Model Checking C/C++ with Mixed-Size Accesses

IASON MARMANIS, MPI-SWS, Germany
MICHALIS KOKOLOGIANNAKIS, ETH Zurich, Switzerland
VIKTOR VAFEIADIS, MPI-SWS, Germany

State-of-the-art model checkers employing dynamic partial order reduction (DPOR) can verify concurrent programs under a wide range of memory models such as sequential consistency (SC), total store order (TSO), release-acquire (RA), and the repaired C11 memory model (RC11) in an optimal and memory-efficient fashion. Unfortunately, these DPOR techniques cannot be applied in an optimal fashion to programs with *mixed-sized accesses* (MSA), where atomic instructions access different (sets of) bytes belonging to the same word. Such patterns naturally arise in real life code with C/C++ union types, and are even used in a concurrent setting.

In this paper, we introduce Mixer, an optimal DPOR algorithm for MSA programs that allows (multi-byte) reads to be revisited by multiple writes together. We have implemented Mixer in the GenMC model checker, enabling (for the first time) the automatic verification of C/C++ code with mixed-size accesses. Our results also extend to the more general case of *transactional* programs provided that the set of read accesses performed by a transaction can be dynamically overapproximated at the beginning of the transaction.

CCS Concepts: • **Theory of computation → Concurrency**; **Verification by model checking**.

Additional Key Words and Phrases: Model Checking, Dynamic Partial Order Reduction, Mixed-Sized Accesses

ACM Reference Format:
Iason Marmanis, Michalis Kokologiannakis, and Viktor Vafeiadis. 2025. Model Checking C/C++ with Mixed-Size Accesses. *Proc. ACM Program. Lang.* 9, POPL, Article 75 (January 2025), 21 pages. https://doi.org/10.1145/3704911

1 Introduction

Stateless model checking with *dynamic partial order reduction* (DPOR) is a state-of-the-art technique for verifying safety properties of concurrent programs that are guaranteed to terminate. Given a concurrent program, a memory consistency model (which determines the semantics of concurrent memory accesses) and a suitable equivalence over program executions, DPOR generates all consistent program executions modulo the equivalence relation, and checks that all these executions satisfy the given safety specification. An optimal DPOR algorithm never generates two equivalent executions nor wastes time on partial explorations that do not lead to a new full program execution. A space-efficient (a.k.a. truly stateless) DPOR algorithm does so using memory proportional to the size of a single program execution.

While under *sequential consistency* (SC), program executions can be represented as traces of individual memory accesses, for weak memory models, such as TSO [Owens et al. 2009], release-acquire (RA) [Lahav et al. 2016], and RC11 [Lahav et al. 2017], they are best represented as *execution graphs*—a generalization of traces where the individual memory accesses are related by a bunch of partial orders.

Authors' Contact Information: Iason Marmanis, MPI-SWS, Kaiserslautern, Germany, imarmanis@mpi-sws.org; Michalis Kokologiannakis, ETH Zurich, Switzerland, michalis.kokologiannakis@inf.ethz.ch; Viktor Vafeiadis, MPI-SWS, Kaiserslautern, Germany, viktor@mpi-sws.org.

ACM 2475-1421/2025/1-ART75
https://doi.org/10.1145/3704911

In this paper, we will consider the most common equivalence notion on execution graphs: *Shasha-Snir equivalence* [Shasha and Snir 1988], which is a generalization of Mazurkiewicz trace equivalence [Mazurkiewicz 1987], and relates two execution graphs if they agree on the set of memory accesses and on the relative order of non-commuting accesses (e.g., a read and a write to the same location).

State-of-the-art DPOR algorithms, such as TruSt [Kokologiannakis et al. 2022], achieve both optimality and space-efficiency, for a wide range of memory consistency models, by building execution graphs incrementally and enforcing consistency at each point, and revisiting existing read events in a graph whenever a new write event is added. To achieve this, TruSt requires that the memory model be *extensible* [Kokologiannakis et al. 2019b]: namely, there exists a way to extend a consistent execution with a memory access and preserve consistency.

Although memory consistency models readily satisfy extensibility, this is not the case for models that support any kind of transactional semantics, where multiple accesses/operations are executed in an atomic fashion (see §2).

For verifying a transactional program P under a (non-extensible) consistency model M, prior work [Bouajjani et al. 2023] therefore uses a suboptimal strategy: it generates all consistent executions of P under a weaker model that satisfies extensibility, and simply suppresses the reporting of errors on executions that are inconsistent according to M. As we discuss in §4.5, such approaches can scale very poorly because P may have many more inconsistent executions than consistent ones.

In this paper, we focus on *mixed-sized accesses* (MSA), a specific class of transactions that occurs in low-level systems code. These are instructions that atomically access different sets of bytes that belong to the same word (e.g., an 8-bit store and 32-bit load at the same address). They arise from a familiar C/C++ coding pattern where a union is accessed through its different members.

Example 1.1 A typical use of MSA appears in the `lockref` [Corbet 2013] data structure in the Linux Kernel, which manipulates a shared reference count efficiently by reducing contention on the associated lock. The core data structure is shown below.

```
struct lockref {
  union {
    u64 lock_count;
    struct {
      u32 lock;
      u32 count;
    };
  };
};
```

```
void lockref_get(struct lockref *lockref) {
  struct lockref old;
  old.lock_count = READ_ONCE(lockref->lock_count);
  while (old.lock == 0) {
    struct lockref new = old;
    new.count++;
    if (CAS(&lockref->lock_count, &old.lock_count, new.lock_count))
      return;
  }
  spin_lock(&lockref->lock)
  lockref->count++;
  spin_unlock(&lockref->lock)
}
```

When attempting to increase the reference count (lockref_get), the whole structure is first read in one 8-byte instruction (lock_count access). If the lock is not held, a CAS (compare-and-swap) operation attempts to atomically increment the reference count, without taking the lock. Since the CAS can fail, this attempt is repeated as long as the lock is not taken. If the lock gets taken, the operation falls back to obtaining the lock (4-byte access to lock), incrementing the counter (two 4-byte accesses to count), and releasing the lock (4-byte access to lock).

In §2 , we observe that even though MSA-compatible consistency models [Alglave et al. 2021; Flur et al. 2017] are not extensible at the level of individual memory accesses, they are extensible at the level of *whole transactions* (MSA instructions). In other words, a consistent program execution containing only fully executed MSA instructions can always be extended with an arbitrary instruction in a consistent way.

In §3 and §4, we thus devise an optimal DPOR algorithm, called Mixer, that handles MSA programs by changing the granularity at which revisits happen—from individual (byte-sized) memory accesses to entire transactions. To do so, we introduce the novel notion of *multi-write* revisits, to support the case where a (multi-byte) read instruction reads from multiple write instructions, one or more of which may have been added to the execution graph after the read.

Contrary, however, to DPOR algorithms like TruSt, Mixer fails to validate a stronger notion of optimality [Bouajjani et al. 2023; Kokologiannakis et al. 2022] where the algorithm avoids any wasteful exploration. Nevertheless, we provide a bound on the depth of such wasteful explorations (i.e., the depth of the exploration tree), which is constant for MSA programs.

In §5, we note that apart from the bound on wasteful explorations, our remaining results extend to *transactional* programs, as long as the individual memory accesses performed by a transaction can be overapproximated at the beginning of the transaction. At the expense of slowing down the verification algorithm, this condition can be trivially fulfilled by saying that a transaction can access every memory location.

In §6, we implement our algorithm in the context of the GenMC model checker [Kokologiannakis and Vafeiadis 2021]. In §7, we evaluate Mixer on challenging benchmarks involving mixed-size accesses, showing that our algorithm is substantially more efficient than applying TruSt and disregarding the atomicity semantics of MSA instructions until the end of the exploration, where the inconsistent ones are filtered out.

2 Background

In this section, we will provide a high-level overview of how DPOR algorithms work, discuss the underlying correctness assumptions, and show how MSA programs invalidate these assumptions.

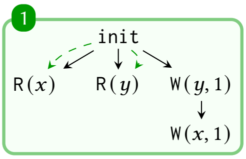
2.1 Execution Graphs

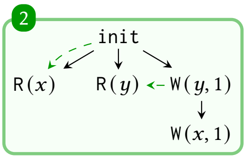
DPOR algorithms systematically explore all the non-equivalent behaviors of a concurrent program, without storing the set of explored behaviors. Instead of exploring the behaviors in the form of traces (modulo the equivalence-relation), many recent DPOR algorithms [Kokologiannakis et al. 2017, 2022, 2019b] explore *execution graphs*, a structure that only partially orders the program's instructions and captures an equivalence class of traces.

