Dynamic Partial Order Reductions for Spinloops

Michalis Kokologiannakis
MPI-SWS
Kaiserslautern, Germany
michalis@mpi-sws.org

Xiaowei Ren
The University of British Columbia
Vancouver, Canada
xiaowei@ece.ubc.ca

Viktor Vafeiadis
MPI-SWS
Kaiserslautern, Germany
viktor@mpi-sws.org

***Abstract*—Stateless model checking (SMC) coupled with dynamic partial order reduction (DPOR) is an effective way for automatically verifying safety properties of loop-free concurrent programs. SMC, however, does not work well for programs with loops because it cannot distinguish loop iterations that make progress from ones that revisit the same state. This results in redundant exploration that dominates the verification time.**

We present SAVER (Spinloop-Aware Verifier), a memory-model-agnostic SMC/DPOR extension that detects *zero-net-effect spinloops* and avoids redundant explorations that lead to the same local state. As confirmed by our experiments, SAVER achieves an exponential reduction in verification time and outperforms state-of-the-art tools in a variety of real-world benchmarks.

***Index Terms*—stateless model checking, spinloops**

I. Introduction

Stateless model checking (SMC) [16] is a prominent technique for verifying safety properties of concurrent programs, especially under weak memory consistency [23, 1, 17, 18, 20]. The key design choice that makes SMC scale is that it does not record the set of states explored, but rather uses alternative techniques, namely *dynamic partial order reduction* (DPOR) [14, 2], to avoid exploring the same state multiple times. The downside of this choice, however, is that SMC struggles with spinloops, i.e., loops that continuously read a shared variable until some condition holds: as SMC does not record the set of visited program states, it cannot distinguish loop iterations that make progress from those that return to the same state. To make matters even worse, such loops are ubiquitous in real-world concurrent programs, whether lock-based or lock-free.

Consequently, spinloops typically have to be *bounded*. Since bounding generally sacrifices the soundness of the verification, one would like to use fairly large loop bounds to be confident enough that the program verified is correct. Doing so, however, is practically infeasible. A loop bound of $N \geq 2$ typically leads to an exponential blowup in the state space, since the model checker explores the possibility of each spinloop failing $0, 1, \ldots, N-1$ times and, for each failure, all possible stores from which the spinloop loads(s) can read.

To avoid the blowup, the solution is to use a bound of $N = 1$. So far, this is typically done manually by rewriting the program to use **`assume`** statements (a.k.a. **`await`**), special verifier commands that block the execution of the relevant thread when the condition of the **`assume`** is violated.

The goal of this paper is to determine *conditions* under which it is sound to do such conversions automatically. As we shall see, this turns out to be quite challenging.

First, spinloops cannot be adequately detected by a simple syntactic criterion. Since programming languages have many ways of creating spinloops (e.g., while loops, repeat-until loops, for-loops, goto statements), their detection is best done after converting each program thread into a *control-flow graph* (CFG). However, even there, simply removing the CFG backedges for side-effect-free loops (i.e., loops with no stores to global variables or to local variables that are live at the loop header) is insufficient, as illustrated by the program below. As a convention, in our examples, we use x, y, z for global (shared) variables and $a, b, c, \ldots$ for registers.

`do` $a := x$	$b := x$
`while` $(a \neq 0)$	**`while`** $(b \neq 0)$ $b := x$

(LOOP-PEEL)

While the loop in thread I can be easily bounded by converting it into $a := x; \texttt{assume}(a = 0)$, the one in thread II cannot because b is "live" at the header of the loop (its value is used in the loop).

Second, some spinloops may have side-effects, but these either do not occur on all their iterations or are never observed by the other threads (e.g., writing to a global variable that is not concurrently read) or cancel each other out (e.g., incrementing and then decrementing a variable, acquiring and releasing a lock). As an example of the latter kind, consider the following *zero-net-effect* (ZNE) spinloops extracted from a lock implementation.

`while` $(true)$	**`while`** $(true)$
$a := \texttt{fetch\_add}(x, 1)$	$b := \texttt{fetch\_add}(x, 1)$
`if` $(a = 0)$ **`break`**	**`if`** $(b = 0)$ **`break`**
$\texttt{fetch\_add}(x, -1)$	$\texttt{fetch\_add}(x, -1)$
// critical section	*// critical section*
$\texttt{fetch\_add}(x, -1)$	$\texttt{fetch\_add}(x, -1)$

(INC-DEC-SPIN)

Each thread tries to acquire the lock by incrementing x. If the lock was already taken, it decrements x and tries again. The lock is finally released by decrementing x. Since each decrement cancels out the previous increment, we would like to avoid considering loop iterations with a decrement, i.e., unsuccessful lock acquisition attempts. The soundness of doing so depends on the context. If, for instance, there is another thread repeatedly reading x, it may observe the value of x flickering, which cannot happen if we bound the ZNE loops to a single iteration. Similarly, if another thread writes to x concurrently, the loop may no longer have a zero net effect, rendering the transformation unsound.

https://doi.org/

To address these challenges, we develop SAVER (Spinloop-Aware Verifier), a model checker that reduces spinloops to a single iteration. SAVER works at the level of reduced control flow graphs, obtained by merging bisimilar nodes. Whenever a spinloop cannot be shown to be side-effect-free statically, SAVER dynamically checks that the reduced spinloop iterations have a zero net effect (in particular, that the context does not observe any of their effects), and if the check fails, it rolls back the transformation.

We remark that our results are independent of the *memory consistency model*: they hold not only for sequential consistency (SC), but also for weak memory models, which admit executions that cannot be expressed as program interleavings.

II. PRELIMINARIES

In this section, we review how programs can be represented as control flow graphs (§ II-A), how their executions can be modeled as execution graphs (§ II-B), and how DPOR enumerates these executions (§ II-C).

A. Control Flow Graphs

To avoid cluttering the presentation, we omit all features irrelevant to loops and concurrency. We represent a concurrent program *P* as a top-level parallel composition of threads, each of which is modeled as a control-flow graph (CFG). A CFG is a directed graph whose nodes are program labels and whose edges are labeled with instructions of the following form:

$$\mathsf{Inst} \ni i ::= r := e \mid \textbf{error} \mid \textbf{assume}(e) \mid r := x \mid x := e \mid r := \texttt{fetch\_add}(x, e) \mid r := \texttt{CAS}(x, e_1, e_2)$$

where r ranges over registers (i.e., local variables), x over global (shared) variables, and e over simple expressions built from integer constants n, registers, and arithmetic operators:

$$\mathsf{Exp} \ni e ::= n \mid r \mid e_1 + e_2 \mid e_1 - e_2 \mid ...$$

Instructions comprise plain assignments; **error**, that halts the program (e.g., due to a safety violation); **assume**(e), that blocks the calling thread if e has the value zero; and memory accesses. Memory accesses include $r := x$, that reads the value of x and stores it in r; $x := e$, that stores the value contained in e in the global variable x; $r := \texttt{fetch\_add}(x, e)$ (fetch-and-increment) that atomically increments the value of x by the value of e and returns the old value to r, and $r := \texttt{CAS}(x, e_1, e_2)$ (compare-and-swap), that atomically compares the value stored in location x with the value of e_1, and if they are equal, replaces the value of x with the value of e_2. The $r := \texttt{CAS}(x, e_1, e_2)$ instruction always returns the result of the comparison in r. We also use the term *load instruction* to refer to $r := x$, $r := \texttt{CAS}(x, e_1, e_2)$, and $r := \texttt{fetch\_add}(x, e)$ instructions, while we use *store instruction* to refer to $x := e$, $r := \texttt{CAS}(x, e_1, e_2)$, and $r := \texttt{fetch\_add}(x, e)$ instructions.

