Awaiting for Godot: Stateless Model Checking that Avoids Executions where Nothing Happens

Bengt Jonsson
Uppsala University, Sweden
Email: bengt@it.uu.se

Magnus Lång
Uppsala University, Sweden
Email: magnus.lang@it.uu.se

Konstantinos Sagonas
Uppsala University, Sweden and NTUA, Greece
Email: kostis@it.uu.se

Abstract—**Stateless Model Checking (SMC) is a verification technique for concurrent programs that checks for safety violations by exploring all possible thread schedulings. It is highly effective when coupled with Dynamic Partial Order Reduction (DPOR), which introduces an equivalence on schedulings and need explore only one in each equivalence class. Even with DPOR, SMC often spends unnecessary effort in exploring loop iterations that are *pure*, i.e., have no effect on the program state. We present techniques for making SMC with DPOR more effective on programs with pure loop iterations. The first is a static program analysis to detect loop purity and an associated program transformation, called *Partial Loop Purity Elimination*, that inserts `assume` statements to block pure loop iterations. Subsequently, some of these `assumes` are turned into `await` statements that completely remove many `assume`-blocked executions. Finally, we present an extension of the standard DPOR equivalence, obtained by weakening the conflict relation between events. All these techniques are incorporated into a new DPOR algorithm, OPTIMAL-DPOR-AWAIT, which can handle both `awaits` and the weaker conflict relation, is optimal in the sense that it explores exactly one execution in each equivalence class, and can also diagnose livelocks. Our implementation in NIDHUGG shows that these techniques can significantly speed up the analysis of concurrent programs that are currently challenging for SMC tools, both for exploring their complete set of interleavings, but even for detecting concurrency errors in them.**

I. INTRODUCTION

Ensuring correctness of concurrent programs is difficult, since one must consider all the different ways in which actions of different threads can be interleaved. Stateless model checking (SMC) [9] is a fully automatic technique for finding concurrency bugs (i.e., defects that arise only under some thread schedulings) and for verifying their absence. Given a terminating program and fixed input data, SMC systematically explores the set of all thread schedulings that are possible during program runs. A special runtime scheduler drives the SMC exploration by making decisions on scheduling whenever such choices may affect the interaction between threads. SMC has been implemented in many tools (e.g., VeriSoft [10], CHESS [20], Concuerror [6], NIDHUGG [2], rInspect [24], CDSCHECKER [21], RCMC [14], and GENMC [18]), and successfully applied to realistic programs (e.g., [11] and [17]).

SMC tools typically employ *dynamic partial order reduction* (DPOR) [8, 1] to reduce the number of explored schedulings. DPOR defines an equivalence relation on executions, which preserves relevant correctness properties, such as reachability of local states and assertion violations. For correctness, DPOR needs to explore at least one execution in each equivalence
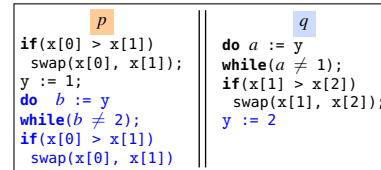

Figure 1: A concurrent program implementing a sorting network. p sorts `x[0]` and `x[1]`, and then uses y to signal that `x[1]` is ready. q waits for y to be 1 and then sorts `x[1]` and `x[2]`, completing one round of bubble sort. In the second round, shown in blue, q signals that the next "generation" of `x[1]` is ready by setting y to 2, upon which p finishes the sort by sorting `x[0]` and `x[1]` again. Initially y = 0.

class. We call a DPOR algorithm *optimal* if it guarantees the exploration of exactly one execution per equivalence class.

In SMC, loops have to be bounded if they do not already terminate in a bounded number of iterations. Loop bounding may in general not preserve assertion failures. Hence a fairly large loop bound should be used, but this is often practically infeasible, and thus loop bounding must strike a balance between these two concerns. However, for loops whose execution has no global effects, the number of equivalence classes that need be explored by SMC can be significantly reduced while still preserving correctness properties, using techniques that we will present in this paper.

Consider the first round of the program snippet in Fig. 1 (shown in black), where thread q executes a loop that waits for thread p to set the shared variable y to 1. A naïve application of SMC with DPOR will explore an unbounded number of executions, since (in the absence of loop bounding) there is an infinite number of equivalence classes, one for each number of performed loop iterations. All iterations of this loop, however, are *pure*, i.e., they have no effect on the program state. For such loops, a bound of one will preserve correctness properties. In our example, the **do-while** loop of thread q can be rewritten into the sequence of statements a := y; **assume**($a = 1$), which will cause the SMC exploration to permanently block thread q whenever the condition of the **assume** is violated.

Using **assume** statements to bound loops causes executions where the condition of the **assume** is violated and its corresponding thread is blocked to be explored. This happens even if the condition will eventually be satisfied, and the original loop will exit, under any fair thread scheduling. *Assume-blocking* of a thread can occur in many contexts, each generating an execution that need not be explored. (We will shortly see this for the example in Fig. 1.) Furthermore, and perhaps more

seriously, this use of **assume**s prevents SMC from diagnosing livelocks in which the loop never exits even under fair thread scheduling. This is because a blocked execution corresponding to a livelock can also result from a spurious execution in which the **assume** reads a shared variable before it has been written to by another thread.

Here is where **await** statements can lead to further reductions. An **await** loads from a shared variable, but only if the loaded value satisfies some condition, otherwise it blocks. In contrast to assume-blocking, *await-blocking* is not permanent but can be repealed if the condition is later satisfied. Thereby, executions where blocking occurs by reading "too early" are avoided. Moreover, such executions can be distinguished from livelocks, in which the condition is not satisfied after some bounded time. For our example, the rewrite of the **do-while** loop into an **await**$(y = 1)$ statement results in a program for which SMC would explore only a single execution in which the **await** reads the value written by thread p.

Consider now the full program in Fig. 1, which performs a concurrent sort of a three-element array using a sorting network. This program can be scaled to larger arrays for increased available parallelism. Since any network sorting an array of size n will have at least $\Omega(n \log n)$ occurrences of a code snippet which exchanges two values after exiting a spinloop, exploring such a program with SMC will explore $\Omega(2^{n \log n})$ executions, even after rewriting the spinloops using **assume** statements. On the other hand, when using **await** statements, all executions fall into the same equivalence class. Thus, an optimal SMC algorithm that can properly handle **await**s will explore only one execution, thereby achieving exponential reduction.

In this paper, we present techniques to (i) automatically transform a program to an intermediate representation that uses **await** as a primitive, and (ii) explore its executions using a provably optimal DPOR algorithm that is **await** aware and also uses a conflict relation between statements which is weaker than the standard one. We first present a static program analysis technique to detect pure loop executions and an associated program transformation, called *Partial Loop Purity (PLP) Elimination*, that inserts **assume** statements which are then turned into **await**s if preceded by the appropriate load. We prove that PLP is sound in the sense that it preserves relevant correctness properties, including local state reachability and assertion failures. We also present and prove conditions under which PLP is guaranteed to remove all pure executions of a loop. Finally, we prove that our new DPOR algorithm OPTIMAL-DPOR-AWAIT, which is an extension of the Optimal-DPOR algorithm of Abdulla et al. [1, 3], is correct and optimal, also with respect to our weaker conflict relation.

All these techniques are available in NIDHUGG, a state-of-the-art SMC tool, and in the paper's replication package [13]. Our evaluation, using multi-threaded programs which are currently challenging for most tools, shows that our techniques can achieve significant (and sometimes exponential) reduction in the total number of executions that need to be explored. Moreover, they enable detection of concurrency bugs which were previously out-of-reach for most concurrency testing tools.
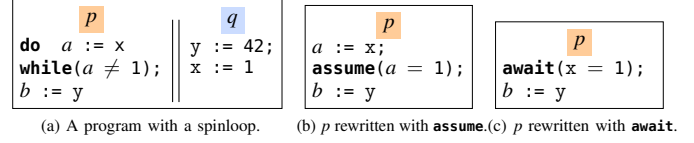

(a) A program with a spinloop.   (b) p rewritten with **assume**. (c) p rewritten with **await**.

