Model checking for hardware memory models," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 1157–1171.

[14] R. J. Colvin and G. Smith, "A wide-spectrum language for verification of programs on weak memory models," in *International Symposium on Formal Methods*. Springer, 2018, pp. 240–257.

[15] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. Cambridge, MA, USA: MIT Press, 2000.

[16] S. Singh, D. Sharma, and S. Sharma, "Dynamic verification of c/c++ 11 concurrency over multi copy atomics," *arXiv preprint arXiv:2103.01553*, 2021.

[17] M. Kokologiannakis, O. Lahav, K. Sagonas, and V. Vafeiadis, "Effective stateless model checking for c/c++ concurrency," *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, p. 17, 2017.

[18] A. Jade and M. Luc, "diy7 tool suite," http://diy.inria.fr/doc/index.html, 2017, accessed: 2020-02-12.

[19] "Competition on software verification," https://sv-comp.sosy-lab.org/2018/benchmarks.php, SV-COMP, 2018, accessed: 2019-10-04.

[20] M. Kokologiannakis, A. Raad, and V. Vafeiadis, "Model checking for weakly consistent libraries," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 96–110.

[21] M. Chalupa, K. Chatterjee, A. Pavlogiannis, N. Sinha, and K. Vaidya, "Data-centric dynamic partial order reduction," *Proc. ACM Program. Lang.*, vol. 2, no. POPL, pp. 31:1–31:30, 2017.

[22] C. Rodríguez, M. Sousa, S. Sharma, and D. Kroening, "Unfolding-based partial order reduction," in *CONCUR*, 2015.

[23] P. A. Abdulla, M. F. Atig, B. Jonsson, and C. Leonardsson, "Stateless model checking for power," in *International Conference on Computer Aided Verification*. Springer, 2016, pp. 134–156.

[24] P. A. Abdulla, M. F. Atig, B. Jonsson, and T. P. Ngo, "Optimal stateless model checking under the release-acquire semantics," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, p. 135, 2018.

[25] V. Forejt, D. Kroening, G. Narayanaswamy, and S. Sharma, "Precise predictive analysis for discovering communication deadlocks in mpi programs," in *International Symposium on Formal Methods*. Springer, 2014, pp. 263–278.

[26] A. Gupta, T. A. Henzinger, A. Radhakrishna, R. Samanta, and T. Tarrach, "Succinct representation of concurrent trace sets," in *ACM SIGPLAN Notices*, vol. 50, no. 1. ACM, 2015, pp. 433–444.

[27] C. Wang, S. Kundu, M. K. Ganai, and A. Gupta, "Symbolic predictive analysis for concurrent programs," in *FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings*, 2009, pp. 256–272.

[28] D. Khanna, S. Sharma, C. Rodríguez, and R. Purandare, "Dynamic symbolic verification of MPI programs," in *Formal Methods - 22nd International Symposium, FM 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 15-17, 2018, Proceedings*, 2018, pp. 466–484.

[29] C. Flanagan, S. N. Freund, and S. Qadeer, "Thread-modular verification for shared-memory programs," in *European Symposium on Programming*. Springer, 2002, pp. 262–277.

[30] M. Kusano and C. Wang, "Flow-sensitive composition of thread-modular abstract interpretation," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 799–809.

[31] M. Kusano and C.Wang, "Thread-modular static analysis for relaxed memory models," in *Proceedings of 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017, 2017.

[32] T. Suzanne and A. Miné, "Relational thread-modular abstract interpretation under relaxed memory models," in *Asian Symposium on Programming Languages and Systems*. Springer, 2018, pp. 109–128.

[33] H. Ponce-de León, F. Furbach, K. Heljanko, and R. Meyer, "Dartagnan: Bounded model checking for weak memory models (competition contribution)," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2020, pp. 378–382.