We assume that input programs are deterministic in that each node n either has at most one successor (for standard program statements), or it has two successors labeled with **assume**(e) and **assume**$(\neg e)$ respectively (for conditionals and loops). As an example, Fig. 1 shows the CFGs for the two threads of the LOOP-PEEL program from §I. The loops generate cycles in the CFGs, and the conditional tests (whether to execute another loop iteration or to exit the loop) generate the edges labeled with **assume** statements.

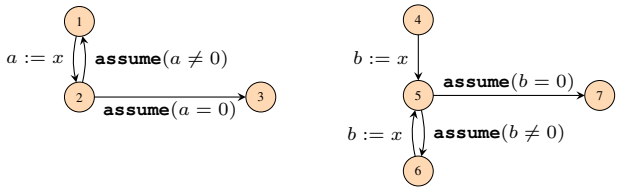


Fig. 1. CFGs for the two threads of LOOP-PEEL.

A *path* π in a CFG is an alternating sequence of nodes and instructions corresponding to edges in the CFG, starting and ending with a node. That is, π is of the form $n_1 i_1 n_2 i_2 n_3 \ldots n_{k-1} i_{k-1} n_k$ where (n_j, i_j, n_{j+1}) is an edge in the CFG for all $1 \leq j < k$. As it is common in the literature, we are primarily interested in *simple paths*, which do not visit the same node twice, except possibly by their last node. A (simple) path is *cyclic* if it starts and ends with the same node, while a *lasso* path is one whose end node is one of its intermediate nodes. We write $|\pi|$ to denote the length of the path (i.e., the number of edges it contains), and $\pi(k)$ to project the k^{th} node and/or instruction of the path.

We say that node a *dominates* b if all paths from the entry node of the CFG to b contain a. Given a path π in a CFG, we say that a node h of π is its *header* if it dominates all nodes in π. By definition, paths can have at most one header; in the case of reducible graphs, every cyclic path has a header. For example, in Fig. 1, nodes 1 and 5 are the headers of the two cyclic paths, respectively.

A loopy path is a simple path that starts and ends at its header. Formally, a simple path π is called a *loopy path* of an edge $n \to h$ if $\pi(1) = \pi(|\pi|) = h$ and $\pi(|\pi| - 1) = n$ and h dominates all nodes in π (i.e., h is a header of π).

B. Execution Graphs

In order to keep our approach as general as possible, we follow the standard axiomatic approach of Alglave et al. [8] and represent the executions of a concurrent program as *execution graphs*. Using execution graphs allows us to keep our formalism memory-model-agnostic, as our contributions do not depend on a particular memory consistency model.

Execution graphs have two basic components:

(i) a set of events (nodes), that represent the memory accesses performed by the program, and
(ii) some relations on these events (edges), such as the *program order*, which relates events in the same thread, and the *reads-from* relation, which relates reads to writes they are reading from.

The semantics of a program *P* is given by the set of execution graphs that correspond to the instructions of the program and satisfy the consistency predicate of the underlying memory model. The purpose of the consistency predicate is to rule

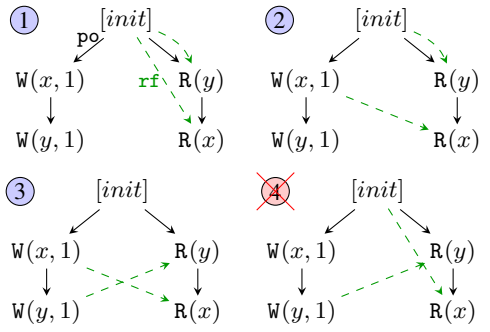


Fig. 2. MP: three consistent execution graphs under SC.

out executions with nonsensical edges, such as a load reading from a store later in program order or a store that has been overwritten by another store before the load.

To see how execution graphs model the executions of a program, consider the following example:

$$\begin{array}{l|\!|l} x := 1 & a := y \\ y := 1 & b := x \end{array} \qquad \text{(MP)}$$

Under SC, the MP program has three consistent executions, shown in Fig. 2, where the solid edges represent the program order and the green dashed edges the reads-from relation. As can be seen, execution (4) is inconsistent under SC—the consistency predicate of SC forbids the load of x to read from the initial state as the load is already aware of the $x := 1$ store. This execution, however, is allowed under certain weak memory models, such as the 'relaxed' fragment of RC11 [21].

Let us now formally describe events and execution graphs. For a more extensive discussion regarding execution graphs, we refer interested readers to Kokologiannakis et al. [18].

Definition 1. *An* event, $e \in \mathsf{Event}$, *is either an initialization event* $\langle \texttt{init}\ l \rangle$ *for a location* $l \in \mathsf{Loc}$ *or a thread event* $\langle t, i, lab \rangle$ *where* $t \in \mathsf{Tid}$ *is a thread identifier,* $i \in \mathsf{Idx} \triangleq \mathbb{N}$ *is a serial number inside each thread, and* $lab \in \mathsf{Lab}$ *is a label that takes one of the following forms:*

- *Read label:* $\texttt{R}(l)$ *where* $l \in \mathsf{Loc}$ *is the location accessed.*
- *Write label:* $\texttt{W}(l, v)$ *where* $l \in \mathsf{Loc}$ *is the location accessed, and* $v \in \mathsf{Val} \triangleq \mathbb{Z}$ *is the value written.*
- *Error label:* `error`.
- *Blocked label:* `blocked`, *generated by* $\texttt{assume}(e)$ *statements when* e *is false.*
- *ZNE label:* $\texttt{zne}(x)$, *which is used to mark ZNE loops.*

Definition 2. *An* execution graph G *consists of:*

1) *a set* $G.\texttt{E}$ *of events that includes initialization events for all locations accessed by the program, and*
2) *a function* $G.\texttt{rf}$, *called the* reads-from *map, that maps each read event of* G *to a same-location write event of* G *from where it gets its value.*

Our formal definition of execution graphs does not record the *program order* (po) as an explicit component because it can be defined directly from our representation of events:

$$\texttt{po} \triangleq \big\{ \langle \langle \texttt{init}\ l \rangle, \langle t, i, lab \rangle \rangle \mid \forall l, t, i, lab \big\} \cup \big\{ \langle \langle t_1, i_1, lab_1 \rangle, \langle t_2, i_2, lab_2 \rangle \rangle \mid t_1 = t_2 \wedge i_1 < i_2 \big\}$$

Initialization events precede all non-initialization events in po, while events in the same thread are ordered according to their serial numbers. Events from different threads are unordered.