Figure 2: Multi-threaded program illustrating the rewrites; initially, $x = y = 0$. For (b) and (c), q is the same as in (a).

II. ILLUSTRATION THROUGH EXAMPLES

In this section, we illustrate our contributions through examples. First, in §II-A we show how **assume** and **await** statements are inserted. In §II-B we illustrate how our optimal DPOR algorithm handles **await** statements, and in §II-C how it handles the weaker conflict relation in which atomic fetch-and-adds on the same variable are not conflicting.

We consider programs consisting of a finite set of threads that share a finite set of shared variables (x, y, z). A thread has a finite set of local registers (a, b, c), and runs a deterministic code, built from expressions, atomic statements, and synchronisation operations, using standard control flow constructs. Atomic statements read or write to shared variables and local registers, including atomic read-modify-write operations, such as compare-and-swap and fetch-and-add. Synchronisation operations include locking a mutex and joining another thread. Executions of a program are defined by an interleaving of statements. We use sequential consistency in this paper, but we note that some weak memory models (e.g., TSO and PSO) can be modelled by an interleaving-based semantics, so our work can be extended to DPOR algorithms [2] that handle such memory models. Our loop transformations introduce **await** statements, that take a conditional expression over a global variable as a parameter and come in several forms: simple awaits (**await**(x = 0)), load-await (a := **await**(x = 0)), and exchange-await (a := **xchgawait**(x = 0, := 1)). These operations block until their condition is satisfied.

A. Introducing Await Statements

Let us show an example of how loops are transformed by introducing **assume** and **await** statements. Consider the loop in Fig. 2a. There, thread p executes a spinloop, waiting for thread q to set the shared variable x. Each iteration of this loop, in which the value loaded into a is different from 1, is pure, i.e., it does not modify shared variables, nor any local register that may be used after the end of the loop. Therefore an **assume** statement is introduced at the point where the thread can distinguish pure executions from impure ones, i.e., after a has been loaded. The result of such a rewrite is shown in Fig. 2b. This program has two traces, one in which the **assume** succeeds, representing the executions in which the original loop terminates, and one where thread p gets *assume-blocked*. The latter trace will exist even in the case where the original loop is guaranteed to terminate under a fair scheduler. This problem is remedied by replacing the load into a and the following **assume** statement by an **await** with a test on the shared variable from which a reads. Such a rewrite results in the program in Fig. 2c. In this case, the **await** statement may permanently block only
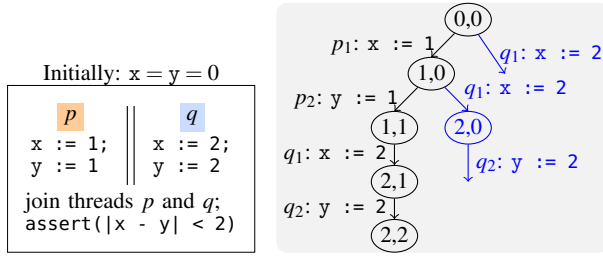
Figure 3: Program with a correctness assertion, and execution trees with the first scheduling of the program; nodes show the values of variables x and y.
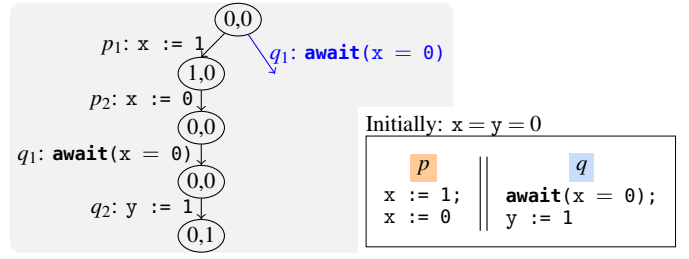

Figure 4: Exploration of a program with an **await** with two satisfying writes.

Figure 5: Exploration of a program with fetch-and-adds. Initially, $x = y = 0$.

if the original loop can livelock under fair scheduling. In our simple example, the rewritten program has only a single trace, since the original loop is guaranteed to terminate and can be replaced by the **await**. Programs with more complex loops (e.g., loops that are pure only along a subset of their paths) are also handled by our program transformation (§III), but the loop is not eliminated when **assume**s or **await**s are introduced.

B. OPTIMAL-DPOR-AWAIT *by Example*

DPOR algorithms are based on regarding executions as equivalent if they induce the same ordering between executions of conflicting statements. The standard conflict relation regards two accesses to the same variable as conflicting if at least one is a write. We begin by illustrating the Optimal-DPOR algorithm [3] on the simple program in Fig. 3. There two threads, p and q, write to two shared variables x and y in sequence. Optimal-DPOR starts by exploring an arbitrary interleaved execution of the program. Assume it is $p_1 \cdot p_2 \cdot q_1 \cdot q_2$ as shown in Fig. 3 (we will denote executions by sequences of thread identifiers, possibly subscripted by sequence numbers). Each explored execution is then analysed to find *races*, i.e., pairs of conflicting events that are adjacent in the happens-before order induced by the conflict relation. (An *event* is a particular execution step of a thread in an execution.) Our first execution contains two races, (p_1, q_1) and (p_2, q_2). For each race, Optimal-DPOR creates a so-called *wakeup sequence*, i.e., a sequence which continues the analysed execution up to the first event in a way which reaches the second event instead of the first event. For the first race, the wakeup sequence is q_1, and for the second race, it is $p_1 \cdot q_1 \cdot q_2$. The wakeup sequences are inserted as new branches just before the first event of the corresponding race, thereby gradually building a tree consisting of the explored executions and added wakeup sequences. The execution tree after the first execution is shown in Fig. 3.

After processing the first execution, Optimal-DPOR then picks the leftmost unexplored leaf in the tree, and extends it arbitrarily to a full execution, in which races are analysed, etc. As the algorithm backtracks, it deletes the nodes it backtracks from in the execution tree. The second execution has two races, (p_1, q_1) as well as (p_2, q_2). However, the corresponding wakeup sequences will result in executions that are redundant, i.e., equivalent to already inserted ones, so no further insertion takes place. The algorithm proceeds in this way until there are no more unexplored leafs corresponding to wakeup sequences. In total, there are four executions explored by Optimal-DPOR, corresponding to the four possible final valuations of x and y.
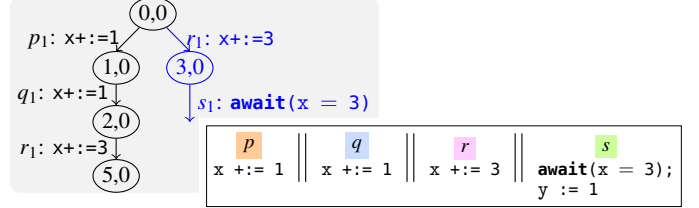
Let us now look at how OPTIMAL-DPOR-AWAIT extends Optimal-DPOR to work for programs with **await**s. Consider the program in Fig. 4. There, p writes to the global variable x, first updating it to 1, and then back to 0. Assume that the first execution is $p_1 \cdot p_2 \cdot q_1 \cdot q_2$. The analysis of races performed by Optimal-DPOR must now be extended to consider that **await** statements are sometimes blocked. First, the conflict between p_2 with q_1 will not be handled like a race, since q_1 is blocked just before p_2. Therefore, we find the closest preceding point in the execution at which q_1 is not blocked, which in this case is at the beginning. We then construct the wakeup sequence q_1 and insert it at the beginning; cf. Fig. 4. Since this program only has two traces, OPTIMAL-DPOR-AWAIT will terminate after exploring the second execution.