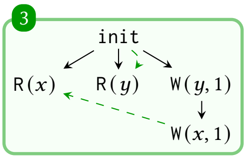
An execution graph comprises a set of nodes E (also called *events*), including the set of read events R and write events W, and a few relations on these nodes, including the *program order* po, the *reads-from order* rf, and the *coherence order* co. Each program instruction corresponds to a node in the graph, and the relations capture ordering requirements among them. Concretely, po captures the intra-thread order of execution, rf captures the data flow from write events to read events, and co totally orders the writes to the same location.

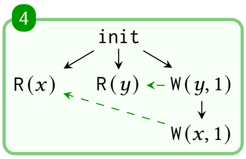
Example 2.1 For example, the program below gives rise to four execution graphs, which correspond to the different possible combinations of values that the first two threads can read.

$$\begin{array}{l||l||l} \text{T1: } a := x & \text{T2: } b := y & \text{T3: } y := 1 \\ & & \phantom{\text{T3: }} x := 1 \end{array} \qquad \text{(R+R+WW)}$$









As one can see, the execution graphs above only differ in their rf component, i.e., in which writes the reads are reading from.

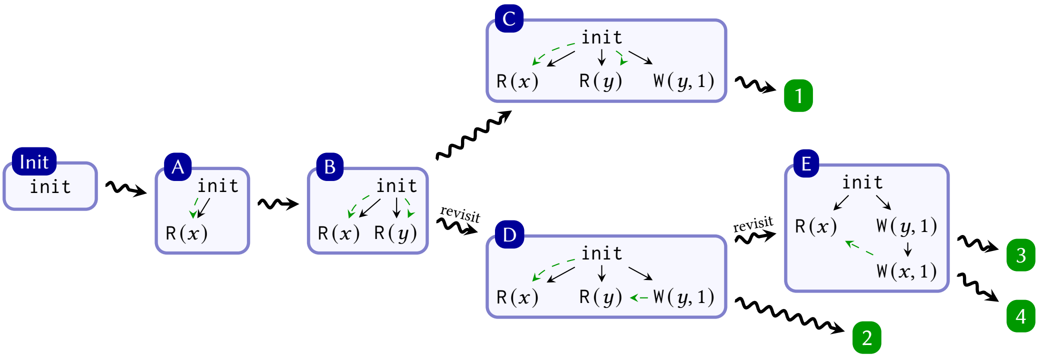
Naturally, not all execution graphs make sense: they have to correspond to the program and also satisfy the constraints imposed by the given *memory consistency model*. The strongest such model is *sequential consistency* (SC), which requires the relation $\mathsf{po} \cup \mathsf{rf} \cup \mathsf{co} \cup \mathsf{rb}$ to be acyclic, where $\mathsf{rb} \triangleq \mathsf{rf}^{-1} ; \mathsf{co}$ is the *reads-before* relation that orders each read event before its subsequent writes.

2.2 DPOR: Basics and Assumptions

DPOR algorithms explore all possible consistent execution graphs by following a fixed schedule among the program's threads and building an execution graph incrementally. For simplicity, we assume a left-to-right order on the threads. As the graph is constructed, independent events (i.e., accesses to different locations or two read events) remain unordered, but conflicting events are ordered on the fly by co, rf, or rb.

Concretely, when a read event is added, multiple subexplorations are initiated, each exploring a different consistent read-from option (rf) for the read. More interestingly, when a write event is added, besides exploring its possible placements in the co order, DPOR also explores the scenario where an existing read reads from the new write. Such race reversal scenarios are also referred to as *revisiting*: the later write *revisits* the read. During a revisit, all events that are added after the read but are not *causally-before* the write are removed from the execution, and the read is made to read from the write. In the context of an execution graph, an event is causally-before another one if there is a path of po and rf edges from the first to the second event.

Example 2.1 (Cont.). Consider how a DPOR algorithm would reach the execution 4 of R+R+WW, where both reads read 1, assuming a left-to-right scheduling. DPOR would first need to reach one of C or D to then add the $\mathtt{W}(x, 1)$ and revisit the T1 read, leading to execution E, before finally reaching 4.



Observe that both executions are an extension of the execution that only has the write access to y (the execution E without the revisited read and the revisiting write). It is reasonable to assume that this execution is consistent, since it is a "sub-execution" of the consistent execution 4. Without additional assumptions about the memory consistency model, there is, however, no reason to assume that C or D are consistent, which is necessary for DPOR to reach 4.

DPOR algorithms solve this by requiring an *extensibility* property: given a consistent execution and an event that corresponds to the next instruction picked by the scheduler, there is a way to add the event to the execution and preserve consistency.

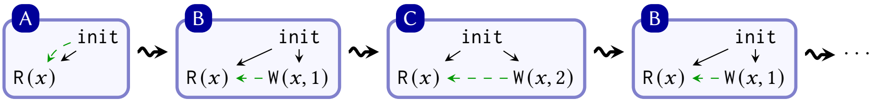
Memory models readily satisfy this assumption by guaranteeing that adding the event *maximally* in the coherence order (i.e., writes are added at the end of co, and reads read from co-latest writes) preserves consistency. This corresponds to the reasonable expectation that the operational semantics of the memory model do not get "stuck".

Algorithms like TruSt leverage this insight, and among all the possible extensions pick the maximal one as the only extension where the revisit is performed: the revisit is only allowed from the execution C, and not from D. Alternative approaches avoid the duplication by instead storing part of the execution, leading, however, to possible memory consumption blow-up [Abdulla et al. 2014; Kokologiannakis et al. 2019b].

Picking the maximal extension as the one to allow the revisit is not required for the algorithm's correctness, as long as the allowed extension is consistent and unique. There is, however, one general condition that it should obey: among the events affected by the revisit (the events removed from the execution and the read that is being revisited), a read cannot read from a write that was added later. This is to ensure that there is no infinite repetition of revisits that would make the algorithm diverge. Both GenMC [Kokologiannakis et al. 2019b] and TruSt enforce an equivalent condition, which guarantees this invariant.

Example 2.2 As an example, consider the program below. Without enforcing this restriction, the two writes can initiate an infinite sequence of revisits to the read event: each write revisits the read, before being deleted during a revisit by the other write.

$$\text{T1: } a := x \;\Big\|\; \text{T2: } x := 1 \;\Big\|\; \text{T3: } x := 2 \qquad\qquad (\text{R+W+W})$$



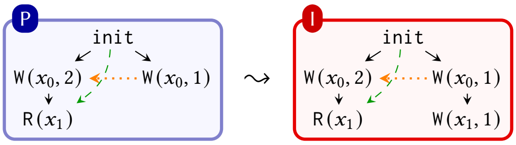
2.3 Mixed-Sized Accesses and Extensibility

When trying to extend DPOR to support programs with mixed-sized accesses, a seemingly easy approach would be to break down accesses to their byte-level constituent accesses and apply DPOR as before. However, this solution is flawed because the resulting memory model is no longer extensible: after consistently executing part of the byte-level accesses that correspond to a larger access, there is no guarantee that the remaining byte-level accesses can be consistently executed as well.

To illustrate, we fix for now our memory model to Sequential Consistency: the consistent behaviors can be explained by a linearization of the byte-level accesses, such that accesses corresponding to the same instruction are adjacent in the linearization order.

Example 2.3 As a first illustrative example of how extensibility can fail to hold, consider the program below, where the shared location x contains (at least) two bytes; T1 writes to byte 1 and then reads byte 0, while T2 performs one 16-bit store, writing 1 to both bytes comprising x. We use a range of numbers as subscripts to specify which bytes of a word an instruction accesses, and similarly use this notation in the execution graphs.

$$\begin{array}{l|l} \text{T1: } x_0 := 2 & \text{T2: } x_{0:1} := \texttt{0x101} \\ \quad a := x_1 & \end{array}$$

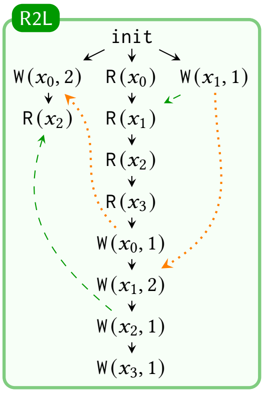


The partial execution P, where the write of T2 is co-ordered before the write of T1, is consistent but cannot be extended in a consistent way: the execution I is inconsistent. To see this, observe that the co ordering forces the instruction of T2 to be ordered before the read of T1, and therefore if the write to the second byte (x_1) is also present, it must be ordered before the read of T1, i.e., it must be read.