Algorithm 1 Dynamic Partial Order Reduction

```
1:  procedure VERIFY(P)
2:      ⟨G, Γ⟩ ← ⟨G_∅, Γ_∅⟩
3:      do
4:          VISITONE(P, G, Γ)
5:      while ⟨G, Γ⟩ ← pop(Γ)

6:  procedure VISITONE(P, G, Γ)
7:      while consistent(G) ∧ a ← next_P(G) do
8:          G.E ← G.E ⊎ {a}
9:          if a ∈ error then exit("error")
10:         else if a ∈ R then
11:             let {w_0} ⊎ ws = G.E ∩ W_loc(a)
12:             G ← SetRF(G, w_0, a)
13:             Γ ← push(Γ, {SetRF(G, w, a) | w ∈ ws})
14:         else if a ∈ W then
15:             CALCREVISITS(G, Γ, a)
16:         CHECKZNEVALIDITY(G, a)
```

C. Dynamic Partial Order Reduction

DPOR verifies a program by generating all of its consistent execution graphs and checking that none of them contains an error. To do so, DPOR typically assumes some basic properties of the consistency predicate, such as prefix-closedness and extensibility [18], which are satisfied by all known memory models that follow the graph representation of § II-B.

This graph representation is also very helpful for DPOR because it encodes the independence relation that is traditionally used by DPOR algorithms to decide which interleavings should be explored. Indeed, under sequential consistency, each graph corresponds to the set of thread interleavings that are equivalent under the *reads-from equivalence* [3, 10] (or under Mazurkiewicz equivalence if we extend the graphs to also record the coherence order).

Algorithm 1 shows the general structure of a DPOR algorithm. The procedure VERIFY verifies a concurrent program P by starting from the graph $G_\emptyset$ containing only the initialization events and an empty *environment* $\Gamma_\emptyset$ (Line 2), and exploring the executions of P one by one by calling VISITONE (Line 4). VISITONE does most of the exploration work: it explores one full execution of P and populates Γ with alternative exploration options. These exploration options recorded in Γ are later explored by VERIFY (Line 5).

At each step, VISITONE extends the current execution G by one event a (obtained via $\mathsf{next}_P(G)$), as long as G remains consistent according to the memory model (Line 7). If there

are no more events to add, then G is complete, and VISITONE returns. If a denotes an error (e.g., an assertion violation), it is reported to the user and verification terminates (Line 9).

If a is a read, then it must read from some write in G. To this end, VISITONE calculates the set of all writes in G on the same location as a (Line 11), and chooses one write w_0 as the reads-from option for a (Line 12). For all other same-location writes, an alternative execution is added to Γ so that it can be explored later by VERIFY (Line 13).

If a is a write, it needs to revisit existing reads of the same location in G, because a was not present in the graph when VISITONE was considering possible reads-from options for these reads. To that end, VISITONE calls CALCREVISITS (Line 15), which extends Γ with such alternative explorations. Since the discussion on how these explorations are calculated is not relevant for this paper, we do not present it here; we refer interested readers to Kokologiannakis et al. [18], where CALCREVISITS is explained in detail.

Note that Algorithm 1 does not have any special treatment for **assume** statements. Whenever $\mathsf{next}_P(G)$ encounters an **assume** statement whose condition is not satisfied, it returns a `blocked` event and stops scheduling that thread thereafter. When VERIFY later pops some graph that does not contain the `blocked` label (e.g., because the graph represents an alternative exploration choice before the blocked event), the thread will be again schedulable, and other options that might not block the **assume** will be considered.

III. BOUNDING EFFECT-FREE SPINLOOPS

Effect-free loop iterations that do not exit the loop are almost unobservable: they do not affect the set of reachable program states, and so can be ignored when verifying safety properties of a program. (We note that for liveness properties, effect-free loop iterations cannot be discarded that simply. An infinite sequence of such effect-free iterations, unless prevented by some fairness assumption about the program's semantics, yields a non-terminating run of the program.)

What remains to be clarified is what exactly constitutes an effect-free loop iteration. Clearly, the iteration should not be writing to a global variable, as otherwise other threads may be able to observe whether the iteration took place or not. Similarly, it should also not be assigning to any local registers that could affect the subsequent execution of the thread itself, i.e., to any variables that are *live* at the header of the loop. Assigning to a dead variable is harmless because, by definition, it does not affect the subsequent execution of the thread, even if technically it might reach a slightly different local state (differing only in the values of dead variables).

We note that spinloops need to be effect-free only along looping paths—they may well have side-effects on paths exiting the loop. This is frequently the case for `CAS`-loops, such as the following implementation of an atomic increment:

```
do
    a := x
    success := CAS(x, a, a + 1)          (CAS-LOOP)
while (¬success)
```

```
while (true)
  h := head
  t := tail
  n := next[h]
  h' := head
  if (h ≠ h') continue
  if (h = t)
      if (n) break
      CAS(tail, t, n)
  else
      b := CAS(head, h, n)
      if (b) break
```

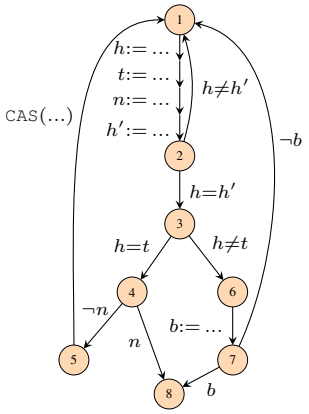


Fig. 3. Simplified dequeue operation from the `ms-queue` benchmark and its CFG, whose instructions are abbreviated. In the code, $head$, $next$, and $tail$ are global variables, while b, h, h', n, and t are local registers.

Here, even though the loop contains a `CAS`, which is generally an effectful instruction, along the looping path, the `CAS` fails, and so the path is effect-free.

We also note that loops often have multiple looping paths, only some of which are effect-free. Consider, for instance, the **while** loop in Fig. 3, which is extracted from the `ms-queue` benchmark of §VIII. It contains three loopy paths. The first (through the **continue** statement) is trivially effect-free because it contains only loads and assignments to dead variables. (All local variables are dead at the loop header.) The second path (when $h = t$) can have side-effects—the `CAS` to $tail$. The third path (when $h \neq t$) is again effect-free because whenever its `CAS` succeeds, the function returns.