C. *Handling Atomic Fetch-and-Add Instructions in DPOR*

To reduce the number of equivalence classes that need be explored by a DPOR algorithm, one can weaken the standard conflict relation between statements by considering two atomic fetch-and-add (FAA) statements on the same variable as non-conflicting if the loaded values are afterwards unused. In the absence of **await** statements, many existing DPOR algorithms like Optimal-DPOR handle this definition without modification. However, this weakening has a subtle interaction with **await** statements that must be handled by OPTIMAL-DPOR-AWAIT.

Consider the program in Fig. 5. In this program, three threads, p, q, and r, add atomically to the shared variable x, and a thread s awaits x having the value 3. We assume that DPOR considers the FAA statements p_1, q_1, and r_1 to be non-conflicting, but conflicting with the statement s_1, should it execute.

Assume that the first explored execution is $p_1 \cdot q_1 \cdot r_1$. From this point, we cannot substitute s_1 for either of p_1, q_1, or r_1, as s_1 is not enabled after any of $q_1 \cdot r_1$, $p_1 \cdot r_1$ or $p_1 \cdot q_1$, respectively. Yet, there is another execution in which s_1 is enabled. In order to construct this execution, we must not only schedule s_1 before one of the other events, but before two, both of p_1 and q_1, so that only r_1 remains. Then, we could construct the wakeup sequence $r_1 \cdot s_1$. In general, OPTIMAL-DPOR-AWAIT

may need to reorder the sequence of independent FAAs that precede an `await` statement and select a subsequence of them, in order to unblock the `await` statement. This can be done in several ways, and OPTIMAL-DPOR-AWAIT is optimised to avoid enumerating all of them. In §IV-B, we will see how.

III. PARTIAL LOOP PURITY ELIMINATION

In this section, we describe *Partial Loop Purity Elimination*, a technique that prevents SMC from exploring executions with pure loop iterations. It consists of (1) a static analysis technique which annotates programs with conditions under which a loop will execute a pure iteration, and (2) a program transformation which inserts `assume` statements based on the analysis.

We consider loops consisting of a set of basic blocks, with a single header block. Each basic block contains a sequence of program statements. Blocks are connected via edges, labelled by conditions. We also consider program representations on Static Single Assignment (SSA) form, which means that each register is assigned by exactly one statement. Thus, a register uniquely identifies the statement that assigns to it. When the value of a register in one block depends on which predecessor block was executed, this is expressed using a *phi node*. For example, in a block C with predecessors A and B containing registers a and b, respectively, the statement $c := \phi(A : a, B : b)$ defines the register c to get the value of a when the previous basic block was A and of b when the previous block was B.

An execution of a loop iteration is *pure* if the execution starts and ends at the header of the loop, and during the iteration (i) no modification of a global variable is performed, (ii) nor of any local variable that may be used after the end of the iteration, and (iii) no *internal* (not to the header) backedge is taken. In SSA form, modification of local variables can be inferred from the phi nodes in the header. If such a phi node uses a different value on the backedge to the header than when first entering, then the loop iteration modified a local variable that is used on some path after the iteration, and we call the header *impure* along the backedge. Our definition considers executions that complete inner loop iterations to be non-pure. However, our PLP transformation will block inner loops from completing pure iterations.

A register a *reaches* a program point l if all paths to l pass a's definition. During a loop execution, we say that an expression over registers is *defined-true* at some program point l in the loop, if the expression evaluates to true under (i) the current valuation of registers that were assigned either outside the loop or during the current loop iteration, and (ii) any valuation of all other registers. We now define a central concept; that of the Forward Purity Condition.

Definition 1 (Forward Purity Condition). *Let l be a program point in a loop. Then, a* Forward Purity Condition *(FPC) at l is an expression in Disjunctive Normal Form over the registers such that if an execution, without leaving the loop or taking an internal backedge, proceeds to a program point l', at which the expression is defined-true, then*

(i) the execution from l' will reach the loop header without taking an internal backedge, and

(a) A loop with non-purity and conditional branches.

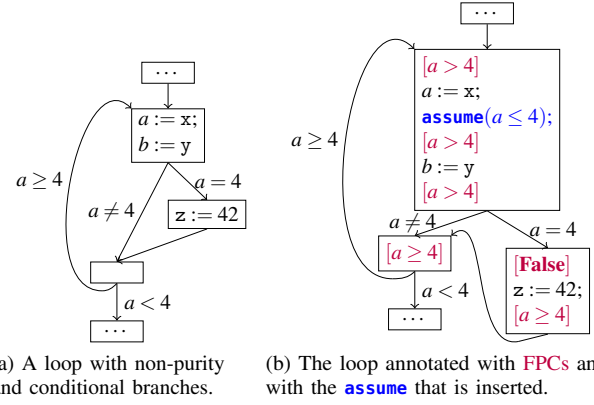(b) The loop annotated with FPCs and with the `assume` that is inserted.

Figure 6: Program snippet illustrating the concepts of the PLP transformation.

(ii) the execution from l to the loop header will not modify any global variables nor any local variable that may be used after execution has reached the loop header. □

We will denote a FPC with brackets, for example $[c > 42]$ or [**False**]. A *purity condition* (PC) of a loop is a FPC of the loop at the beginning of its header. Thus, whenever a loop iteration passes a program point where the PC is defined-true, and has not taken an internal backedge, then that iteration is pure.

We illustrate these concepts for the program snippet in Fig. 6a. In it, the loop loads x and y into registers a and b, then branches on the value of a, and along the path where $a = 4$, there is a write to z. Since a write to a global variable is non-pure, the loop is not pure whenever $a = 4$. The two paths converge in a common block where a loop condition ($a \geq 4$) is checked. This loop is pure if (i) it takes the backedge, i.e., $a \geq 4$ holds, and (ii) the write to z is not performed, i.e., $a \neq 4$ also holds. The conjunction of these conditions, $a > 4$, becomes a purity condition for the entire loop. We thereafter insert an `assume` with the negation of a disjunct of the PC at the earliest point that it is defined-true, i.e., after the load of x, shown in blue in Fig. 6b.

Let us now describe the analysis stage for computing purity conditions. Its first step is to compute FPCs at all points in the loop. Intuitively, the FPC at a point l is a disjunction $c_1 \vee \cdots \vee c_n$, where each c_i is a (forward) path condition for reaching the header via a pure execution from l. We compute FPCs by backwards propagation through statements and basic blocks. Let FPC($s\bullet$) be the FPC immediately after statement s, let FPC($\bullet s$) be the FPC immediately before statement s, let FPC($\bullet B$) be the FPC at the beginning of block B, and let FPC($B\bullet$) be the FPC at the end of block B.

For each statement s, we compute FPC($\bullet s$) as FPC($s\bullet$) $\wedge g$, where g is the condition under which s does not update a global variable. For instance, g is **False** for stores, **True** for loads, $a = 0$ for an atomic add of form x +:= a, $a = b$ for an atomic exchange of form $b := $ xchg(x, a), and $c = 1$ for an atomic compare-exchange of form $c := $ cmpxchg(x, a, b).