Let us now see how this issue can actually arise in practice and crucially affect DPOR's exploration. This is easier understood in the presence of read-modify-write (RMW) instructions, where the read and write accesses happen atomically (i.e., the accesses are adjacent in the linearization order).

Example 2.4 Consider the program below, and assume a DPOR run with a left-to-right scheduling. An algorithm that follows the principle of not removing from the execution a write that revisited cannot obtain the consistent execution R2L on the right, where the threads are executed from right to left, irrespective of which revisits it allows.

$$\begin{array}{l|l|l} \text{T1: } x_0 := 2 & \text{T2: } \texttt{fetch\_add}(x_{0:3}, 0x1010101) & \text{T3: } x_1 := 1 \\ \quad a := x_2 & & \end{array}$$



After adding the events of T1, T2 is scheduled. In order to eventually have byte x_0 read 0, it must initially read 0 as well, since it cannot be removed later. Similar to before, this forces an ordering between T2's RMW and the read to x_2 of T1, and therefore the later write to x_2 cannot be added. On the other hand, it also cannot revisit the read to x_2 since this would effectively block T2's read to x_1 to be revisited. Thus, the execution R2L will not be explored.

2.4 A Simple Workaround

A simple way to bypass the lack of extensibility is to invoke DPOR with a weaker consistency model that satisfies the extensibility property, and only check for full consistency before reporting an error. The simplest way to obtain a weaker consistency model for exploration purposes is to disregard the parts of the consistency model enforcing atomicity of multi-byte accesses and treat each byte access as completely independent from the other byte accesses of the same instruction.

While this approach is correct, it does explore many inconsistent executions wasting time and resources: a program can have *unboundedly* more executions that are consistent under the weaker that the ones that are consistent according to the stronger one. In §4.5 we discuss a stronger notion of optimality that precludes such solutions.

3 Mixer: Overview of our Solution

In this section, we introduce a direct way to support programs with mixed-sized accesses. We present the key ideas of Mixer in an informal fashion, and relegate the formal definitions to §4.

3.1 Performing DPOR at the Instruction Level

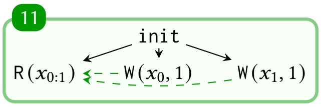
The key observation underlying Mixer is that transactional consistency models—and MSA-compatible memory models, in particular—are extensible at the level of entire transactions (MSA instructions), not at the level of individual memory accesses. That is, given a consistent execution containing no events due to partially executed transactions, we can extend it in a consistent way by adding all the events corresponding to an additional transaction. In particular, we can add its events so that every read access reads from the last, in the coherence order, same-location write access, and every write access is placed at the end of the coherence order.

Mixer leverages this observation and picks at each point one instruction and adds, in one step, every access that corresponds to the instruction, exploring all possible rf and co placements for each of the constituent accesses. (The read and write parts of an RMW can be split, for simplicity, but the scheduler must always follow the read part with the corresponding write part.) Additionally, —and more importantly, since this could so far be simulated in the original DPOR scheme— a whole instruction revisits another one: this guarantees that no execution is generated where some, but not all, the accesses of an instruction are present, satisfying our extensibility constraint.

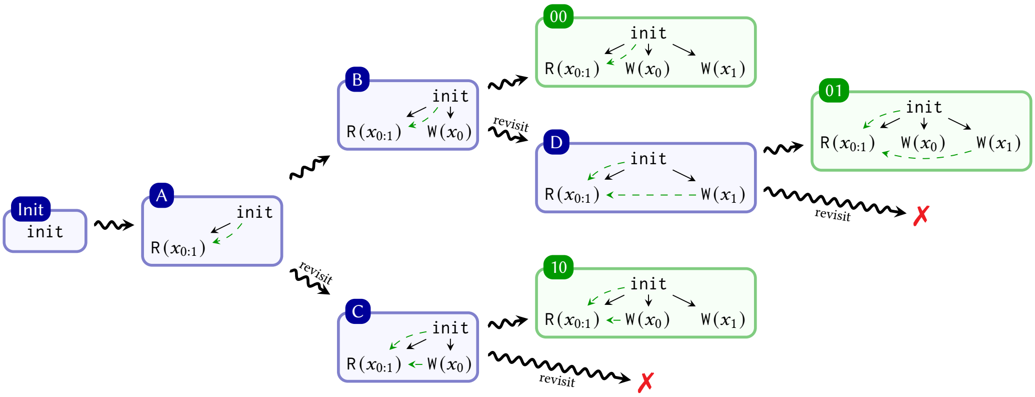
As the following example demonstrates, this instruction-level treatment, however, is not enough.

Example 3.1 Consider the program below, and focus on how we could obtain the execution 11 where both accesses of the read instruction read 1.

T1: $a := x_{0:1}$ || T2: $x_0 := 1$ || T3: $x_1 := 1$



After adding T1 (A), DPOR can either simply execute T2 (B) or execute it and revisit the read of T1 (C). In either case, the write of T3 should also revisit the read. In the first case, this would lead to execution D, where DPOR can only add T2 without a revisit (because revisiting would delete T3). In the second case (C), the revisit from T3 is similarly not allowed because it would delete T2.



The desired execution 11 is therefore missed. We need to allow one of the revisits from either C or D.

3.2 Multi-Write Revisits

We resolve this problem by introducing the concept of a *multi-write revisit*, where multiple instructions together revisit a previous instruction. Mixer implements this idea by allowing an (optional) set of instructions, which we refer to as a *keep set*, that would be normally deleted during a revisit, to remain in the execution and to perform the revisit together with the write accesses of the newly added instruction.

Concretely, in the execution C and D, we consider both the case where the missing write tries to revisit the read, and fails since the other write cannot be removed, as well as the case where the other write is kept, allowing the read to read from both writes.

Multi-write revisits are sufficient to ensure *completeness* of the model checking algorithm, namely that it explores all consistent program executions. Unless suitably restricted, however, the extra choice that DPOR can make can easily lead to duplication. To guarantee that no exploration is explored twice, we therefore need to impose a set of conditions on the keep sets to consider them valid.

Tie-breaking. First, we need to address the problem arising in the example that we have just seen: either C and D should perform a revisit while keeping the other write, but not both. Otherwise 11 would be explored twice.

For this, we perform a tie-break on the write instruction that performs the final revisit: we pick the rightmost one, and enforce that the scheduler picks threads in a left-to-right fashion. This scheduling constraint is crucial for correctness: when the predetermined write that should perform the revisit is added, the rest of the writes that need to perform the revisit together must already be in the execution.

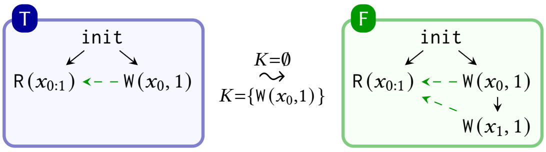
Therefore, we enforce that keep sets can now only include instructions whose thread is to the left of the instruction's thread that performs the revisit.

For example, in the execution D, T2 is not allowed to revisit by keeping T3, since the kept write would be to the write of the new write. This way, execution 11 is reached exactly once, from the execution C.

Independence. Another constraint on the instructions of the keep set is that they are *independent*, i.e., the are not causally related (in a po or rf sense) to each other, or to the newly added instruction. This is to avoid having the exactly same revisit being performed with a different keep set.

Example 3.2 As an example, consider the program below. After reaching the execution T where the read reads from the write to x_0, and the write to x_1 is about to be added, revisiting with either an empty keep set, or with keeping the write to x_0, would result in the same execution F.

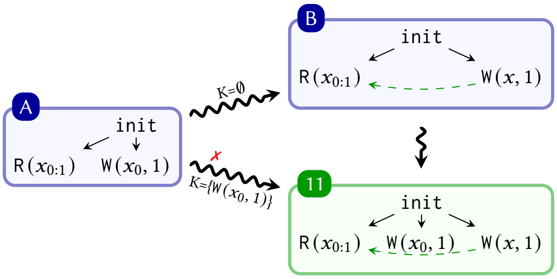
T1: $a := x_{0:1}$ || T2: $x_0 := 1$
$x_1 := 1$



Relevance. All of the instructions in the keep set during a revisit must be *relevant* after the revisit: at least one byte-level access must be read by the revisited read instruction. This ensures that the revisit with the specific keep set was necessary, and the resulting execution could not be recovered in another way.

Example 3.3 As an example, consider the program below and focus on the incomplete execution A where the instruction of T3 is not yet added.

T1: $a := x_{0:1}$ || T2: $x_0 := 1$ || T3: $x_1 := 1$



When the write of T3 is added and revisits are considered, keeping the write of T2 without reading from it should not be considered. Mixer will recover execution 11 from the execution B obtained from revisiting A with an empty keep set, by adding the now missing write instruction of T2 thread again.