Let us now make these intuitions more formal. A path π is *pure* if it either contains no store instructions or, if it contains any, all of them are failed `CAS`es. That is, whenever $\pi(i)$ is a store instruction, then it is of the form $r := \mathtt{CAS}(x, e_1, e_2)$ and there is $i < j < |\pi|$ such that $\pi(j) = \mathbf{assume}(\neg r)$ and for all $i < k < j$, $\pi(k)$ does not assign to r.

Pure paths do not affect the global state, but can affect the local state. A loopy path does not affect the local state if it always reaches the same local state it started from. A simple approximation to reaching the same state is for the path to not assign to any variable that is live at its header. Putting these conditions together, an *effect-free spinloop* is a pure loopy path that does not assign to any variable live at its header. Formally:

Definition 3. *A CFG edge $n \to h$ is an* effect-free spinloop backedge *if every loopy path of $n \to h$ is pure and assigns only to registers dead at h.*

The *spin-assume transformation* removes all effect-free spinloop backedges from the CFG. Returning to the example in Fig. 1, the edge $2 \to 1$ is an effect-free spinloop backedge; removing it transforms thread I of LOOP-PEEL into $a := x; \mathbf{assume}(a = 0)$. In contrast, the backedge of thread II ($6 \to 5$) is not effect-free and so the spin-assume transformation does not affect thread II.

IV. Detecting More Kinds of Spinloops

While the spin-assume transformation defined in the previous section can detect typical cases of **do-while** spinloops, it does not apply to **while** loops that have a non-trivial condition.

The main problem is that the registers used to evaluate the condition are live at the loop header, and so any loop iterations that update these registers are deemed effectful. As a simple example, consider the spinloop of thread II of LOOP-PEEL from §I: register b is live at the beginning of the loop, and so the body of the loop ($b := x$) is effectful. (Formally, in the CFG of Fig. 1, register b is live at node 5—the loop header.)

One simple way to resolve this problem is to apply a compiler transformation called *loop rotation*, which moves the loop exit checks to the end of the loop. Applying loop rotation transforms the second thread of LOOP-PEEL as follows:

$$\begin{array}{ll} b := x & b := x \\ \textbf{while}\ (b \neq 0) \quad \rightsquigarrow & \textbf{if}\ (b \neq 0) \\ \quad b := x & \quad \textbf{do}\ \ b := x\ \textbf{while}\ (b \neq 0) \end{array}$$

The transformed loop can be bounded with the spin-assume transformation yielding executions with at most two loads of x. We note that this bounding outcome is suboptimal, since thread I of LOOP-PEEL is bounded with a single load of x.

A better approach for this example is to exploit *bisimilarity* among CFG nodes. Two nodes are bisimilar if they produce the exact same computations, i.e., if their outgoing edges can be matched 1-to-1 in a way that every two matched edges are labeled with the same instruction and lead to bisimilar nodes. Bisimilarity can be computed as a greatest fixed point, starting with the identity relation (i.e., each node being bisimilar to itself) and adding pairs of nodes whenever they have matching outgoing edges to nodes already calculated to be bisimilar. For example, in Fig. 1, nodes 4 and 6 are bisimilar because they both have only one outgoing edge labeled with the same instruction ($b := x$) and leading to the same node (5).

Having detected that two (distinct) nodes a and b are bisimilar, we can then merge them into one node by redirecting b's incoming edges to a and deleting node b. For example, merging nodes 4 and 6 of Fig. 1 would add an edge from 5 to 4 with label **assume**$(b \neq 0)$, and remove node 6. Effectively, this transformation converts the second thread of LOOP-PEEL to a **do-while** loop analogous to that in its first thread, which makes the spin-assume transformation applicable.

We note that merging bisimilar nodes is not always strictly better than loop rotation. There are cases where loop rotation (or a similar transformation called *jump threading*) can transform a loop into the **do-while** form, but no two distinct bisimilar nodes exist. Such cases frequently arise with **CAS** loops like the following.

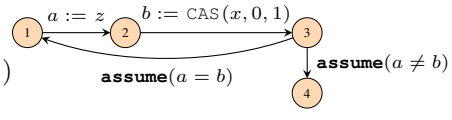
$$\begin{array}{l} \mathit{success} := \mathit{false} \\ \textbf{while}\ (\neg \mathit{success}) \\ \quad a := x \\ \quad \mathit{success} := \texttt{CAS}(x, a, a+1) \end{array} \qquad \text{(CAS-LOOP2)}$$

Here, the spin-assume transformation is not directly applicable to CAS-LOOP2 because *success* is live at the loop header and is updated by the loop body. Loop rotation and/or jump threading, followed by dead assignment elimination, convert this program to CAS-LOOP, which can by handled by the spin-assume transformation. By contrast, merging bisimilar nodes does not change the program, since the program does not contain the same instruction twice.

V. Dynamically Checking Purity

The spin-assume transformation as described in §III uses a completely static definition of purity. If a CAS along a CFG path cannot be determined to always fail, the path is deemed effectful. This is, however, suboptimal for two reasons.

First, using a static purity definition prevents us from transforming paths that are pure only under certain contexts. For instance, consider the thread below, and assume that it is running as part of a program that only writes the value 0 to z (this might not be inferable statically):



In this case, the (only) loopy path of this thread will not be deemed pure (as the CAS is not followed by an **assume**$(\neg b)$ statement), even though it will never produce observable effects in its running context as a will always be 0.

Second, in cases where a loopy path contains a CAS that *does* have observable effects, it is wasteful to explore executions where such a CAS fails. To see this, consider again the dequeue operation of the `ms-queue` example in Fig. 3. As explained in §III, the second loopy path of this operation is not pure, as it potentially has side-effects. Still, it does not make sense to consider iterations where the CAS of this path fails, as they both do not contribute to the loop exiting, and they produce no observable side-effects.

Leveraging the insights above, we say that a CFG backedge $n \to h$ is a *potentially effect-free spinloop backedge* if every loopy path of $n \to h$ assigns only to registers dead at h. The *dynamic-spin-assume transformation* marks all potentially effect-free spinloop backedges with a dynamic purity check. Whenever the $\texttt{next}_P(G)$ function of Algorithm 1 encounters such a check, it validates whether G contains any write event originating from the respective loop iteration and, if not, it returns a `blocked` event, thereby blocking the execution of the respective thread. Otherwise, if the loop iteration did generate a write event, $\texttt{next}_P(G)$ proceeds with the next event.

In fact, the dynamic purity check described above can be relaxed even further: SAVER allows loop iterations to contain write events, as long as these only affect memory locations that are not reachable by other threads. In turn, this proves very useful in cases where some initialization writes need to take place as part of a loop.