FPCs for basic blocks are computed as follows. First, for an edge with condition g from a block A in the loop to a block B, let FPC(A, B) be the FPC along that edge, defined as follows;

- if B is outside the loop, then $\text{FPC}(A,B) = [\textbf{False}]$,
- if B is the header block, then if B is impure along (A,B), then $\text{FPC}(A,B) = [\textbf{False}]$, otherwise $\text{FPC}(A,B) = [g]$.
- if B is inside the loop, then $\text{FPC}(A,B) = [\textbf{False}]$ if the edge from A to B is an internal backedge (A,B), otherwise $\text{FPC}(A,B) = [\text{FPC}(\bullet B) \wedge g]$,

We propagate FPCs backwards through basic blocks by the above rules for statements. We then compute the FPC at the end of a block A with outgoing arcs to $B_1, \ldots, B_k$ as $\text{FPC}(A\bullet) = \bigvee_{i=1}^{k} \text{FPC}(A, B_i)$. We can thereafter calculate FPCs for basic blocks by starting from the edges that leave the loop or go back to its header. Cycles in the control flow graph are no issue, since the FPC of a backedge (A,B) does not depend on B. In Fig. 6b, we can see the FPCs computed by the analysis on the example.

After the analysis, we insert **assume** statements. Given a purity condition of form $c_1 \vee c_2 \vee \cdots \vee c_n$, for each c_i we insert an **assume**$(\neg c_i)$ at the earliest point that is textually after the definitions of all registers in c_i. For registers that do not reach the insertion location, arbitrary values can be used when execution does not pass their definitions. Moreover, if any memory access along the path corresponding to c_i cannot be statically determined not to segfault, we must not insert c_i before that memory access. For this purpose, we associate an optional "earliest insertion point" with every c_i in each FPC computed by the analysis. Finally, to exclude paths that took some internal backedge, a "took internal backedge" boolean register is introduced, computed by phi-nodes, and included in the conjunction c_i.

Theorem 1, whose proof appears in the extended version [12] of this paper, states two essential properties of PLP. These properties intuitively say that PLP removes pure executions while preserving relevant correctness properties. If σ is a local state occurring in a loop $\mathbb{L}$ of a thread p, we say that $\mathbb{L}$ is *unavoidably pure* from σ to denote that whenever thread p is in local state σ during an execution, then p is in the process of completing a pure iteration of $\mathbb{L}$.

Theorem 1. *Let $\mathbb{P}'$ be the program resulting from applying PLP to $\mathbb{P}$. Then $\mathbb{P}'$ satisfies the following properties.*

1) Local State Preservation: *each local state σ of a thread p which is reachable in $\mathbb{P}$ is also reachable in $\mathbb{P}'$, provided no loop of p is unavoidably pure from σ.*
2) Pure Loop Elimination: *no execution of $\mathbb{P}'$ exhibits a completed pure loop iteration of some thread.*

We remark that in the definition of pure loop iterations, we assume possibly conservative characterisations of "global variable" and "local variable that may be used after the end of the iteration" that can be determined by a standard syntactical analysis of the program, and hence used in the PLP analysis.

IV. THE OPTIMAL-DPOR-AWAIT ALGORITHM

In this section, we present OPTIMAL-DPOR-AWAIT, a DPOR algorithm for programs with **await** statements, which is both correct and optimal. Given a terminating program on

given input, it explores exactly one maximal execution in each equivalence class induced by the equivalence relation $\simeq$.

A. Happens-Before Ordering and Equivalence

DPOR algorithms are based on a partial order on the events in each execution. Given an execution E of a program $\mathbb{P}$, an *event* of E is a particular execution step by a single thread; the i'th event by thread p is identified by the tuple $\langle p, i \rangle$, and $\widehat{e}$ denotes the thread p of an event $e = \langle p, i \rangle$. Let $dom(E)$ denote the set of events in E. We define a *happens-before relation* on $dom(E)$, denoted $\xrightarrow{\text{hb}}_E$, as the smallest transitive relation such that $e \xrightarrow{\text{hb}}_E e'$ if e occurs before e' in E, and either

(i) e and e' are performed by the same thread, e spawns the thread which performs e', or e' joins the thread which performs e, or

(ii) e and e' access a common shared variable x, at least one of them writes to x, and they are not both atomic fetch-and-add operations.

Note that the last condition makes atomic fetch-and-add operations on the same shared variable independent. It follows that $\xrightarrow{\text{hb}}_E$ is a partial order on $dom(E)$. We define two executions, E and E', as equivalent, denoted $E \simeq E'$, if they induce the same happens-before relation on the same set of events, (i.e., $dom(E) = dom(E')$ and $\xrightarrow{\text{hb}}_E = \xrightarrow{\text{hb}}_{E'}$). If $E \simeq E'$, then all variables are modified by the same sequence of statements, implying that each thread runs through the same sequence of local states in E and E'.

B. The Working of the OPTIMAL-DPOR-AWAIT Algorithm

OPTIMAL-DPOR-AWAIT is shown in Algorithm 1. It performs a depth-first exploration of executions using the recursive procedure *Explore*(E), where E is the currently explored execution, which can also be interpreted as the stack of the depth-first exploration. In addition, for each prefix E' of E, the algorithm maintains

- a *sleep set sleep*(E'), i.e., a set of threads that should not be explored from E', for the reason that each extension of form $E'.p$ for $p \in sleep(E')$ is equivalent to a previously explored sequence,
- a *wakeup tree wut*(E'), i.e., an ordered tree $\langle B, \prec \rangle$, where B is a prefix-closed set of sequences, whose leaves are called *wakeup sequences*, and $\prec$ is the order in which sequences were added to *wut*(E'). For each $w \in B$ the sequence $E'.w$ will be explored during the call *Explore*(E') in the order given by $\prec$.

All previously explored sequences together with the current wakeup tree (i.e., all sequences of form $E'.w$ for $w \in wut(E')$ and a prefix E' of E) form the current execution tree, denoted $\mathscr{E}$. The branches of $\mathscr{E}$ are ordered by the order in which they were added to the tree. Note that the recursive call to *Explore*(E) may insert into *wut*(E') for prefixes E' of E.

Let $v \setminus p$ denote the sequence v with the first occurrence of an event by thread p (if any) removed. Let $next_{[E]}(p)$ denote the next event performed by thread p after E. Two important concepts are *races* and *weak initials*.

Definition 2 (Non-Blocking Races)**.** *Let e, e' be two events in different threads in an execution E, where e occurs before e'. Then e and e' are in a* non-blocking race*, denoted $e \precsim_E e'$, if (i) e and e' are adjacent in $\xrightarrow{hb}_E$ (i.e., $e \xrightarrow{hb}_E e'$, and for no other event e'' we have $e \xrightarrow{hb}_E e'' \xrightarrow{hb}_E e'$), and (ii) e' cannot be enabled or disabled by an event in another thread.* $\square$

Definition 3 (Weak Initials)**.** *For an execution $E.w$, the set of* weak initials *of w (after E), denoted $WI_{[E]}(w)$, is the set of threads p such that $E.w \simeq E.p.(w \setminus p)$ if p is in w, and $E.w.p \simeq E.p.w$ if p is not in w.* $\square$

Intuitively, $p \in WI_{[E]}(w)$ if $next_{[E]}(p)$ is independent with all events that precede it in w in the case that p is in w, otherwise with all events in w. If $p \in WI_{[E]}(w)$ we say that w is *redundant* wrt. $E.p$, since some extension of $E.w$ is equivalent to some extension of $E.p$. An important property of the execution tree $\mathscr{E}$ that is maintained by the algorithm is that an extension w of an existing sequence E is added only if $\mathscr{E}$ does not contain an execution of form $E'.p$ such that E' but not $E'.p$ is a prefix of E, and $w'.w$ is redundant wrt. $E'.p$, where E' is defined by $E = E'.w'$.

For the OPTIMAL-DPOR-AWAIT algorithm, we define

- $pre(E,e)$ as the prefix of E up to but not including e,
- $notdep(e,E)$ as the subsequence of E of events that occur after e but do not happen-after e.
- $u \precsim_{[E]} w$ to denote that $E.u.v \simeq E.w$ for some v; intuitively u is a "happens-before prefix" of w.