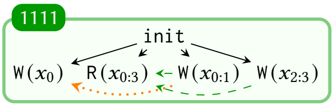
3.3 Mixer: Reconsidering Unconstrained Reads

Apart from the multi-write revisits, there is one last change necessary to recover a complete and optimal DPOR algorithm, which again pertains to the revisit mechanism.

Specifically, when revisiting a read instruction, relevance forces (some of) the read accesses to read (parts of) the kept write instructions and the revisiting write instruction. However, it can be that parts of the read instruction are left unconstrained. These unconstrained accesses must be *reconsidered*: every possible read-from option is considered again, allowing them to read from a different write instruction than before the revisit.

Example 3.4 To see why this is necessary, consider the following program and the depicted target execution 1111 where every byte of the read instruction reads 1, and the access of T1 to x_0 is co after the one of T3.

T1: x_0 := 2 $\|$ T2: a := $x_{0:3}$ $\|$ T3: $x_{0:1}$:= 0x101 $\|$ T4: $x_{2:3}$:= 0x101



To reach this execution, the instruction of T4 should be eventually added and revisit the read instruction, while keeping the instruction of T3.

Notice, however, that this cannot happen from an execution where the read to x_0 reads 1: the read instruction is not reading maximal (and, in principle, there is no reason why such an execution would be consistent). To reach the desired execution, our algorithm will reach the execution where it reads 2 for x_0, and during the revisit from T4 (keeping T3) the option of reading 1 instead will also be explored.

4 Mixer: Formal Definition, Correctness, and Optimality

In this section, (i) we define how executions are represented as executions graphs, (ii) we present our assumptions on the underlying memory model, (iii) we present Mixer, our DPOR algorithm for MSA programs, (iv) we explain Mixer's correctness arguments, and (v) we discuss its standing w.r.t. stronger notions of optimality.

4.1 Execution Graphs

An execution graph comprises a set of events (nodes), and a few relations on these events (edges).

We assume that each location comprises B bytes and denote as Byte the set of natural numbers from 0 to $B - 1$. We refer as *footprint* to the set of bytes accessed by a memory instruction, and assume a valid set F of footprints. Usually, footprints are contiguous, but our formalism does not depend on this assumption.

Definition 4.1. An *event*, $e \in$ Event, is either the initialization event init, or a thread event $\langle t, i, f, lab\rangle$ where $t \in$ Tid is a thread identifier, $i \in$ Idx is a serial number (denoting the index of an event within a thread), $f \in$ F is the footprint of the corresponding access, if applicable, and $lab \in$ Lab is a label that takes (at least) one of the following forms:

- Write label: $\mathtt{W}^k(l, v) \in \mathtt{W}$, where k records the write attributes, $l \in$ Loc the location accessed, and $v \in$ Val the value written.
- Read label: $\mathtt{R}^k(l) \in \mathtt{R}$, where k records the read attributes and $l \in$ Loc the location accessed.

Read and write attributes include the exclusivity flag excl for RMWs, and the access mode for RC11-style models. (Additional kinds of events exist for memory allocations, deallocations, assertion violations, *etc.*, but these do not affect the model checking algorithm in any meaningful way.)

We refer to as *plain* events those who have a footprint of a single byte access $b \in f$. Each (memory access) event $e = \langle t, i, f, lab\rangle$[1] induces a set of plain labels $\mathtt{plain}(e) \triangleq \left\{\langle t, i, b, lab\rangle \;\middle|\; b \in f\right\}$, by breaking down the access to its byte-level constituents. We subscript with P to refer to the set of plain events that corresponds to a set of events, i.e., we write $\mathtt{W}_P$ to refer to the set of plain write events.

Having defined events, we define execution graphs as follows.

Definition 4.2. An *execution graph* G comprises the following components:

(1) a set of events E that includes init and does not contain multiple events with the same thread identifier and serial number;

[1] We assume that programs give rise to accesses whose values fit into the respective footprint.

(2) $\mathtt{rf} : \mathsf{E} \cap \mathsf{R}_P \rightarrow \mathsf{E} \cap \mathsf{W}_P$, called the *reads-from* function, mapping each plain read event to the plain write where it gets its value for the specified byte;
(3) $\mathtt{co} \subseteq \bigcup_{l \in \mathrm{Loc}, b \in \mathrm{Byte}} \mathsf{W}_{l,b} \times \mathsf{W}_{l,b}$ (where $\mathsf{W}_{l,b} \triangleq \{\mathtt{init}\} \cup \{\langle t, i, b, lab\rangle \in E \mid lab = \mathsf{W}^{-}(l, \_)\}$) called the *coherence order*, a strict partial order that is total on $\mathsf{W}_{l,b}$ for every location $l \in$ Loc and byte $b \in$ Byte; and
(4) $\leq$, a total order on E that represents the order in which events were incrementally added to the graph.

We write $G.\mathsf{E}$, $G.\mathtt{rf}$, $G.\mathtt{co}$ and $\leq_G$ to project the various components of an execution graph. Given two events $e_1, e_2 \in G.\mathsf{E}$, we write $e_1 <_G e_2$ if $e_1 \leq_G e_2$ and $e_1 \neq e_2$.

In relational algebra expressions, we abuse notation and also use $G.\mathtt{rf}$ to denote the relation $\{\langle G.\mathtt{rf}(r), r\rangle \mid r \in G.\mathsf{R}\}$, which relates a write event to the read events that read from it.

We assume that $\mathtt{init} \in \mathsf{W}$, and omit the $\emptyset$ for read/write labels with no attributes.

The functions tid, idx, footp, loc, and mod respectively return the thread identifier, serial number, footprint, location, and access mode, when applicable.

We write $G.\mathsf{W}$ for $G.\mathsf{E} \cap \mathsf{W}$ (and similarly for other sets), and use superscript and subscripts to restrict label sets (e.g., $\mathsf{W}_l \triangleq \{\mathtt{init}\} \cup \{w \in \mathsf{W} \mid \mathtt{loc}(w) = l\}$).

An execution graph does not explicitly track the *program order* (po) via another component, since it can deduced by the thread identifiers and serial numbers of its events. The initialization event is ordered before every other event in the po order.

We define the causal relation $G.\mathsf{po}\mathtt{rf} \triangleq (G.\mathsf{po} \cup G.\underline{\mathtt{rf}})^{+}$, where underlining lifts a relation A relating plain events to a relation that relates whole (non-plain) events

$$\underline{A} \triangleq \left\{\langle e_1, e_2\rangle \;\middle|\; \langle b_1, b_2\rangle \in A, b_i \in \mathtt{plain}(e_i)\right\}$$

4.2 Conditions on the Memory Consistency Model

The memory consistency model (or memory model, for short) is a predicate consistent$(\cdot)$ on execution graphs, specifying which executions are consistent.

Similar to existing DPOR algorithms [Kokologiannakis et al. 2022, 2024, 2019b], we do not assume a specific memory model; Mixer is parametric in the choice of the memory model, provided it satisfies the following basic assumptions.

(1) *prefix-closed* w.r.t. to the causal relation (i.e., for every consistent execution G and set $D \subseteq G.\mathsf{E}$ s.t. $dom(G.\mathsf{po}\mathtt{rf}; D) \subseteq D$, the restriction G_D of G to the events of D is consistent),
(2) po$\mathtt{rf}$-*acyclic* (i.e., if G is consistent, then $G.\mathsf{po}\mathtt{rf}$ is acyclic), and
(3) *extensible*, i.e., we can preserve consistency of an execution G when extending it with the events of an instruction by adding them *maximally*: a read instruction will read from co-maximal plain write, and a write instruction's plain writes will be placed at the end of co.

These conditions are readily satisfied by most memory models: (1) any model s.t. removing edges does not invalidate consistency is prefix-closed, (2) any model whose operational semantics disallow instructions being executed out of their program order, and loads being executed before the stores they read from, is porf-acyclic, and (3) any model whose operational semantics does not get "stuck", is extensible. For a more thorough discussion we refer the reader to [Kokologiannakis et al. 2022]

Memory models such as SC and its direct lifting to MSA, RC11 [Lahav et al. 2017], Relase-Acquire (RA), x86-TSO [Owens et al. 2009], as well as a suggested MSA extension of x86-TSO [Alglave et al.

