Quasi-Optimal Partial Order Reduction

Huyen T.T. Nguyen[1] [0000−0002−9255−219X], César Rodríguez[1,3], Marcelo Sousa[2], Camille Coti[1] [0000−0002−1224−7786], and Laure Petrucci[1] [0000−0003−3154−5268]

[1] Université Paris 13, Sorbonne Paris Cité, CNRS, France
[2] University of Oxford, United Kingdom
[3] Diffblue Ltd. Oxford, United Kingdom

Abstract. A dynamic partial order reduction (DPOR) algorithm is optimal when it always explores at most one representative per Mazurkiewicz trace. Existing literature suggests that the reduction obtained by the non-optimal, state-of-the-art Source-DPOR (SDPOR) algorithm is comparable to optimal DPOR. We show the first program[4] with $\mathcal{O}(n)$ Mazurkiewicz traces where SDPOR explores $\mathcal{O}(2^n)$ redundant schedules. We furthermore identify the cause of this blow-up as an NP-hard problem. Our main contribution is a new approach, called Quasi-Optimal POR, that can arbitrarily approximate an optimal exploration using a provided constant k. We present an implementation of our method in a new tool called Dpu using specialised data structures. Experiments with Dpu, including Debian packages, show that optimality is achieved with low values of k, outperforming state-of-the-art tools.

1 Introduction

Dynamic partial-order reduction (DPOR) [10,1,19] is a mature approach to mitigate the state explosion problem in stateless model checking of multithreaded programs. DPORs are based on Mazurkiewicz trace theory [13], a true-concurrency semantics where the set of executions of the program is partitioned into equivalence classes known as Mazurkiewicz traces (M-traces). In a DPOR, this partitioning is defined by an independence relation over concurrent actions that is computed dynamically and the method explores executions which are representatives of M-traces. The exploration is *sound* when it explores all M-traces, and it is considered *optimal* [1] when it explores each M-trace only once.

Since two independent actions might have to be explored from the same state in order to explore all M-traces, a DPOR algorithm uses independence to compute a provably-sufficient subset of the enabled transitions to explore for each state encountered. Typically this involves the combination of forward reasoning (persistent sets [11] or source sets [1,4]) with backward reasoning (sleep sets [11])

[4] As this paper was under review, we were made aware of the recent publication of another paper [3] which contains an independently-discovered example program with the same characteristics.

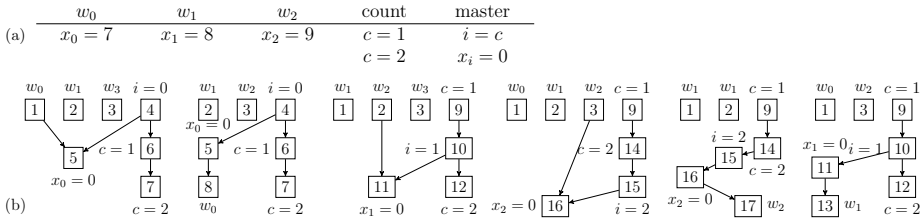


Fig. 1: (a): Programs; (b): Partially-ordered executions;

to obtain a more efficient exploration. However, in order to obtain optimality, a DPOR needs to compute sequences of transitions (as opposed to sets of enabled transitions) that avoid visiting a previously visited M-trace. These sequences are stored in a data structure called *wakeup trees* in [1] and known as *alternatives* in [19]. Computing these sequences thus amounts to deciding whether the DPOR needs to visit yet another M-trace (or all have already been seen).

In this paper, we prove that computing alternatives in an optimal DPOR is an NP-complete problem. To the best our knowledge this is the first formal complexity result on this important subproblem that optimal and non-optimal DPORs need to solve. The program shown in Fig. 1 (a) illustrates a practical consequence of this result: the non-optimal, state-of-the-art SDPOR algorithm [1] can explore here $\mathcal{O}(2^n)$ interleavings but the program has only $\mathcal{O}(n)$ M-traces.

The program contains $n := 3$ *writer* threads w_0, w_1, w_2, each writing to a different variable. The thread *count* increments $n - 1$ times a zero-initialized counter c. Thread *master* reads c into variable i and writes to x_i.