The algorithm runs in two phases: race detection (lines 3–22) and exploration (lines 24–33). Exploration picks the next unexplored leaf of the exploration tree and extends it with arbitrary scheduling to a maximal execution. This leaf is reached step-by-step: at each step, the current execution E is extended by the leftmost child of the root of $wut(E)$ and used in a recursive call to *Explore* (lines 28–31) in order to perform the next step. If $wut(E)$ only contains the empty sequence, an arbitrary thread is chosen for the next step and added to $wut(E)$ (line 26). This step-by-step extension of the current execution is continued until a maximal execution is reached. At each step, the new sleep set after $E.p$ is constructed by taking the elements of $sleep(E)$ that are independent with p. After a recursive call to $E.p$, the subtree rooted at $E.p$ can be removed from the wakeup tree. To remember that we should not attempt to explore any sequences that are redundant wrt. $E.p$, we add p to $sleep(E)$.

The race detection phase is entered when the explored sequence E is maximal. There we examine E for races and construct new non-redundant executions. We distinguish between two types of races: non-blocking races, such as between a write and a read, handled on lines 3–6, and blocking races, such as involving an **await** event, handled on lines 7–22.

For each non-blocking race $e \precsim_E e'$, we let E' be the prefix of E that precedes e, and construct a wakeup sequence v by appending $\widehat{e'}$ to the subsequence of events that occur after e in E but do not happen-after e (line 5). By construction, the sequence $E'.v$ is an execution. Moreover $\widehat{e} \notin WI_{[E']}(v)$ since

---

Algorithm 1: OPTIMAL-DPOR-AWAIT

Initial call: *Explore*($\langle\rangle$) with $wut(\langle\rangle) = \langle \{\langle\rangle\}, \emptyset \rangle$, $sleep(\langle\rangle) = \emptyset$

```
 1  Explore(E)
 2  |  if enabled(E) = ∅ then
 3  |  |  foreach e, e' ∈ dom(E) such that (e ≾_E e') do
 4  |  |  |  let E' = pre(E, e)
 5  |  |  |  let v = (notdep(e, E).ê')
 6  |  |  |  if sleep(E') ∩ WI_{[E']}(v) = ∅ then insert(v, E')
 7  |  |  foreach ⟨e', E'⟩ ∈ ({⟨next_{[E]}(p), E⟩| p is blocked after E}
 8  |  |  |         ∪ {⟨e', pre(E, e')⟩| e' is in E and may block}) do
 9  |  |  |  can-stop := False
10  |  |  |  foreach e in E' (starting from the end)
11  |  |  |          that may enable or disable e'  do
12  |  |  |  |  let E'' = pre(E, e)
13  |  |  |  |  let w = notdep(e, E)
14  |  |  |  |  if e conflicts with all events that may
15  |  |  |  |      enable or disable e' then can-stop := True
16  |  |  |  |  did-insert := False
17  |  |  |  |  foreach maximal subsequence u of w such that
18  |  |  |  |      u ≾_{[E'']} w and e' is enabled after E''.u do
19  |  |  |  |  |  did-insert := True
20  |  |  |  |  |  let v = u.ê'
21  |  |  |  |  |  if sleep(E'') ∩ WI_{[E'']}(v) = ∅  then insert(v, E'')
22  |  |  |  |  if can-stop and did-insert then break
23  |  |  else
24  |  |  |  if wut(E) = ⟨{⟨⟩}, ∅⟩ then
25  |  |  |  |  choose p ∈ enabled(E)
26  |  |  |  |  wut(E) := ⟨{p}, ∅⟩
27  |  |  |  while ∃p ∈ wut(E) do
28  |  |  |  |  let p = min_≺{p ∈ wut(E)}
29  |  |  |  |  sleep(E.p) := {q ∈ sleep(E) | p, q independent after E}
30  |  |  |  |  wut(E.p) := subtree(wut(E), p)
31  |  |  |  |  Explore(E.p)
32  |  |  |  |  add p to sleep(E)
33  |  |  |  |  remove all sequences of form p.w from wut(E)
34  insert(v, E')
35  |  u := ⟨⟩
36  |  let c be the list of children of u in wut(E') from left to right
37  |  foreach sequence u.p in c do
38  |  |  if p ∈ WI_{[E'.u]}(v) then
39  |  |  |  if p ∉ v or (v := v \ p) = ⟨⟩ then return
40  |  |  |  u := u.p
41  |  |  |  if u is a leaf of wut(E') then return
42  |  |  |  goto line 36
43  |  add v as a new rightmost descendant of u in wut(E')
44  |  return
```

---

the occurrence of e' in v does not happen-after e. Thus, v is non-redundant wrt. $E'.\widehat{e}$. If v is also non-redundant wrt. $E'.p$ for each $p \in sleep(E')$, then v is inserted into the wakeup tree at E', extending $wut(E')$ with a new leaf if necessary.

Races involving events that can be blocked are handled at lines 7–22. For each such event e', we extract the prefix E' that precedes e'. Then, for each e in E' that potentially conflicts with e', we extract the prefix E'' preceding e and the sequence w of events that does not happen-after e. For each maximal happens-before prefix u of w after which e' is enabled, we construct a wakeup sequence v as $u.\widehat{e'}$ (line 20), which is checked for redundancy and possibly inserted into the wakeup tree in the same way as for a nonblocking race. Such prefixes can be enumerated by recursively removing the

suffix of one event that may enable or disable e' at a time, stopping whenever e' is enabled by the current prefix. As an optimisation, implemented by the flags *can-stop* and *did-insert*, once the algorithm has found a wakeup sequence that enables e' before some event that conflicts with every event that may enable or disable e', it needs not consider reversing e' with even earlier events e, as those reversals will be considered in a later recursive call.

The function $insert(v, E)$ for inserting a sequence v into a wakeup tree $wut(E')$ is shown in lines 34–44. Starting from the root, represented by the empty sequence, it traverses $wut(E')$ downwards (the current point being u), always descending (line 40) to the leftmost child $u.p$ such that p is a weak initial of the remainder of v until either (i) arriving at a leaf indicating that v was redundant to begin with and $wut(E')$ can be left unchanged (line 41), (ii) encountering a p which is not in v, or exhausting v (line 39), or (iii) arriving at a node with no child passing the test at line 38, and then adding the remainder of v as a new leaf (line 43), since it was shown to be non-redundant.

Algorithm OPTIMAL-DPOR-AWAIT is correct and optimal in the sense that it explores exactly one maximal execution in each equivalence class, as stated in the following theorem whose proof is in the extended version of this paper [12].

Theorem 2. *For a terminating program* $\mathbb{P}$*,* OPTIMAL-DPOR-AWAIT *has the properties that (i) for each maximal execution E of $\mathbb{P}$, it explores some execution E' with $E' \simeq E$, and (ii) it never explores two different but equivalent maximal executions.*

V. IMPLEMENTATION AND EVALUATION

We have implemented our techniques on top of the NIDHUGG tool. NIDHUGG is a state-of-the-art stateless model checker for C/C++ programs with Pthreads, which works at the level of LLVM Intermediate Representation (IR), typically produced by the Clang compiler. We have added our PLP analysis and transformations, as well as the rewrite from load-assume, exchange-assume, and compare-exchange-assume pairs into load-await and exchange-await, as passes over LLVM IR. NIDHUGG comes with a selection of SMC algorithms. One of them is Optimal-DPOR, which we have used as a basis for our implementation of OPTIMAL-DPOR-AWAIT including IFAA, the optimisation of treating fetch-and-add instructions to the same memory location as independent. All the techniques in this paper are now included in upstream NIDHUGG and are enabled when giving the `-optimal` flag.