Algorithm 1 Mixer: DPOR for MSA programs

```
procedure Explore_P(G)
    if IsErroneous then exit("Error")
    $a \leftarrow$ AddNextEvent_P(G)
    if $a \in \mathsf{R}$ then
        $b, \ldots, b+l-1 \leftarrow \mathtt{footp}(a)$
        for $Bs \in G.\mathsf{W}_{\mathtt{loc}(a),b} \times \ldots \times G.\mathsf{W}_{\mathtt{loc}(a),b+l-1}$ do
            ExploreIfConsistent_P(SetRF(G, a, Bs))
    else if $a \in \mathsf{W}$ then
        ExploreCOs_P(G, a)
        $b, \ldots, b+l-1 \leftarrow \mathtt{footp}(a)$
        for $K \subseteq G.\mathsf{W}_{\mathtt{loc}(a)}$ such that $\mathsf{TieBreak}(K, a) \land \mathsf{Independent}(G, K \cup \{a\})$ do
            for $r \in G.\mathsf{R}_{\mathtt{loc}(a)}$ such that $\langle r, a \rangle \notin G.\mathsf{porf}$ do
                $Deleted \leftarrow \{e \in G.\mathsf{E} \setminus \{a\} \mid r <_G e \land \forall w \in K \cup \{a\}.\, \langle e, w \rangle \notin G.\mathsf{porf}\}$
                if MaximalExtension(G, $\{r\} \cup Deleted$) then
                    for $Bs \in G.\mathsf{W}_{\mathtt{loc}(a),b} \times \ldots \times G.\mathsf{W}_{\mathtt{loc}(a),b+l-1}$ do
                        if $\mathsf{Relevant}(K \cup \{a\}, Bs)$ then
                            ExploreCOs_P(SetRF($G \setminus Deleted$, r, Bs), a)
    else if $a \neq \bot$ then
        Explore_P(G)

procedure ExploreIfConsistent_P(G)
    if $\mathsf{consistent}_{\mathsf{M}}(G)$ then Explore_P(G)

procedure ExploreCOs_P(G, a)
    $b, \ldots, b+l-1 \leftarrow \mathtt{footp}(a)$
    for $Bs \in G.\mathsf{W}_{\mathtt{loc}(a),b} \times \ldots \times G.\mathsf{W}_{\mathtt{loc}(a),b+l-1}$ do
        ExploreIfConsistent_P(SetCO(G, a, Bs))
```

2021], directly satisfy these conditions[2]. Models such as POWER and ARM [Alglave et al. 2014], as well as ARM's MSA extension [Alglave et al. 2021], allow load buffering behaviors, and therefore porf-cyclic behaviors. However, they satisfy similar criteria, with the only difference being that they use a weakening of the program order po. Supporting such models is achieved by effectively lifting the DPOR to use this weakened relation instead of po [Kokologiannakis and Vafeiadis 2020]. The same approach[Kokologiannakis and Vafeiadis 2021] applies for the Linux Kernel Memory Model, although the semantics of mixed-size accesses are not defined [Alglave et al. 2018]. Our work is orthogonal to these differences, and extends to such models as well by following the same approach.

4.3 The Mixer Exploration Algorithm

Let us now proceed by showing how Mixer enumerates all consistent execution graphs of a MSA program P. The algorithm is shown in Algorithm 1.

[2] To see this for the MSA extension of x86-TSO, observe that (1) removing $\mathsf{po} \cup \mathsf{rf}$-maximal events does not add any edge and cannot affect consistency, (2) any porf cycle can be rewritten to use rf edges between different threads and po edges from a read (rf entering the thread) to a write (rf edge exiting the thread), both of which are included in a relation which is postulated to be acyclic, and (3) any instruction added maximally in coherence cannot introduce a cycle since it has no $\mathsf{co} \cup \mathsf{rb} \cup \mathsf{rf}$-successors.

Mixer enumerates all consistent execution of a MSA program P by incrementally adding one event at a time. Exploration is initiated by calling Explore$_P$ with the empty execution graph $G_\emptyset$.

The AddNextEvent returns the event corresponding to the leftmost thread that has not fully executed, if such an event exists, otherwise it returns $\bot$.

If the new event is a read (Line 4), Mixer considers all sets of plain writes that the new read can read from, sets the rf component of the graph to reflect this choice (Line 7), and recursively calls Explore$_P$ with the new graph, if it is consistent (Line 21).

If the new event is a write (Line 8), Mixer considers all co placements for the new write by enumerating the possible co-predecessors, for each byte in the write's footprint (Line 24), sets the co component of the graph to reflect this choice (Line 25), and recursively calls Explore$_P$ with the new graph, if it is consistent (Line 21).

Additionally, in the case of a write event a, Mixer considers revisiting a previously added read event. For this, it enumerates all valid keep set options (Line 11), i.e., a set of writes K accessing the same location such that they are in a thread to the left of a's thread ($\mathsf{TieBreak}(K, a)$), and, together with the new write a, they form a porf-independent set K' ($\mathsf{Independent}(G, K')$).

$$\mathsf{TieBreak}(K, a) \triangleq \forall w \in K.\, w <_{\mathsf{next}} a$$
$$\mathsf{Independent}(G, K') \triangleq \forall w_1, w_2 \in K'.\, w_1 = w_2 \vee \langle w_1, w_2 \rangle \notin G.\mathtt{porf}$$

For each such keep set and each previously added read event that accesses the same location and is not porf-before the write a (Line 12), Mixer considers revisiting the read event, by deleting all events that are are added after the read and are not porf-before the new write or a write in K. Similar to TruSt, Mixer employs a maximality condition to check if the revisit should be considered, which is crucial to guarantee that no duplicate executions are explored [Kokologiannakis et al. 2022].

Concretely, the condition checks that all the events A affected by the revisit, i.e., the read r and the deleted events, form a maximal extension w.r.t. the execution without the events of A, in the order $\leq_G$ that they were added.

$$\textsc{MaximalExtension}(G, A) \triangleq \forall a \in A.\, \textsc{IsMaximal}(G, a, \left\{e \in G.\mathsf{E} \;\middle|\; e \notin A \vee e \leq_G a\right\})$$
$$\textsc{IsMaximal}(G, r \in \mathsf{R}, S) \triangleq \mathit{rng}([r]; G.\underline{\mathtt{rb}}; [S]) = \emptyset$$
$$\textsc{IsMaximal}(G, w \in \mathtt{W}, S) \triangleq \mathit{rng}([w]; G.\underline{\mathtt{co}}; [S]) = \emptyset$$

If this check succeeds, Mixer considers again all set of plain writes Bs that the read can read from, provided that each choice makes the keep set relevant, i.e., each write in $K \cup \{a\}$ is, at least partly, read ($\mathsf{Relevant}(K', Bs)$), and then proceeds to consider the possible co placements of the new write, and explore the resulting graph, if consistent (Line 17).

$$\mathsf{Relevant}(K', Bs) \triangleq \forall w \in K'.\, \exists b \in Bs.\, b \in \mathtt{plain}(w)$$

4.4 Termination, Soundness, Completeness, and Optimality

Under any memory model satisfying the assumptions of § 4.2, Mixer terminates and is sound, complete, and optimal.

Theorem 4.3 (Correctness).

(1) Explore$_P$($G_\emptyset$) terminates.
(2) Explore$_P$($G_\emptyset$) only explores consistent executions.
(3) Explore$_P$($G_\emptyset$) explores every consistent execution.
(4) Explore$_P$($G_\emptyset$) never explores the same G twice.

Soundness is trivial, since consistency of the explored execution is checked at each step (Line 21).

Completeness, i.e., every consistent execution is explored, and optimality, i.e., no execution is explored twice, are more involved. Our proof partially follows the proof of Awamoche [Kokologiannakis et al. 2023b] and drove most of the design of our algorithm.

Concretely, the restrictions on the keep sets enable a key property for the correctness of the algorithm: given a partial execution and a read instruction that needs to read from a set of write instructions not yet present in the execution, there is a predetermined write instruction among them and a unique keep set that will revisit the read instruction in order to reach the desired execution.

4.5 Strong Optimality

Apart from the regular notion of optimality, there exists a stronger version of optimality enjoyed by DPOR algorithms [Kokologiannakis et al. 2022, 2023b], which states that no *wasteful* exploration takes place.