The statements $x_0 = 7$ and $x_1 = 8$ are independent because they produce the same state regardless of their execution order. Statements $i = c$ and any statement in the *count* thread are dependent or *interfering*: their execution orders result in different states. Similarly, $x_i = 0$ interferes with exactly one *writer* thread, depending on the value of i.

Using this independence relation, the set of executions of this program can be partitioned into six M-traces, corresponding to the six partial orders shown in Fig. 1 (b). Thus, an optimal DPOR explores six executions ($2n$-executions for n *writers*). We now show why SDPOR explores $\mathcal{O}(2^n)$ in the general case. Conceptually, SDPOR is a loop that (1) runs the program, (2) identifies two dependent statements that can be swapped, and (3) reverses them and re-executes the program. It terminates when no more dependent statements can be swapped.

Consider the interference on the counter variable c between the *master* and the *count* thread. Their execution order determines which *writer* thread interferes with the *master* statement $x_i = 0$. If $c = 1$ is executed just before $i = c$, then $x_i = 0$ interferes with w_1. However, if $i = c$ is executed before, then $x_i = 0$ interferes with w_0. Since SDPOR does not track relations between dependent statements, it will naively try to reverse the race between $x_i = 0$ and *all writer threads*, which results in exploring $\mathcal{O}(2^n)$ executions. In this program, exploring only six traces requires understanding the entanglement between both interferences as the order in which the first is reversed determines the second.

As a trade-off solution between solving this NP-complete problem and potentially explore an exponential number of redundant schedules, we propose a hybrid approach called Quasi-Optimal POR (QPOR) which can turn a non-optimal DPOR into an optimal one. In particular, we provide a polynomial algorithm to compute alternative executions that can arbitrarily approximate the optimal solution based on a user specified constant k. The key concept is a new notion of *k-partial alternative*, which can intuitively be seen as a "good enough" alternative: they revert two interfering statements while remembering the resolution of the last $k-1$ interferences.

The major differences between QPOR and the DPORs of [1] are that: 1) QPOR is based on prime event structures [17], a partial-order semantics that has been recently applied to programs [19,21], instead of a sequential view to thread interleaving, and 2) it computes k-partial alternatives with an $\mathcal{O}(n^k)$ algorithm while optimal DPOR corresponds to computing ∞-partial alternatives with an $\mathcal{O}(2^n)$ algorithm. For the program shown in Fig. 1 (a), QPOR achieves optimality with $k = 2$ because races are coupled with (at most) another race. As expected, the cost of computing k-partial alternatives and the reductions obtained by the method increase with higher values of k.

Finding k-partial alternatives requires decision procedures for traversing the causality and conflict relations in event structures. Our main algorithmic contribution is to represent these relations as a set of trees where events are encoded as one or two nodes in two different trees. We show that checking causality/conflict between events amounts to an efficient traversal in one of these trees.

In summary, our main contributions are:

- Proof that computing alternatives for optimal DPOR is NP-complete (Sec. 4).
- Efficient data structures and algorithms for (1) computing k-partial alternatives in polynomial time, and (2) represent and traverse partial orders (Sec. 5).
- Implementation of QPOR in a new tool called Dpu and experimental evaluations against SDPOR in Nidhugg and the testing tool Maple (Sec. 6).
- Benchmarks with $\mathcal{O}(n)$ M-traces where SDPOR explores $\mathcal{O}(2^n)$ executions (Sec. 6).

Furthermore, in Sec. 6 we show that: (1) low values of k often achieve optimality; (2) even with non-optimal explorations Dpu greatly outperforms Nidhugg; (3) Dpu copes with production code in Debian packages and achieves much higher state space coverage and efficiency than Maple.

Proofs for all our formal results are available in the unabridged version [15].

2 Preliminaries

In this section we provide the formal background used throughout the paper.

Concurrent Programs. We consider deterministic concurrent programs composed of a fixed number of threads that communicate via shared memory and synchronize using mutexes (Fig. 1 (a) can be trivially modified to satisfy this). We also

assume that local statements can only modify shared memory within a mutex block. Therefore, it suffices to only consider races of mutex accesses.