To see an example of this, consider the push operation of the treiber-stack benchmark (cf. Fig. 4). First, a node to be inserted to the stack is created, but this node cannot be initialized fully: its *next* field needs to point to the existing

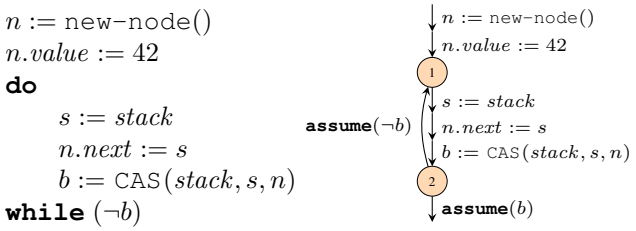


Fig. 4. Simplified push operation from the `treiber-stack` benchmark with its CFG: *stack* is a global variable, while b, n, and s are registers.

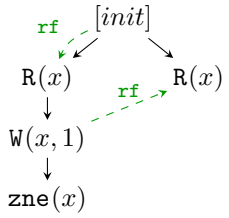


Fig. 5. Execution graph encountered during the exploration of ZNE-OBS.

top of the stack, but the stack top might change between the time it is read, and the time the node is created. Thus, the push operation first reads the stack, sets it as the node's *next*, and then tries to atomically replace the stack with the newly created node. If the replacement succeeds, the operation exits; otherwise, it tries again. Notice, however, that, as long as the replacement CAS does not succeed, the store to the node's *next* remains unobserved by the other threads. Thus, it is safe to consider failed CAS loop iterations as effect-free, and block their exploration.

As a final remark, we observe that validating effect-free loops dynamically makes SAVER resilient to more aggressive loop rotation passes that convert loops to a canonical form containing a single backedge (see §VII).

VI. HANDLING ZERO-NET-EFFECT SPINLOOPS

Let us now consider the more challenging case of *zero-net-effect* (ZNE) loops. Recall that these are spinloop iterations that do have side-effects but (1) whose side-effects cancel each other out, and (2) whose intermediate effects are not observed by other threads. While condition (1) can be checked pretty well statically, condition (2) has to be checked dynamically. In the discussion below, we focus on ZNE loops that arise because of an atomic increment being followed by an atomic decrement of the same location and value.

A decrement instruction at node k is a *canceling decrement* in a loop h if all of h's loopy paths that contain node k also contain a prior opposite increment instruction, and the paths are effect-free modulo two instructions. More formally:

Definition 4. *A node k in a (minimal) CFG cycle with header h is a* canceling decrement *if it has a (unique) outgoing edge of the form $r_1 := \texttt{fetch\_add}(x, n)$, and for every loopy path π of h such that $\pi(i) = k$ for some $1 < i < |\pi|$, there exists $j < i$ such that $\pi(j) = r_2 := \texttt{fetch\_add}(x, -n)$ for some r_2, and replacing the instructions at $\pi(i)$ and $\pi(j)$ with plain assignments to r_1 and r_2 yields an effect-free path.*

SAVER's *spin-zne* transformation annotates all canceling decrements so that when $\texttt{next}_P(G)$ encounters them for the first time (cf. Algorithm 1, Line 7), it generates a $\texttt{zne}(x)$ event and blocks the thread instead of generating a read event and afterwards a write event. The $\texttt{zne}(x)$ event serves as a marker for SAVER to validate that the transformation is sound.

Validation of ZNE loops happens every time a new event e is added to the graph by calling the CHECKZNEVALIDITY routine (Algorithm 1, Line 16). If we use the pair $\langle w, z \rangle$ to represent a blocked ZNE loop iteration with w being the event corresponding to the increment of the ZNE loop and z being the `zne` event, the addition of e can render the reduction of the $\langle w, z \rangle$ loop unsound in one of the following two ways.

First, if e writes to the same location as w, it can be ordered (in coherence) between w and the blocked decrement (after z), and so, unless e is also an atomic increment, w and its corresponding decrement will no longer cancel each other out.

Second, if e reads from w and there is already some other read event reading from w, then, in an alternate execution, it is possible for e to read from the canceling decrement instead of w, thereby observing the value of the shared variable flickering. To see this, consider the example below.

while $(true)$	$b := x$	
$a := \texttt{fetch\_add}(x, 1)$	**if** (b)	(ZNE-OBS)
if $(a = 42)$ **break**	$c := x$	
$\texttt{fetch\_add}(x, -1)$	**assert**(c)	

Note that the loop of the first thread fulfills the conditions of a ZNE loop, and so the second `fetch_add()` will be annotated by the spin-zne transformation.

Figure 5 shows the execution graph arising from adding the events of thread I and then adding the read event corresponding to the $b := x$ instruction of thread II in the case it reads the incremented value of x. Next, we have to add the event corresponding to $c := x$. In this graph, the only consistent option for this event is to also read the incremented value of x, which satisfies the subsequent assertion. Yet, if we had the decrement of x instead of the `zne` event in the graph, c could also have read the value 0 from the decrement, and the **assert** would have failed. Thus, it is clear that concurrent reads can render the transformation of ZNE spinloops unsound.

Therefore, CHECKZNEVALIDITY(G, e) (cf. Algorithm 2) checks whether either of these two conditions holds for any existing $\texttt{zne}(x)$ event in the graph (where x is the location accessed by e), and if so, it removes the `zne` event(s) and unblocks the corresponding thread(s), which will eventually add the missing decrement event(s) and restore soundness.

Other cases of ZNE loops can be handled in a similar manner. For example, consider spinloops containing matching lock acquisitions and releases. In such a case, acquiring the lock acts as the increment operation and releasing the lock as the matching decrement. Statically, it therefore suffices to check that each lock release in the spinloop has its corresponding lock acquisition earlier in the same spinloop iteration.

Algorithm 2 ZNE Spinloop Validity Check

```
1: procedure CHECKZNEVALIDITY(G, e)
2:     if e is a write other than from a fetch_add() then
3:         G.E ← G.E \ zne(location-of(e))
4:     else if e ∈ R_loc(x) ∧ ∃e' ≠ e. G.rf(e') = G.rf(e) then
5:         G.E ← G.E \ matching-zne(G.rf(e))
```

Dynamically, we simply check that no other thread accesses the lock besides by calling the acquire and release methods.

VII. Implementation

We implemented Saver as an extension to the open-source GenMC tool [18, 19]. GenMC is a state-of-the-art stateless model checker for C/C++ programs that works at the level of LLVM Intermediate Representation (LLVM-IR), and can verify programs under weak memory models such as RC11 [21] and IMM [24]. Saver is implemented as (a) a collection of transformation passes that modify GenMC's input before the latter starts the verification procedure, and (b) slight modifications to GenMC's DPOR algorithm that handle the dynamic checks for pure and ZNE loops.