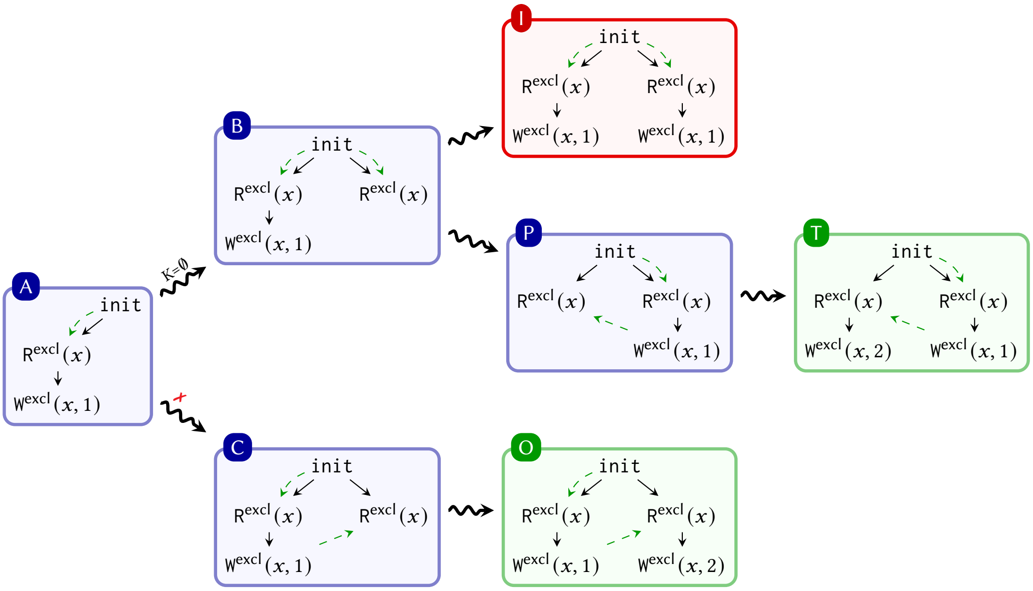
Intuitively, exploring a (partial) execution G, i.e., initiating a call to $\textsc{Explore}_P(G)$, is wasteful if it does not eventually lead to call to $\textsc{Explore}_P(G_c)$, for a consistent and complete execution G_c. We refer to such executions are *fruitless*.

Strong optimality states that any explored fruitless execution is *blocked*: it does not initiate any further exploration. Unfortunately, our algorithm does not satisfy this condition, but a weaker, generalized version of it.

Fruitless executions in TruSt. Let us first explain why a fruitless execution is encountered in the first place in algorithms like TruSt, why it is guaranteed to be blocked, and why this fails to hold for our algorithm.

Example 4.1 Consider the program below and how TruSt would obtain its two executions O and T.

$$\mathsf{T1}\text{: } \texttt{fetch\_add}(x, 1) \;\Big\|\; \mathsf{T2}\text{: } \texttt{fetch\_add}(x, 1) \tag{U+U}$$


A
init
$\mathsf{R}^{\mathsf{excl}}(x)$
$\mathsf{W}^{\mathsf{excl}}(x, 1)$
K=∅
B
C
I
P
T
O
$\mathsf{W}^{\mathsf{excl}}(x, 2)$

Since RMW instructions perform both a read and a write access, it is convenient to treat them as two events that are added in succession (with the appropriate exclusivity attribute excl, to capture the atomicity semantics). TruSt would first add the two events corresponding to the first RMW in the unique consistent way (A). Afterwards, it needs to consider two options B and C for the read event of the second RMW: reading from the first leads to O, while reading from the initialization write will lead to T. In the latter case, the read event will read from the initialization write, and adding the following write event of the RMW, and revisiting the first read (P) leads to the desired execution. Notice, however, that adding the write and **not** performing the revisit leads to an inconsistent execution: in I, both RMWs read from the same write.

Extensibility holds for the events corresponding to the RMW only when considered together. If the read event is added already in an arbitrary way, there is no guarantee that the write event can also be added, as it happened it the previous example. Moreover, it can be the case that the revisit step is also not allowed, e.g., due to the maximality check, rendering the execution with the pending RMW wasteful. TruSt, however, guarantees that this wasteful exploration can only lasts for one "step": as soon as a fruitless execution is reached, it is blocked, i.e., it does not lead to another recursive call (which would further increase the size of the wasteful exploration).

Mixer: Depth-Bounded Fruitless Exploration. Mixer can initiate fruitless explorations that are not immediately blocked, and therefore is not strongly optimal. It does, however, satisfy a more generalized version of this property. Concretely, given that an instruction can access at most B

bytes, Mixer can do at most B consecutive fruitless steps, i.e., successive calls to $\text{Explore}_P(\cdot)$, before stopping the exploration[3].

Formally, we say that an execution G is N-fruitless if it is fruitless ($\text{Explore}_P(G)$ does not lead to a call $\text{Explore}_P(G_c)$, for complete execution G_c) and there exists a sequence of at least N consecutive calls to $\text{Explore}_P(\cdot)$ starting from $\text{Explore}_P(G)$. Mixer guarantees that there exist no $(B + 1)$-fruitless execution, which allows us to bound the overhead of the wasteful executions to a polynomial factor w.r.t. the size of the executions (i.e., the number of threads and events).

Theorem 4.4 (Bounded Waste). *For any K-fruitless execution explored by $\text{Explore}_P(G_\emptyset)$, $K \leq B$.*

Proof sketch. To see this, let G be a K-fruitless execution. Then, the K calls to Explore_P all perform revisits from and to events of $K + 1$ RMW instructions. Due to the maximality condition, it is easy to see that when an exclusive write w (the write part of an RMW) revisits an exclusive read r (the read part of an RMW) and forces r to read byte i of location x from w, then r is the only event in the resulting execution that reads x_i maximally. Therefore no other RMW accessing x_i can later be revisited, i.e., the rest RMWs in the revisit chain must access different bytes (of the same location). This bounds the number of the K RMW instructions (not counting the initial one) to the maximum number of distinct footprints that instructions can access. □

5 Extension for Transactions

As we discussed in §2, the main issue that renders traditional DPOR algorithm inappropriate for MSA programs is the lack of extensibility. However, there is a larger class of programs that suffers from the same issue: transactional programs.

Similar to a program with MSA, given a transactional program where a transaction has been partially executed, there is no guarantee that there exists a consistent way to complete the transaction. All the examples presented in §2 and §3 can be trivially translated to the equivalent transactional variants with each MSA instruction being replaced with the equivalent byte-size accesses in a transactions assuming Serialization semantics (i.e., the allowed behaviors are interleavings of transactions), and all our observations still hold.

In fact, Mixer as well can be easily extended to cover transactional programs. To see this, observe how a transaction's accesses can be represented by a set of read accesses, followed by a set of write accesses: due to the minimal guarantees under a transactional program, a read that is preceded by a write to the same location must read from this write, while read and write accesses to different locations inside a transaction can be reordered.

To handle transactional programs, Mixer can assume an AddNextEvent_P function that treats a transactional program as two successive placeholder events, a read and a write. A read event r, however, no longer tracks locations, since they cannot be known in advance. Instead, the locations are discovered by executing the transaction: for each encountered read access, a subexploration is initiated for each possible rf option, in order to discover the rest of the accesses in the transaction. In the end, all co choices for the encountered write accesses are considered, similar to Algorithm 1.

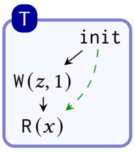
For the fragment of transactional programs where the read accesses do not depend on each other, the modifications discussed so far are sufficient. Otherwise, a more liberal revisiting condition is needed that considers revisits even if there are (currently) no conflicting read accesses to the transaction to be revisited.

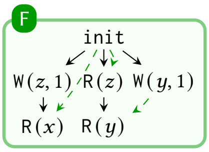
Example 5.1 To see why, consider the program below where the first thread comprises two transactions containing one instruction each, while the other two threads contain a single transaction

[3] Similar to TruSt, these fruitless executions can only arise in the presence of RMW instructions

each. (We use square brackets to denote the transactions.)

```
T1: [z := 1]   ||  T2: [if(z == 0)         ]  ||  T3: [y := 1]
    [a := x]   ||      [   if(y == 0)      ]  ||
               ||      [      x := 1       ]  ||
```





In order to reach the right-to-left execution F, Mixer would need to reach the execution T that has only the first thread, and extend it in a consistent way with the transaction of the second thread, before executing the third thread's transaction and revisiting the second thread.

Observe, however, that while it is consistent to maximally extend the second thread to read 1 from z, the access to y does not appear in this case and therefore the conflict with the third thread is not present. On the other hand, extending the second thread to read 0 from y is not consistent, since the write to x cannot be added: the first thread's read to x would need to read from it.

To overcome this, Mixer would need to conservatively revisit any transaction that might have read from the locations that the current transaction is writing to, and drop the post-revisit execution if no conflict actually occurred, in line with the relevance restriction of §3.2.