Formally, a *concurrent program* is a structure $P := \langle \mathcal{M}, \mathcal{L}, T, m_0, l_0 \rangle$, where $\mathcal{M}$ is the set of *memory states* (valuations of program variables, including instruction pointers), $\mathcal{L}$ is the set of *mutexes*, m_0 is the *initial memory state*, l_0 is the *initial mutexes state* and T is the set of *thread statements*. A thread statement $t := \langle i, f \rangle$ is a pair where $i \in \mathbb{N}$ is the *thread identifier* associated with the statement and $f \colon \mathcal{M} \to (\mathcal{M} \times \Lambda)$ is a *partial* function that models the transformation of the memory as well as the *effect* $\Lambda := \{\texttt{loc}\} \cup (\{\texttt{acq}, \texttt{rel}\} \times \mathcal{L})$ of the statement with respect to thread synchronization. Statements of `loc` effect model local thread code. Statements associated with $\langle \texttt{acq}, x \rangle$ or $\langle \texttt{rel}, x \rangle$ model lock and unlock operations on a mutex x. Finally, we assume that (1) functions f are PTIME-decidable; (2) `acq`/`rel` statements do not modify the memory; and (3) `loc` statements modify thread-shared memory only within lock/unlock blocks. When (3) is violated, then P has a *datarace* (undefined behavior in almost all languages), and our technique can be used to find such statements, see Sec. 6.

We use *labelled transition systems* (*LTS*) semantics for our programs. We associate a program P with the *LTS* $M_P := \langle \mathcal{S}, \to, A, s_0 \rangle$. The set $\mathcal{S} := \mathcal{M} \times (\mathcal{L} \to \{0, 1\})$ are the *states* of M_P, i.e., pairs of the form $\langle m, v \rangle$ where m is the state of the memory and v indicates when a mutex is locked (1) or unlocked (0). The *actions* in $A \subseteq \mathbb{N} \times \Lambda$ are pairs $\langle i, b \rangle$ where i is the identifier of the thread that executes some statement and b is the effect of the statement. We use the function $p \colon A \to \mathbb{N}$ to retrieve the thread identifier. The *transition relation* $\to \subseteq \mathcal{S} \times A \times \mathcal{S}$ contains a triple $\langle m, v \rangle \xrightarrow{\langle i,b \rangle} \langle m', v' \rangle$ exactly when there is some thread statement $\langle i, f \rangle \in T$ such that $f(m) = \langle m', b \rangle$ and either (1) $b = \texttt{loc}$ and $v' = v$, or (2) $b = \langle \texttt{acq}, x \rangle$ and $v(x) = 0$ and $v' = v_{|x \mapsto 1}$, or (3) $b = \langle \texttt{rel}, x \rangle$ and $v' = v_{|x \mapsto 0}$. Notation $f_{x \mapsto y}$ denotes a function that behaves like f for all inputs except for x, where $f(x) = y$. The *initial state* is $s_0 := \langle m_0, l_0 \rangle$.

Furthermore, if $s \xrightarrow{a} s'$ is a transition, the action a is *enabled* at s. Let $enabl(s)$ denote the set of actions enabled at s. A sequence $\sigma := a_1 \dots a_n \in A^*$ is a *run* when there are states $s_1, \dots, s_n$ satisfying $s_0 \xrightarrow{a_1} s_1 \dots \xrightarrow{a_n} s_n$. We define $state(\sigma) := s_n$. We let $runs(M_P)$ denote the set of all runs and $reach(M_P) := \{state(\sigma) \in \mathcal{S} \colon \sigma \in runs(M_P)\}$ the set of all *reachable states*.

Independence. Dynamic partial-order reduction methods use a notion called independence to avoid exploring concurrent interleavings that lead to the same state. We recall the standard notion of independence for actions in [11]. Two actions $a, a' \in A$ *commute at* a state $s \in \mathcal{S}$ iff

- if $a \in enabl(s)$ and $s \xrightarrow{a} s'$, then $a' \in enabl(s)$ iff $a' \in enabl(s')$; and
- if $a, a' \in enabl(s)$, then there is a state s' such that $s \xrightarrow{a.a'} s'$ and $s \xrightarrow{a'.a} s'$.