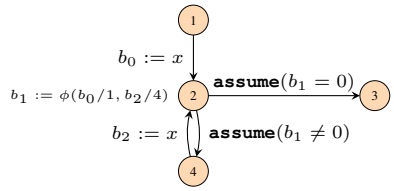
As expected, Saver imposes negligible overhead over GenMC, as its transformations take place statically, before the verification procedure starts, and the dynamic conditions for purity and ZNE loops can be checked in $\mathcal{O}(n)$ time (where n is the size of the graph), which is dominated by GenMC's existing consistency checks.

We conclude this section with some remarks regarding the implementation of loop rotation and the merging of bisimilar nodes over GenMC/LLVM.

In the case of loop rotation, we have implemented our own custom loop rotation pass that applies to loops whose rotation is deemed worthwhile. Although LLVM already contains an implementation of loop rotation, that implementation performs a more aggressive transformation by converting loops to a canonical form containing a single backedge. That is, if the loop contains multiple backedges, it constructs a new node with a backedge to the loop header and redirects all the existing backedges to the new node. This latter transformation is detrimental to the static detection of effect-free paths because it would, for example, conflate the three loopy paths of `ms-queue`'s dequeue operation (Fig. 3), thereby disabling the spin-assume transformation for the two that are effect-free. To avoid this unintended consequence, one would then have to undo this transformation (e.g., by invoking a form of jump threading) or rely on dynamic purity checks (§V). Instead, and to be able to statically transform as many loops are possible, we opted for implementing our own loop rotation pass, that transforms simple loops like CAS-LOOP2; loops that are not captured by our loop rotation pass are handled dynamically.

In the case of merging of bisimilar nodes, there are also a couple of points worth mentioning. First, detecting bisimilar nodes on LLVM is more complicated than what was discussed in §IV because LLVM represents programs in *static single assignment* (SSA) form. The effect of this design choice is that there are never two nodes with identical assignments on their outgoing edges, since by the SSA definition each assignment is to a different register. Therefore, the standard bisimilarity algorithm outlined earlier in this section will not detect any nodes as being bisimilar!

As an example, consider the "SSA-CFG" of thread II of the LOOP-PEEL program from §I, which is shown below.

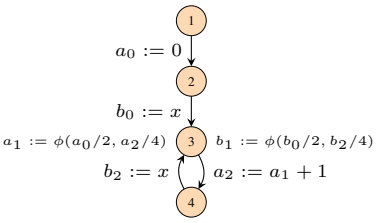


The SSA-CFG is an enriched kind of CFG whose nodes may have ϕ-guards that define a variable differently depending on the incoming control flow path. For instance, in the SSA-CFG above, at node 2, b_1 is defined to be equal to b_0 if node 2 is reached from node 1, or to b_2 if it is reached from node 4.

In order to match nodes 1 and 4, our bisimilarity implementation has not only to account for ϕ-nodes, but also unify the variables b_0 and b_2. It does so by collecting equality constraints and solving them by unification. For each node with more than one incoming edge, the algorithm starts iterating backwards for each pair of predecessors, and collects the constraints under which these predecessors are equal, simplifying them along the way. The iteration stops when some nodes cannot be equal under any constraints, or the entry node has been reached. At that point, any pair of nodes whose constraints can be trivially solved (namely, nodes 1 and 4 above) are deemed bisimilar.

Besides making bisimilarity detection more complex, SSA also affects the merging of bisimilar nodes. Consider the program below along with its SSA-CFG.

```
a := 0
b := x
while (true)
    a := a + 1
    b := x
```



As can be seen, each of the assignments is to a different register, and node 3 contains two ϕ-guards (one for a and one for b) selecting the appropriate register to use depending on the incoming branch. With the algorithm outlined above one can detect that nodes 2 and 4 are bisimilar. However, one cannot simply add an edge $a_2 := a_1 + 1$ from node 3 to node 2 because that would violate the SSA form. To ensure that the resulting CFG is well-formed we also have to introduce a ϕ-guard at node 2 to say which version of a should be used for node 2. Our implementation achieves this by *moving* ϕ-guards the incoming values of which have not been deemed bisimilar (e.g., the ϕ-guard for a here) to the new loop header, along with any other incoming edges these ϕ-guards have.

VIII. Evaluation

In this section, we evaluate the effectiveness of Saver's optimizations on a variety of benchmarks. Our evaluation comprises two distinct parts, with the first part concerning the overall performance of Saver in a real-world setting, and the second part evaluating the effectiveness of employing individual transformations.

In general, we observe that applying the transformations introduced in this paper typically leads to *exponential gains* in real-world benchmarks with spinloops. Key to these gains are Saver's dynamic checks for spinloop purity and/or validity of ZNE spinloops, as well as the bisimilarity-based reduction of CFGs, which enables more spinloops to be bounded.

We conducted all experiments on a system with an Intel(R) Core(TM) i5-6600 CPU (4 cores @ 3.30GHz) and 16GB of RAM, running a custom Debian-based distribution. We used LLVM 7 for GenMC (v0.5.3). All reported times are in seconds. We set the timeout limit to 30 minutes.

A. Overall Performance

We start by evaluating Saver on some challenging data structures utilizing weak-memory atomics that we harvested from the literature, including all data-structure benchmarks from GenMC's original paper [18]. Since we want to measure the effectiveness of Saver's optimizations over the existing GenMC implementation, we do not compare against other tools and use GenMC as a baseline for our comparison. Since GenMC already contains a simple heuristic that converts some very simple **do-while** spinloops into **assume** statements, we use two versions of GenMC: one with its heuristic disabled and one with it enabled.

As can be seen in Table I, these benchmarks demonstrate that Saver is extremely effective in a real-world setting, and that Saver's extensions combined lead to exponential gains. For all these benchmarks apart from mutex-musl, we have used an unroll value of $N + 1$ (where N is the number of threads, shown in parentheses) for both GenMC and Saver to avoid manually unrolling any loops that spawn threads or initialize thread-local variables. For mutex-musl an unroll value of 2 and some manual unrolling was used, to keep the state space manageable. The transformations that Saver applies are shown on the rightmost column, where S, D, Z, L, and B stand for spin-assume, dynamic-spin-assume, zne-assume, loop-rotation, and bisimilarity, respectively.

As can also be seen, GenMC's simple heuristic is of rather limited value. It works very well only for the first two benchmarks (mcslock and qspinlock), where it matches the performance of Saver. For the next three benchmarks (seqlock, mpmc-queue, and linuxrwlocks), it reduces the number of executions explored, but is still much slower than Saver. Specifically, for mpmc-queue(4) and linuxrwlocks(4) GenMC does not manage to terminate within the time limit, while for seqlock(4) it needs 30.71 seconds. For the remaining eight benchmarks, GenMC's heuristic does not apply at all.