While the modifications discussed in this section do not impact our main correctness results (Theorem 4.3), the bound on wasteful explorations presented in §4.5 no longer holds: one cannot assume a bounded set of locations accessed by transactions. Instead, the depth of the fruitless exploration can be bounded by the number of threads, if we assume a maximum number of threads T for the program to be verified. To see this, observe that after all T threads include a transaction (write) that has revisited (and therefore it is the last transaction in the thread), any other revisit is blocked since it would delete such a write.

6 Implementation

6.1 Tool

We implemented Mixer as a tool for C/C++ programs based on the LLVM Intermediate Representation (IR) [Lattner and Adve 2004].

Mixer is implemented on top of the GenMC [Kokologiannakis et al. 2019b] tool and required significant changes, mainly on the data-structures used to capture the execution's rf and co components. Read events no longer have a single rf edge, but instead track a set of edges to write events, annotated with the footprint accessed.

To minimize the intrusive changes to the implementation, coherence co is still stored as a set of per-word co_w coherence lists. However, each such co_w list no longer represents a total order, since some same-word writes access disjoint sets of bytes. Instead, this list is just a linearization of a partial order of (conflicting) writes to the same word.

A downside of this treatment is that one needs to calculate the possible ways a new write can be inserted in coherence by exploring permutations of these orderings: only exploring possible placements of the new write is not enough. To see why, consider a coherence list with two disjoint write accesses, and a new write that conflicts with both of them. We need to consider all three placements of the new write, as well as permuting the two old writes, and adding the new write in the middle (since it transitively induces an ordering between the disjoint writes).

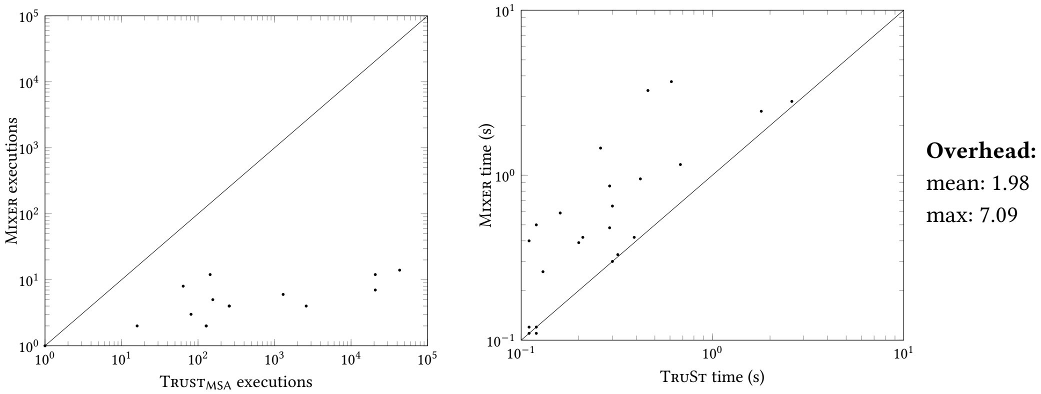


Fig. 1. Performance comparison $\text{Trust}_{\text{MSA}}$/Mixer (left) and Mixer overhead over TruSt (right)

6.2 RA_{MSA}: An Example Memory Model

While our algorithm is parametric to the memory model, our implementation currently only supports RA_{MSA}, a simple lifting of the relaxed and acquire/release fragment of the repaired C/C++11 memory model to MSA accesses.

Given the usual happens before relation $\mathtt{hb}$ defined by this fragment of RC11[Lahav et al. 2017], an execution G is consistent under RA_{MSA} if (1) $G.\mathtt{porf}$ is acyclic, (2) $G.\mathtt{hb}; G.\mathtt{eco}$ is irreflexive, and (3) $G.\mathtt{rmw} \cap (G.\mathtt{rb}; G.\mathtt{co}) = \emptyset$, where $G.\mathtt{eco} \triangleq ((\underline{\mathtt{co}} \cup \underline{\mathtt{rb}}); \underline{\mathtt{rf}}? \cup \underline{\mathtt{rf}})^+$ and $G.\mathtt{rmw}$ relates a plain read to a plain write belonging to the same RMW instruction. It is easy to see that RA_{MSA} satisfies the conditions of §4.2.

7 Evaluation

We now evaluate Mixer's performance on a set of synthetic and non-synthetic benchmarks. Our evaluation aims to establish the following points:

§7.1 Mixer is exponentially faster than approaches that do not directly tackle mixed-size accesses
§7.2 Mixer only incurs a moderate overhead on non-mixed-size-accesses benchmarks
§7.3 Mixer can handle realistic code with mixed-size accesses employed in production

To accomplish this, we perform the following case studies. First, we compare Mixer with $\text{Trust}_{\text{MSA}}$, a naive DPOR that treats mixed-size accesses as byte sequences and achieves correctness by enforcing atomicity at the end of each execution. Then, we measure the overhead of Mixer over TruSt (implemented in the GenMC tool[Kokologiannakis and Vafeiadis 2021]) on GenMC's standard test suite. Finally, we run Mixer on some mixed-size-accesses code extracted from the Linux kernel.

Experimental Setup. We conducted all experiments on a Dell PowerEdge R6525 system running a custom Debian-based distribution with 2 AMD EPYC 7702 CPUs (256 cores @ 2.80 GHz) and 2TB of RAM. We set the timeout limit to 30 minutes (denoted by 🕒). All times are in seconds.

7.1 $\text{Trust}_{\text{MSA}}$ vs Mixer

We compare Mixer and $\text{Trust}_{\text{MSA}}$ on a set of synthetic benchmarks. Although these benchmarks are small, they suffice to demonstrate that naive approaches do not scale since even a small number of mixed-size accessed (e.g., 3-7) are enough to make $\text{Trust}_{\text{MSA}}$ explore a huge number of redundant executions.

Table 1. Realistic benchmarks

	Execs	*Time*
lockref(2)	5	0.03
lockref(3)	142	0.04
lockref(4)	27 192	3.52
lockref(5)	🕒	🕒

The results can be seen on the left in Fig. 1. (For Trust$_{\text{MSA}}$, the number of executions reported is the total number of executions considered, before filtering out the ones that are inconsistent.) As can be seen, naive approaches are clearly impractical, and do not scale even for very simple benchmarks. Mixer, on the other hand, only considers a very small number of executions as it is able to check full consistency at each step, thereby ruling out a large number of infeasible behaviors.

7.2 Mixer's Overhead

We compare Mixer and GenMC on a set of benchmarks that do not have mixed-size accesses to evaluate the overhead that Mixer imposes on the implementation of GenMC. We removed from our tests the small benchmarks where both tools terminate immediately (less that 0.1s).

The results can be seen on the right in Fig. 1. Mixer only imposes a moderate overhead (on average takes a bit less than twice the execution time). In the worst case, the overhead is around 7 times, but this is due to the fact that the current Mixer's implementation does not support an optimization related to the handling of locks.

A more methodical implementation can, in principle, almost fully negate the overhead imposed by the MSA implementation. The only inherent exception is the restriction imposed on the scheduler (§3.2) to follow a left-to-right scheduling. This precludes a common heuristic strategy to always prioritize write instructions, since it reduces the number of revisits (and therefore reinterpretation of the program) performed. However, we expect that non-MSA programs can be easily detected (either manually, or with a conservative static pass), disabling the MSA flag of our tool, which would revert to the default heuristics of GenMC.

7.3 Realistic Benchmarks

We run Mixer on lockref [Corbet 2013], a data-structure extracted from the Linux kernel. As we have already discussed in §1, lockref uses a union to pack a number of fields in a single word. The union is either accessed as a whole (effectively atomically exchanging all the fields with new values), or parts of it (by accessing the specified field).

We use a simple client that is parametric to the number of threads (denoted as lockref(N)). The first thread acquires and releases the lock, while the other threads try to increase the reference count. Mixer manages to verify a client of 4 threads, before the timeout, due to the large number of behaviors this data-structure can produce. Observe that, even with three threads, there are more than 100 different possible executions, and adding one more thread quickly blows up the number of executions.

8 Related Work

Abdulla et al. [2014] first presented an optimal DPOR algorithm which explores all sequentially consistent behaviours of a program exactly once, while possibly consuming an exponential amount of memory. In subsequent work, Abdulla et al. [2015] developed DPOR algorithms for weaker memory models, such as x86-TSO.

In another line of work, Kokologiannakis et al. [2017, 2022, 2019a,b] developed graph-based DPOR algorithms, which culminated in the TruSt algorithm, the first optimal DPOR algorithm with polynomial space requirements. TruSt is implemented in the genmc model checker [Kokologiannakis and Vafeiadis 2021], and is parametric in the choice of the memory consistency model. The genmc implementation has builtin support for a bunch of memory models –SC, TSO, RA, RC11, and IMM– and can easily extended with any declarative model that can be expressed in a restricted fragment of the relational calculus with the kater tool [Kokologiannakis et al. 2023a].