A. Overall Performance

First, we evaluate our technique and compare its performance against baseline NIDHUGG and the SAVER [16] technique, implemented in a recent version of GENMC [18]. SAVER has a similar goal to our PLP transformation, but tries to identify pure loop iterations dynamically, aborting threads if they perform a pure loop iteration. SAVER's approach does not allow further rewrite with **await**s.

For our evaluation, we used a set of real-world benchmarks similar to those used by the SAVER [16] paper. We note that all atomic memory accesses in these benchmarks have been converted to SC, as this is the only common memory model that both tools support. Where relevant, benchmarks are ran with the same loop bound as in the SAVER paper. For most benchmarks, this is one greater than the number of threads. After the benchmark name, the number of threads are shown in parentheses. Benchmarks mcslock, qspinlock and seqlock are tests of data structures from the Linux kernel. Benchmarks ttaslock and twalock are mockups based on, but not the same as, the benchmarks in the SAVER paper, because its authors were not at liberty to share the original benchmark sources. Both are tests of locking algorithms. Benchmark mpmc-queue tests a multiproducer-multiconsumer queue algorithm, linuxrwlocks tests a readers-writers lock algorithm, treiber-stack tests a lock-free stack algorithm, and ms-queue tests a lock-free queue. Benchmarks mutex and mutex-musl test two mutex algorithms, the second one used in the musl C standard library implementation. Benchmark sortnet is an extended version of the concurrent sort program from Fig. 1. In this version, the sorting networks are generated using Batcher's odd-even mergesort. The number of elements sorted is twice the number of threads, so sortnet(6) sorts 12 elements. In our replication package [13], all the tools and benchmarks are provided, as well as scripts that can replicate the tables in this section.

We evaluate all techniques based on the number of executions they explore. In fact, we show this number using an addition of form $T + B$, where T is the number of explored completed executions and B is the number of executions that are blocked in the sense that either an **await** is deadlocked or some thread is blocked for executing **assume**(false) (in NIDHUGG) or a pure loop iteration (in SAVER). We remark that the SAVER paper reports only the T part, but, as we will see, often the number of blocked executions is significant and outnumbers the number of explored completed executions. Obviously, both numbers contribute to the time an SMC tool takes to explore these programs. The evaluation was performed on a Ryzen 5950X running a July 2022 Arch Linux system.

In Table I, there are four sets of NIDHUGG columns. Baseline shows the performance of unmodified NIDHUGG/Optimal. The PLP columns shows the performance of using unmodified NIDHUGG/Optimal together with Partial Loop Purity Elimination. Pure loops are bounded with **assume**s. The PLP+Await columns shows the result of PLP and transforming **assume**s into **await**s, where possible. Finally, the . . . +IFAA columns report results from when OPTIMAL-DPOR-AWAIT treats atomic fetch-and-add operations as independent. For the two sets of GENMC columns, the SAVER columns show the performance of GENMC v0.6, which implements the SAVER technique, and the Baseline columns show the performance of GENMC v0.5.3, which does not. The timeout we have used for these benchmarks is 1 hour.

Starting at the top of Table I, qspinlock is a benchmark that does not benefit from SAVER nor PLP, but establishes that the baseline algorithms of both tools are very similar but GENMC is faster. In the next four benchmarks (mcslock, twalock, mutex, and mutex-musl), both PLP and SAVER are ineffective, but **await**s eliminate most of the blocked traces (in mcslock) or

Table I: Number of (complete+blocked) executions explored by algorithms implemented in GENMC and NIDHUGG on a set of challenging benchmarks, as well as the execution time (in seconds) taken. The ⊙ symbol means that the exploration did not finish in 1h, and † means that the tool crashed.

| | GENMC | | | | NIDHUGG | | | | | | | |
| | Baseline | | SAVER | | Baseline | | PLP | | PLP+Await | | ...+IFAA | |
Benchmark	Execs	Time	Execs	Time	Execs	Time	Execs	Time	Execs	Time	Execs	Time
qspinlock(2)	6+2	0.02	6+2	0.02	6+2	0.06	6+2	0.08	6+2	0.08	6+2	0.09
qspinlock(3)	564+462	0.06	564+462	0.06	564+462	0.20	564+462	0.20	564+456	0.21	564+456	0.20
mcslock(3)	336+426	0.09	336+426	0.09	336+426	0.20	336+426	0.23	336+72	0.18	336+72	0.18
mcslock(4)	26232+33432	42.06	26232+33432	3.95	26232+33432	16.59	26232+33432	16.95	26232+4824	9.53	26232+4824	9.43
twalock(3)	96+90	0.02	96+90	0.02	96+90	0.09	96+90	0.09	96	0.08	96	0.08
twalock(4)	6144+7224	0.35	6144+7224	0.36	6144+7224	1.40	6144+7224	1.45	6144	0.80	6144	0.81
mutex-musl(2)	20+2	0.02	20+2	0.01	20+2	0.07	20+2	0.07	20	0.06	20	0.07
mutex-musl(3)	136728+12834	4.74	136728+12834	5.03	25146+93000	11.89	25146+93000	12.04	25146+81972	10.90	14736+36846	5.29
mutex(2)	12+2	0.02	12+2	0.02	12+2	0.07	12+2	0.07	12	0.07	10	0.07
mutex(3)	9486+1236	0.35	6582+1188	0.25	9486+1236	1.07	6582+1188	0.84	6582+336	0.76	3618+312	0.44
ms-queue(3)	925+350	0.13	75+284	0.06	901+374	0.58	901+374	0.58	901+374	0.59	901+374	0.60
ms-queue(4)	11696504+8399226	2388.57	10662+192438	18.35	⊙	⊙	⊙	⊙	⊙	⊙	⊙	⊙
linuxrwlocks(3)	38033+31993	3.03	24+59	0.02	38033+31993	6.95	38033+31993	7.24	38033	4.36	3840	0.54
linuxrwlocks(4)	⊙	⊙	1060+5518	0.22	⊙	⊙	⊙	⊙	⊙	⊙	⊙	⊙
ttaslock(3)	162+183	0.02	162+183	0.03	162+183	0.10	36+81	0.08	36	0.07	36	0.07
ttaslock(4)	20760+29440	1.34	20760+29440	1.46	20760+29440	4.94	576+2308	0.30	576	0.15	576	0.15
seqlock(3)	147+230	0.04	9+83	0.02	147+230	0.14	9+83	0.10	9+36	0.08	9+36	0.09
seqlock(4)	87980+105123	19.68	88+2805	0.17	87980+104583	41.58	88+2769	0.44	88+729	0.20	88+729	0.20
mpmc-queue(3)	11206+11612	1.35	166+987	0.09	11206+8188	3.35	166+840	0.24	166+517	0.20	76+421	0.17
mpmc-queue(4)	⊙	⊙	39706+1277783	87.18	⊙	⊙	39706+1123234	226.45	39706+360426	88.29	5410+114208	24.15
treiber-stack(3)	426	0.04	274+80	0.04	426	0.16	274+80	0.14	274+60	0.15	274+60	0.15
treiber-stack(4)	1546168+9216	217.44	250088+167916	33.17	1546168+9216	403.58	250088+167916	98.24	250088+90896	87.92	250088+90896	88.20
sortnet(4)	†	†	1+728	0.33	1+312	0.48	1+312	0.45	1	0.08	1	0.08
sortnet(5)	†	†	1+15231	10.87	1+4517	9.38	1+4517	9.47	1	0.08	1	0.08
sortnet(6)	†	†	1+163292	140.83	1+38285	100.18	1+38285	98.82	1	0.08	1	0.08

all of them (in the remaining three). Moreover, we see that IFAA is effective in mutex and mutex-musl, and manages to almost halve the total number of executions explored.