Saver, on the other hand, is able to employ its transformations (even if only partially) on all the benchmarks and, with the exception of mutex-musl, this leads to a huge reduction in verification time over GenMC. That is, even if in some cases, Saver only applies spin-assume/zne-assume in some of the data-structure's methods, or even in some paths of a particular method, Saver is still orders of magnitude faster than GenMC. Concretely, for all benchmarks, Saver is able to transform at least one of the spinloops completely into an **assume** statement. For seqlock, Saver reduces the read paths; for mpmc-queue, it reduces both the enqueue and dequeue methods; for linuxrwlocks, the read_lock and write_lock methods, for chase-lev, the steal method; for treiber-stack, the pop method; for mutex, mutex-musl, ttaslock, and twalock, various spinloops in the lock and unlock paths; for ms-queue, the enqueue and dequeue methods; and for scgather the check method. Finally, the smaller gains in verification time for mutex-musl are due to the small unroll value used and the fact that Saver's transformations do not apply to all the benchmark loops.

TABLE I
Real-world benchmarks

	GenMC$_{\setminus S}$	GenMC	Saver		
	Execs	*Execs*	*Execs*	*Time*	*Trans*
mcslock(3)	5964	336	336	0.09	S
mcslock(4)	◷	26 232	26 232	6.20	S
qspinlock(2)	12	6	6	0.02	S
qspinlock(3)	13 764	564	564	0.09	S
seqlock(3)	430	147	9	0.03	S
seqlock(4)	3 670 360	87 980	88	0.21	S
mpmc-queue(3)	1 232 884	15 808	166	0.12	S, D
mpmc-queue(4)	◷	◷	39 706	193.41	S, D
linuxrwlocks(3)	14 059 037	38 033	24	0.04	B, S, Z
linuxrwlocks(4)	◷	◷	1060	0.36	B, S, Z
chase-lev(5)	17 367	17 367	3835	0.20	S
chase-lev(6)	778 581	778 581	41 055	2.39	S
treiber-stack(3)	426	426	18	0.10	S, D
treiber-stack(4)	1 546 168	1 546 168	484	0.61	S, D
mutex(2)	18	18	12	0.09	S, D
mutex(3)	59 760	59 760	7086	0.54	S, D
mutex-musl(2)	34	34	26	0.09	S, D
mutex-musl(3)	652 104	652 104	361 296	28.20	S, D
ttaslock(3)	11 031	11 031	162	0.10	S, D
ttaslock(4)	◷	◷	20 760	2.46	S, D
twalock(3)	1338	1338	96	0.10	S
twalock(4)	1 018 872	1 018 872	6144	0.72	S
ms-queue(3)	1389	1389	75	0.09	L, S, D
ms-queue(4)	◷	◷	10 662	28.13	L, S, D
scgather(3)	7560	7560	90	0.04	Z
scgather(4)	1 247 400	1 247 400	2520	1.07	Z

B. Employing Dynamic Purity/Unobservability Checks

As it can be seen from Table I, in more than half of the benchmarks, Saver checked the purity of a spinloop or the non-observability of its intermediate effects dynamically. Dynamic checking proves useful for three cases.

First, in cases like ms-queue, plain spin-assume is not enough to fully transform some spinloop iterations into

TABLE II
BENEFITS OF BISIMILARITY

	SAVER$_{\setminus B \setminus L}$		SAVER$_{\setminus B}$		SAVER	
	Execs	*Time*	*Execs*	*Time*	*Execs*	*Time*
ws+r-peeled(3)	9 264 697	81.01	5418	0.04	1	0.01
ws+r-peeled(4)	83 357 632	1353.39	13 419	0.18	1	0.01
w+rs-peeled(3)	🕒	🕒	893 025	4.75	1	0.01
w+rs-peeled(4)	🕒	🕒	🕒	🕒	1	0.03

assume statements because they contain possibly succeeding CAS operations. Recall from Fig. 3 that the second loopy path of the simplified dequeue implementation is not effect-free. By adding a dynamic check to the relevant backedge, SAVER only considers iterations where the CAS actually succeeds, thus greatly reducing the state space of the program.

Second, in other cases (e.g., mutex and ttaslock), dynamic-spin-assume is necessary as spinloops contain function calls possibly containing side-effects. As it is difficult to determine statically whether these side-effects will actually take place in the particular calling context, the check is deferred to runtime.

Third, the unobservability checks both for initialization writes in failed CAS loops (e.g., treiber-stack) and for ZNE loops (linuxrwlocks and scgather) are very hard to perform statically with sufficient precision. As such, performing them dynamically is the only viable option.

C. Employing Loop Rotation and Bisimilarity Reduction

Loop rotation and bisimilarity reduction are similarly important in some real-world test cases. Even though they do not yield any performance improvements on their own, they are instrumental in making the spin-assume and zne-assume transformations applicable to more complex cases. Specifically, in benchmarks like ms-queue and linuxrwlocks, spin-assume and zne-assume are not applicable without loop rotation and bisimilarity respectively. And, in fact, these are not the only cases that we have encountered; there are many ways to rewrite the same benchmarks so that they also require bisimilarity and/or loop rotation, thus rendering these transformations a necessity, as opposed to an enhancement.

As a further demonstration of their usefulness, we consider two synthetic test cases inspired by the LOOP-PEEL example. In these tests, some threads repeatedly write to a shared variable, which is read by readers that employ schemes similar to LOOP-PEEL's second thread. As explained in §III, spin-assume is not directly applicable in such cases because the live variables of the header are redefined within the loop. Thus, we used an unroll value of 3, and manually unrolled any loops utilized by the writer threads. For these benchmarks, we used three SAVER versions: the default version that employs both bisimilarity and loop rotation (SAVER), a version where bisimilarity is disabled (SAVER$_{\setminus B}$) and a version where both bisimilarity and loop rotation are disabled (SAVER$_{\setminus B \setminus L}$). The results can be seen in Table II.

With bisimilarity reduction, SAVER transforms the spinloops into **assume** statements and only explores one execution, since only one combination of values satisfies the **assume**s. Applying only loop rotation is equivalent to transforming the syntactic spinloops in these programs into **assume** statement but keeping the peeled iteration. Thus, SAVER$_{\setminus B}$ explores a much larger number of executions, which affects the verification time. Applying neither transformation (SAVER$_{\setminus B \setminus L}$) explores a huge number of executions and often timeouts. These results highlight the necessity of being resilient against small syntactic variations as, even if a single read is not taken into account when transforming a spinloop into an **assume**, the state space might grow exponentially.

IX. RELATED WORK AND CONCLUSIONS

We have presented a set of automated techniques for soundly bounding various kinds of spinloops to a single iteration, which empowers SMC to reason effectively about programs containing such spinloops. Although our contribution was presented in terms of SMC, it should be equally applicable to SAT/SMT-based bounded model checking (BMC) implemented by different tools (e.g., [11, 15, 9]).