The work of Bouajjani et al. [2023] explored the application of DPOR on transactional programs. They define a class of DPOR algorithms and show that there exists no strongly optimal such algorithm for memory models such as Serializability (SER) and Snapshot Isolation (SI), which do not satisfy a form of transactional extensibility. Instead, they provide a strongly optimal DPOR algorithm for the weaker memory models that satisfy their criterion. For the case of SER and SI, they suggest verifying under a weaker memory model and only performing the SER (or SI) consistency check when a complete execution in explored.

Regarding the semantics of mixed-sized accesses, Flur et al. [2017] first investigated their behavior under the Arm (version 8) and POWER hardware architectures, and extended their previous operational models of these architectures [Flur et al. 2016; Sarkar et al. 2011] to support MSAs. They also extended the declarative C/C++11 model to support non-atomic MSAs, and established the correctness of compilation mappings to POWER.

More recently, Alglave et al. [2021] provide a formal declarative memory model for the Arm architecture that supports MSAs, as well as a proposal for a similar extension to the x86-TSO declarative model. Both of these models are based on execution graphs and fit within the framework of this paper.

9 Conclusion

We present Mixer, the first DPOR algorithm that supports programs with mixed-sized accesses. Such programs cannot be handled by a simple encoding where instructions are broken down to byte-level accesses due to the lack of extensibility in the resulting model.

Mixer operates on the level of whole instructions and introduces the novel notion of multi-write revisits to obtain a complete and optimal DPOR algorithm for MSA programs. Mixer does initiate executions that are fruitless, but we provide a constant bound on the depth of such explorations. Our implementation shows that Mixer is faster than the naive approach based on prior DPOR algorithms and can verify realistic programs of moderate size that employ mixed-size accesses.

As discussed in §5, our observations and most of our results extend to transactional programs, a class of programs that subsumes MSA programs. We leave the implementation of Mixer for transactional programs and the comparison with Bouajjani et al. [2023] for future work.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. This work was supported by a European Research Council (ERC) Consolidator Grant for the project “PERSIST” under the European Union’s Horizon 2020 research and innovation programme (grant agreement No. 101003349).

References

Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas. 2015. “Stateless model checking for TSO and PSO.” In: *TACAS 2015* (LNCS). Vol. 9035. Springer, Berlin, Heidelberg, 353–367. doi: 10.1007/978-3-662-46681-0_28.

Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. 2014. “Optimal dynamic partial order reduction.” In: *POPL 2014.* ACM, New York, NY, USA, 373–384. doi: 10.1145/2535838.2535845.

Jade Alglave, Will Deacon, Richard Grisenthwaite, Antoine Hacquard, and Luc Maranget. July 2021. "Armed Cats: Formal Concurrency Modelling at Arm." *ACM Trans. Program. Lang. Syst.*, 43, 2, (July 2021). DOI: 10.1145/3458926.

Jade Alglave, Luc Maranget, Paul E. McKenney, Andrea Parri, and Alan Stern. 2018. "Frightening small children and disconcerting grown-ups: Concurrency in the Linux kernel." In: *ASPLOS 2018*. ACM, Williamsburg, VA, USA, 405–418. DOI: 10.1145/3173162.3177156.

Jade Alglave, Luc Maranget, and Michael Tautschnig. July 2014. "Herding cats: Modelling, simulation, testing, and data mining for weak memory." *ACM Trans. Program. Lang. Syst.*, 36, 2, (July 2014), 7:1–7:74. DOI: 10.1145/2627752.

Ahmed Bouajjani, Constantin Enea, and Enrique Román-Calvo. June 2023. "Dynamic Partial Order Reduction for Checking Correctness against Transaction Isolation Levels." *Proc. ACM Program. Lang.*, 7, PLDI, (June 2023). DOI: 10.1145/3591243.

Jonathan Corbet. 2013. *Introducing lockrefs*. (2013). http://lwn.net/Articles/565734/.

Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. 2016. "Modelling the ARMv8 architecture, operationally: Concurrency and ISA." In: *POPL 2016*. ACM, St. Petersburg, FL, USA, 608–621. DOI: 10.1145/2837614.2837615.

Shaked Flur, Susmit Sarkar, Christopher Pulte, Kyndylan Nienhuis, Luc Maranget, Kathryn E. Gray, Ali Sezgin, Mark Batty, and Peter Sewell. 2017. "Mixed-size concurrency: ARM, POWER, C/C++11, and SC." In: *POPL 2017*. ACM, Paris, France, 429–442. DOI: 10.1145/3009837.3009839.

Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis. Dec. 2017. "Effective stateless model checking for C/C++ concurrency." *Proc. ACM Program. Lang.*, 2, POPL, (Dec. 2017), 17:1–17:32. DOI: 10.1145/3158105.

Michalis Kokologiannakis, Ori Lahav, and Viktor Vafeiadis. Jan. 2023a. "Kater: Automating Weak Memory Model Metatheory and Consistency Checking." *Proc. ACM Program. Lang.*, 7, POPL, (Jan. 2023). DOI: 10.1145/3571212.

Michalis Kokologiannakis, Iason Marmanis, Vladimir Gladstein, and Viktor Vafeiadis. Jan. 2022. "Truly stateless, optimal dynamic partial order reduction." *Proc. ACM Program. Lang.*, 6, POPL, (Jan. 2022). DOI: 10.1145/3498711.

Michalis Kokologiannakis, Iason Marmanis, and Viktor Vafeiadis. June 2024. "SPORE: Combining Symmetry and Partial Order Reduction." *Proc. ACM Program. Lang.*, 8, PLDI, (June 2024). DOI: 10.1145/3656449.

Michalis Kokologiannakis, Iason Marmanis, and Viktor Vafeiadis. 2023b. "Unblocking Dynamic Partial Order Reduction." In: *CAV 2023* (LNCS). Ed. by Constantin Enea and Akash Lal. Vol. 13964. Springer, 230–250. DOI: 10.1007/978-3-031-37706-8_12.

Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. Oct. 2019a. "Effective lock handling in stateless model checking." *Proc. ACM Program. Lang.*, 3, OOPSLA, (Oct. 2019). DOI: 10.1145/3360599.

Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. 2019b. "Model checking for weakly consistent libraries." In: *PLDI 2019*. ACM, New York, NY, USA. DOI: 10.1145/3314221.3314609.

Michalis Kokologiannakis and Viktor Vafeiadis. 2021. "GenMC: A model checker for weak memory models." In: *CAV 2021* (LNCS). Ed. by Alexandra Silva and K. Rustan M. Leino. Vol. 12759. Springer, 427–440. DOI: 10.1007/978-3-030-81685-8_20.

Michalis Kokologiannakis and Viktor Vafeiadis. 2020. "HMC: Model checking for hardware memory models." In: *ASPLOS 2020* (ASPLOS '20). ACM, Lausanne, Switzerland, 1157–1171. DOI: 10.1145/3373376.3378480.

Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. 2016. "Taming Release-acquire Consistency." In: *POPL 2016*. ACM, St. Petersburg, FL, USA, 649–662. DOI: 10.1145/2837614.2837643.

Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. "Repairing sequential consistency in C/C++11." In: *PLDI 2017*. ACM, Barcelona, Spain, 618–632. DOI: 10.1145/3062341.3062352.

Chris Lattner and Vikram Adve. 2004. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation." In: *CGO 2004*. IEEE Computer Society, Palo Alto, California, 75. DOI: 10.5555/977395.977673.

Antoni Mazurkiewicz. 1987. "Trace Theory." In: *PNAROMC 1987* (LNCS). Vol. 255. Springer, Berlin, Heidelberg, 279–324. DOI: 10.1007/3-540-17906-2_30.

Scott Owens, Susmit Sarkar, and Peter Sewell. 2009. "A better x86 memory model: x86-TSO." In: *TPHOLs 2009*. Springer, Munich, Germany, 391–407. DOI: 10.1007/978-3-642-03359-9_27.

Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. 2011. "Understanding POWER multiprocessors." In: *PLDI 2011*. ACM, 175–186. DOI: 10.1145/1993498.1993520.

Dennis Shasha and Marc Snir. Apr. 1988. "Efficient and correct execution of parallel programs that share memory." *ACM Trans. Program. Lang. Syst.*, 10, 2, (Apr. 1988), 282–312. DOI: 10.1145/42190.42277.

Received 2024-07-11; accepted 2024-11-07