PLP fails to identify the loop purity in ms-queue. The restriction on the form of purity conditions imposed by our implementation in NIDHUGG is underapproximating the purity condition to [**False**]. This demonstrates a downside with doing purity analysis statically, as SAVER never needs to represent purity conditions in order to eliminate pure loop iterations.

In linuxrwlocks, PLP is ineffective, because this benchmark does not contain pure loop iterations as we have defined them. Rather, the loop contains a pair of fetch-and-add and fetch-and-sub that cancel out, which is called a "zero-net-effect" loop in the SAVER paper [16]. These are out of scope for a static analysis, as SAVER has to dynamically undo the elimination if a read appears to have observed the intermediate effect. Despite the lack of PLP, OPTIMAL-DPOR-AWAIT significantly speeds up linuxrwlocks.

In ttaslock, we believe some implementation issue is preventing SAVER from eliminating pure loop iterations. PLP does work, however, and **await**s eliminate all the blocked executions.

In the next three benchmarks (seqlock, mpmc-queue and treiber-stack), PLP discovers the same pure loop iterations as SAVER, and permits a rewrite to **await**s that significantly reduces the search space, even by an order of magnitude for seqlock, and on mpmc-queue IFAA further halves it.

Finally, OPTIMAL-DPOR-AWAIT really shines on sortnet. GENMC cannot take advantage of **await**s, and so has to explore

an exponential number of (assume-blocked) traces, where NIDHUGG can explore the program in just one. Unfortunately, GENMC v0.5.3 crashes on this benchmark, but we believe it would yield the same numbers as SAVER, which also explores a significant number of redundant executions.

B. Effectiveness on SafeStack

Next, we evaluate the ability of OPTIMAL-DPOR-AWAIT to expose difficult-to-find bugs in real-world code bases. The benchmark we will use is called safestack. It was first posted to the CHESS forum, and subsequently included in the SCTBench [23] and SVComp benchmark suites. The original safestack code attempts to implement a lock-free stack but contains an ABA bug which is quite challenging for concurrency testing and SMC tools to find, in the sense that exposing the bug requires at least five context switches. The test harness is also quite big, containing three threads each performing four operations on the stack. Let us refer to this original harness as safestack-444 to indicate that each of its three threads performs four operations (pop, push, pop, push). We will also use shortened versions of this harness: four versions with just two threads, and four versions where each of the three threads performs fewer operations. The smallest harness that exposes the bug is safestack-331.

We first compare the two SMC tools and their algorithms on versions of safestack that do not exhibit the bug and thus require exhaustive exploration of all traces. Table II shows the results. First, notice that the dynamic technique that SAVER implements

Table II: Number of (complete+blocked) executions that SMC algorithms in GENMC and NIDHUGG explore on shortened, bug-free versions of safestack.

Benchmark	GENMC		NIDHUGG			
	Baseline	SAVER	Baseline	PLP	PLP+Await	…+IFAA
safestack-21(2)	119+6	119+6	119+6	34+2	34+1	19+1
safestack-31(2)	928+107	928+107	928+107	103+27	103+25	56+25
safestack-32(2)	7189+296	7189+296	7189+296	1073+27	1073+12	463+12
safestack-33(2)	121334+12652	121334+12652	121334+12652	6434+1636	6434+1584	2600+1160
safestack-211(3)	1267120+325932	995224+325932	1259280+324382	2690+1126	2690+928	962+686
safestack-311(3)	0+286818740	0+275399108	⊙	0+26536	0+24078	0+14960
safestack-321(3)	⊙	⊙	⊙	906529+388117	906529+331337	288057+216830

is completely or mostly ineffective in these programs; compare it to the baseline numbers. In contrast, PLP achieves significant reduction of the set of executions that NIDHUGG explores. Finally, both the transformation of **assume**s to **await**s and the IFAA optimisation are applicable and result in further reductions in the number of explored executions. The number of complete traces is 0 on safestack-311 since the code does not allow popping the last element, so all traces end up with one thread livelocking in pop with the queue containing only one element. For Table II, the timeout used is 10 hours.

With our next and last experiment, using safestack-331, we can evaluate the tools' abilities to expose the bug. Neither GENMC, with or without SAVER, nor baseline NIDHUGG find anything after running for more than 2 000 hours! On the other hand, if we run NIDHUGG with PLP, awaits, and IFAA, it discovers the bug in just 8 minutes, after exploring $2 + 2\,453\,474$ traces. How much of its search space an SMC tool has to search before it encounters a bug can be up to "luck", so to ensure that this result is not due to luck we "fix" the bug by commenting out all the assertions in the benchmark and run NIDHUGG again. This gives us an upper bound on the size of the search space, i.e., how much would need to be searched to find the bug in the worst case, and also provides an indication of how long it might take to verify the program after fixing the bug. On the fixed safestack-331, NIDHUGG terminates in only 24 minutes after exploring $5\,772 + 8\,521\,721$ traces. This demonstrates how the techniques we presented in this paper substantially reduce the search space on safestack, allowing the bug to be found or its absence verified by an exhaustive SMC technique. To our knowledge, no other exhaustive technique has ever been able to discover the bug in safestack.

VI. RELATED WORK

Since SMC tools assume the analysed program to terminate, they must first bound unbounded loops. Several tools [2, 21, 14, 15] have an automatic loop unroller that is parameterised by a chosen loop bound. Several SMC tools, including NIDHUGG [2], RCMC [14] and GENMC [15], transform simple forms of spinloops, such as the one shown in Fig. 2a, to **assume** statements, but only transform simple polling loops that can be recognised syntactically. We are not aware of any tool that transforms loops into **await** statements, meaning existing tools are susceptible to scalability problems for programs like the sorting networks shown in Fig. 1. An SMC technique that can diagnose livelocks of spinloops under fair scheduling is VSYNC [22]. However, to do so it enforces fairness, and cannot bound the loop even with an **assume**, thus exploring many more traces than tools which transform spinloops to **assume**s.

SAVER [16] also aims to block pure loop iterations by introducing **assume** statements. It identifies pure loop iterations dynamically, instead of by static analysis as in our approach. SAVER's approach allows to detect a larger class of pure loop iterations, but it does not allow further rewrite with **await**s. Furthermore, our PLP transformation can block a looping thread at any point in the loop, not just at the back edge. SAVER also employs several smaller program transformations, such as loop rotation and merging of bisimilar control flow graph nodes, that can increase the number of loops that may qualify as pure. These transformations are orthogonal to the detection of pure loop iterations, and could also be used in our framework.

Checking for purity of loop iterations is an idea that has appeared in other contexts, such as to verify atomicity for concurrent data structures [7, 19] and to reduce complexity for model checking them (e.g., [4]).

The Optimal-DPOR algorithm implemented in NIDHUGG, handles mutex locks but not **await** statements. In the journal article of the Optimal-DPOR algorithm [3], principles for handling other blocking statements are presented. Our OPTIMAL-DPOR-AWAIT develops these principles into a practical and efficient algorithm, which we have also implemented in NIDHUGG. As future work, the Optimal-DPOR with Observers [5] algorithm, which allows two statements to only conflict in the presence of a third event, could also be extended (potentially at higher cost) to handle **await**s.

VII. CONCLUDING REMARKS

We have presented techniques for making SMC with DPOR more effective on loops that perform pure iterations, including a static program analysis technique to detect pure loop executions, a program transformation to block and also remove them, a weakening of the standard conflict relation, and an optimal DPOR algorithm which handles the so introduced concepts. We have implemented the techniques in NIDHUGG, showing that they can significantly speed up the analysis of concurrent programs with pure loops, and also detect concurrency errors.