Although there is a large body of work on model checking concurrent programs (e.g., [22, 6, 7, 10, 26, 5]), we are not aware of any other automated technique for bounding such a wide range of spinloops including potentially effect-free and ZNE loops. NIDHUGG [1, 4], RCMC [17] and GENMC [18, 19] are the only other tools we are aware of that automatically transform some spinloops to **assume** statements but they limit themselves to very simple busy-wait loops with no side-effects and no CAS instructions and they are not resilient to simple syntactic variations of such loops. POET [25] does recognize spinloop iterations that do not make progress, but saves the program state in order to do so.

Since both SMC and BMC cannot handle programs with executions of unbounded length, most tools bound the number of allowed loop iterations by a user-specified bound. Other tools like CDSCHECKER [23] use a memory-liveness bound to ensure termination for spinloops. As shown in §VIII, bounding techniques in general are inferior to converting spinloops to **assume** statements in terms of scalability.

Bounding of spinloops to a single iteration is, however, not a totally new idea. In a rather different context, Flanagan et al. [13] have used purity for proving atomicity of concurrent libraries treating effect-free spinloops as though they had been reduced to **assume** statements. Elmas et al. [12] have also performed similar transformations in their tool QED, which allows a programmer to initiate a sequence of reductions and abstractions to statically establish correctness of a program.

ACKNOWLEDGMENTS

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

1. Abdulla, P.A., Aronis, S., Atig, M.F., Jonsson, B., Leonardsson, C., Sagonas, K.: Stateless model checking for TSO and PSO. In: TACAS 2015, LNCS, vol. 9035, pp. 353–367. Springer, Heidelberg (2015)
2. Abdulla, P.A., Aronis, S., Jonsson, B., Sagonas, K.: Optimal dynamic partial order reduction. In: POPL 2014, pp. 373–384. ACM, New York, NY, USA (2014)
3. Abdulla, P.A., Atig, M.F., Jonsson, B., Lång, M., Ngo, T.P., Sagonas, K.: Optimal stateless model checking for reads-from equivalence under sequential consistency. Proc. ACM Program. Lang. 3, 150:1–150:29 (2019)
4. Abdulla, P.A., Atig, M.F., Jonsson, B., Leonardsson, C.: Stateless model checking for POWER. In: CAV 2016, LNCS, vol. 9780, pp. 134–156. Springer, Heidelberg (2016)
5. Abdulla, P.A., Atig, M.F., Jonsson, B., Ngo, T.P.: Optimal stateless model checking under the release-acquire semantics. Proc. ACM Program. Lang. 2(OOPSLA), 135:1–135:29 (2018)
6. Albert, E., Arenas, P., de la Banda, M.G., Gómez-Zamalloa, M., Stuckey, P.J.: Context-sensitive dynamic partial order reduction. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017, pp. 526–543. Springer International Publishing, Cham (2017)
7. Albert, E., Gómez-Zamalloa, M., Isabel, M., Rubio, A.: Constrained dynamic partial order reduction. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018, pp. 392–410. Springer International Publishing, Cham (2018)
8. Alglave, J., Maranget, L., Tautschnig, M.: Herding cats: Modelling, simulation, testing, and data mining for weak memory. ACM Trans. Program. Lang. Syst. 36(2), 7:1–7:74 (2014)
9. Burckhardt, S., Alur, R., Martin, M.M.K.: CheckFence: Checking consistency of concurrent data types on relaxed memory models. In: PLDI 2007, pp. 12–21. ACM, New York, NY, USA (2007)
10. Chalupa, M., Chatterjee, K., Pavlogiannis, A., Sinha, N., Vaidya, K.: Data-centric dynamic partial order reduction. Proc. ACM Program. Lang. 2(POPL), 31:1–31:30 (2017)
11. Clarke, E.M., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: TACAS 2004, LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004)
12. Elmas, T., Qadeer, S., Tasiran, S.: A calculus of atomic actions. In: Shao, Z., Pierce, B.C. (eds.) POPL 2009, pp. 2–15. ACM (2009)
13. Flanagan, C., Freund, S.N., Qadeer, S.: Exploiting Purity for Atomicity. IEEE Trans. Software Eng. 31(4), 275–291 (2005)
14. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: POPL 2005, pp. 110–121. ACM, New York, NY, USA (2005)
15. Gavrilenko, N., Ponce-de-León, H., Furbach, F., Heljanko, K., Meyer, R.: BMC for weak memory models: Relation analysis for compact SMT encodings. In: Dillig, I., Tasiran, S. (eds.) CAV 2019, pp. 355–365. Springer International Publishing, Cham (2019)
16. Godefroid, P.: Model checking for programming languages using VeriSoft. In: POPL 1997, pp. 174–186. ACM, Paris, France (1997)
17. Kokologiannakis, M., Lahav, O., Sagonas, K., Vafeiadis, V.: Effective stateless model checking for C/C++ concurrency. Proc. ACM Program. Lang. 2(POPL), 17:1–17:32 (2017)
18. Kokologiannakis, M., Raad, A., Vafeiadis, V.: Model checking for weakly consistent libraries. In: PLDI 2019, ACM, New York, NY, USA (2019)
19. Kokologiannakis, M., Vafeiadis, V.: GenMC: A model checker for weak memory models. In: Silva, A., Leino, K.R.M. (eds.) CAV 2021, LNCS, vol. 12759, pp. 427–440. Springer, Heidelberg (2021)
20. Kokologiannakis, M., Vafeiadis, V.: HMC: Model checking for hardware memory models. In: ASPLOS 2020, ASPLOS '20, pp. 1157–1171. ACM, Lausanne, Switzerland (2020)
21. Lahav, O., Vafeiadis, V., Kang, J., Hur, C.-K., Dreyer, D.: Repairing sequential consistency in C/C++11. In: PLDI 2017, pp. 618–632. ACM, Barcelona, Spain (2017)
22. Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P.A., Neamtiu, I.: Finding and reproducing Heisenbugs in concurrent programs. In: OSDI 2008, pp. 267–280. USENIX Association (2008)
23. Norris, B., Demsky, B.: CDSChecker: Checking concurrent data structures written with C/C++ atomics. In: OOPSLA 2013, pp. 131–150. ACM (2013)
24. Podkopaev, A., Lahav, O., Vafeiadis, V.: Bridging the gap between programming languages and hardware weak memory models. Proc. ACM Program. Lang. 3(POPL), 69:1–69:31 (2019)
25. Rodríguez, C., Sousa, M., Sharma, S., Kroening, D.: Unfolding-based Partial Order Reduction. In: CONCUR 2015, LIPIcs, pp. 456–469. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2015)
26. Zhang, N., Kusano, M., Wang, C.: Dynamic partial order reduction for relaxed memory models. In: PLDI 2015, pp. 250–259. ACM, New York, NY, USA (2015)