REFERENCES

[1] P. Abdulla, S. Aronis, B. Jonsson, and K. Sagonas, "Optimal dynamic partial order reduction," in *Symposium on Principles of Programming Languages*, ser. POPL 2014. New York, NY, USA: ACM, 2014, pp. 373–384. [Online]. Available: http://doi.acm.org/10.1145/2535838.2535845

[2] P. A. Abdulla, S. Aronis, M. F. Atig, B. Jonsson, C. Leonardsson, and K. Sagonas, "Stateless model checking for TSO and PSO," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS, vol. 9035. Berlin, Heidelberg: Springer, 2015, pp. 353–367. [Online]. Available: http://dx.doi.org/10.1007/978-3-662-46681-0_28

[3] P. A. Abdulla, S. Aronis, B. Jonsson, and K. Sagonas, "Source sets: A foundation for optimal dynamic partial order reduction," *Journal of the ACM*, vol. 64, no. 4, pp. 25:1–25:49, Sep. 2017. [Online]. Available: http://doi.acm.org/10.1145/3073408

[4] P. A. Abdulla, F. Haziza, L. Holík, B. Jonsson, and A. Rezine, "An integrated specification and verification technique for highly concurrent data structures," *Int. J. Softw. Tools Technol. Transf.*, vol. 19, no. 5, pp. 549–563, 2017. [Online]. Available: https://doi.org/10.1007/s10009-016-0415-4

[5] S. Aronis, B. Jonsson, M. Lång, and K. Sagonas, "Optimal dynamic partial order reduction with observers," in *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference*, ser. LNCS, vol. 10806. Cham: Springer, Apr. 2018, pp. 229–248. [Online]. Available: https://doi.org/10.1007/978-3-319-89963-3_14

[6] M. Christakis, A. Gotovos, and K. Sagonas, "Systematic testing for detecting concurrency errors in Erlang programs," in *Sixth IEEE International Conference on Software Testing, Verification and Validation*, ser. ICST 2013. Los Alamitos, CA, USA: IEEE, Mar. 2013, pp. 154–163. [Online]. Available: https://doi.org/10.1109/ICST.2013.50

[7] C. Flanagan, S. Freund, and S. Qadeer, "Exploiting purity for atomicity," *IEEE Trans. Software Eng.*, vol. 31, no. 4, pp. 275–291, Apr. 2005. [Online]. Available: https://doi.org/10.1109/TSE.2005.47

[8] C. Flanagan and P. Godefroid, "Dynamic partial-order reduction for model checking software," in *Principles of Programming Languages, (POPL)*. New York, NY, USA: ACM, Jan. 2005, pp. 110–121. [Online]. Available: http://doi.acm.org/10.1145/1040305.1040315

[9] P. Godefroid, "Model checking for programming languages using VeriSoft," in *Principles of Programming Languages, (POPL)*. New York, NY, USA: ACM Press, Jan. 1997, pp. 174–186. [Online]. Available: http://doi.acm.org/10.1145/263699.263717

[10] ——, "Software model checking: The VeriSoft approach," *Formal Methods in System Design*, vol. 26, no. 2, pp. 77–101, Mar. 2005. [Online]. Available: http://dx.doi.org/10.1007/s10703-005-1489-x

[11] P. Godefroid, R. S. Hanmer, and L. Jagadeesan, "Model checking without a model: An analysis of the heart-beat monitor of a telephone switch using VeriSoft," in *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA. New York, NY, USA: ACM, Mar. 1998, pp. 124–133. [Online]. Available: https://doi.org/10.1145/271771.271800

[12] B. Jonsson, M. Lång, and K. Sagonas, "Awaiting for Godot: Stateless model checking that avoids executions where nothing happens," arXiv CoRR, Aug. 2022, Extended Version with Proofs. [Online]. Available: https://arxiv.org/abs/2208.09259

[13] ——, "Replication Package for Awaiting for Godot: Stateless Model Checking that Avoids Executions where Nothing Happens," Aug. 2022, artifact for the FMCAD 2022 paper with the same title. [Online]. Available: https://doi.org/10.5281/zenodo.6979940

[14] M. Kokologiannakis, O. Lahav, K. Sagonas, and V. Vafeiadis, "Effective stateless model checking for C/C++ concurrency," *Proc. ACM on Program. Lang.*, vol. 2, no. POPL, pp. 17:1–17:32, Jan. 2018. [Online]. Available: https://doi.org/10.1145/3158105

[15] M. Kokologiannakis, A. Raad, and V. Vafeiadis, "Model checking for weakly consistent libraries," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2019. New York, NY, USA: ACM, Jun. 2019, pp. 96–110. [Online]. Available: https://doi.org/10.1145/3314221.3314609

[16] M. Kokologiannakis, X. Ren, and V. Vafeiadis, "Dynamic partial order reductions for spinloops," in *Formal Methods in Computer Aided Design*, ser. FMCAD 2021. IEEE, Oct. 2021, pp. 163–172. [Online]. Available: https://doi.org/10.34727/2021/isbn.978-3-85448-046-4_25

[17] M. Kokologiannakis and K. Sagonas, "Stateless model checking of the Linux kernel's hierarchical read-copy-update (tree RCU)," in *Proceedings of International SPIN Symposium on Model Checking of Software*, ser. SPIN 2017. New York, NY, USA: ACM, 2017, pp. 172–181. [Online]. Available: https://doi.org/10.1145/3092282.3092287

[18] M. Kokologiannakis and V. Vafeiadis, "GenMC: A model checker for weak memory models," in *Computer Aided Verification - 33rd International Conference, CAV 2021, Proceedings, Part I*, ser. LNCS, vol. 12759. Springer, Jul. 2021, pp. 427–440. [Online]. Available: https://doi.org/10.1007/978-3-030-81685-8_20

[19] M. Lesani, T. D. Millstein, and J. Palsberg, "Automatic atomicity verification for clients of concurrent data structures," in *Computer Aided Verification, CAV 2014*, ser. LNCS, A. Biere and R. Bloem, Eds., vol. 8559. Cham: Springer, Jul. 2014, pp. 550–567. [Online]. Available: https://doi.org/10.1007/978-3-319-08867-9_37

[20] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu, "Finding and reproducing heisenbugs in concurrent programs," in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, ser. OSDI '08. Berkeley, CA, USA: USENIX Association, Dec. 2008, pp. 267–280. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855741.1855760

[21] B. Norris and B. Demsky, "A practical approach for model checking C/C++11 code," *ACM Trans. Program. Lang. Syst.*, vol. 38, no. 3, pp. 10:1–10:51, May 2016. [Online]. Available: http://doi.acm.org/10.1145/2806886

[22] J. Oberhauser, R. L. d. L. Chehab, D. Behrens, M. Fu, A. Paolillo, L. Oberhauser, K. Bhat, Y. Wen, H. Chen, J. Kim, and V. Vafeiadis, "Vsync: Push-button verification and optimization for synchronization primitives on weak memory models," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 2021. New York, NY, USA: ACM, 2021, p. 530–545. [Online]. Available: https://doi.org/10.1145/3445814.3446748

[23] P. Thomson, A. F. Donaldson, and A. Betts, "Concurrency testing using controlled schedulers: An empirical study," *ACM Trans. Parallel Comput.*, vol. 2, no. 4, pp. 23:1–23:37, 2016. [Online]. Available: http://doi.acm.org/10.1145/2858651

[24] N. Zhang, M. Kusano, and C. Wang, "Dynamic partial order reduction for relaxed memory models," in *Programming Language Design and Implementation (PLDI)*. New York, NY, USA: ACM, Jun. 2015, pp. 250–259. [Online]. Available: http://doi.acm.org/10.1145/2737924.2737956