Independence between actions is an under-approximation of commutativity. A binary relation $\diamondsuit \subseteq A \times A$ is an *independence* on M_P if it is symmetric, irreflexive, and every pair $\langle a, a' \rangle$ in $\diamondsuit$ commutes at every state in $reach(M_P)$.

In general M_P has multiple independence relations, clearly $\emptyset$ is always one of them. We define relation $\diamondsuit_P \subseteq A \times A$ as the smallest irreflexive, symmetric

relation where $\langle i, b\rangle \diamondsuit_P \langle i', b'\rangle$ holds if $i \neq i'$ and either $b = \texttt{loc}$ or $b = \texttt{acq}\ x$ and $b' \notin \{\texttt{acq}\ x, \texttt{rel}\ x\}$. By construction $\diamondsuit_P$ is always an independence.

Labelled Prime Event Structures. *Prime event structures* (PES) are well-known non-interleaving, partial-order semantics [16,8,7]. Let X be a set of actions. A PES *over* X is a structure $\mathcal{E} := \langle E, <, \#, h\rangle$ where E is a set of *events*, $< \subseteq E \times E$ is a strict partial order called *causality relation*, $\# \subseteq E \times E$ is a symmetric, irreflexive *conflict relation*, and $h\colon E \to X$ is a labelling function. Causality represents the happens-before relation between events, and conflict between two events expresses that any execution includes at most one of them. Fig. 2 (b) shows a PES over $\mathbb{N} \times \Lambda$ where causality is depicted by arrows, conflicts by dotted lines, and the labelling h is shown next to the events, e.g., $1 < 5$, $8 < 12$, $2 \# 8$, and $h(1) = \langle 0, \texttt{loc}\rangle$. The *history* of an event e, $\lceil e\rceil := \{e' \in E\colon e' < e\}$, is the least set of events that need to happen before e.

The notion of concurrent execution in a PES is captured by the concept of *configuration*. A configuration is a (partially ordered) execution of the system, i.e., a set $C \subseteq E$ of events that is *causally closed* (if $e \in C$, then $\lceil e\rceil \subseteq C$) and *conflict-free* (if $e, e' \in C$, then $\neg(e \# e')$). In Fig. 2 (b), the set $\{8, 9, 15\}$ is a configuration, but $\{3\}$ or $\{1, 2, 8\}$ are not. We let $\mathit{conf}(\mathcal{E})$ denote the set of all configurations of $\mathcal{E}$, and $[e] := \lceil e\rceil \cup \{e\}$ the *local configuration* of e. In Fig. 2 (b), $[11] = \{1, 8, 9, 10, 11\}$. A configuration represents a set of *interleavings* over X. An interleaving is a sequence in X^* that labels any topological sorting of the events in C. We denote by $\mathit{inter}(C)$ the set of interleavings of C. In Fig. 2 (b), $\mathit{inter}(\{1, 8\}) = \{ab, ba\}$ with $a := \langle 0, \texttt{loc}\rangle$ and $b := \langle 1, \texttt{acq}\ m\rangle$.

The *extensions* of C are the events not in C whose histories are included in C: $\mathit{ex}(C) := \{e \in E\colon e \notin C \wedge \lceil e\rceil \subseteq C\}$. The *enabled* events of C are the extensions that can form a larger configuration: $\mathit{en}(C) := \{e \in \mathit{ex}(C)\colon C \cup \{e\} \in \mathit{conf}(\mathcal{E})\}$. Finally, the *conflicting extensions* of C are the extensions that are not enabled: $\mathit{cex}(C) := \mathit{ex}(C) \setminus \mathit{en}(C)$. In Fig. 2 (b), $\mathit{ex}(\{1, 8\}) = \{2, 9, 15\}$, $\mathit{en}(\{1, 8\}) = \{9, 15\}$, and $\mathit{cex}(\{1, 8\}) = \{2\}$. See [20] for more information on PES concepts.

Parametric Unfolding Semantics. We recall the program PES semantics of [19,20] (modulo notation differences). For a program P and any independence $\diamondsuit$ on M_P we define a PES $\mathcal{U}_{P,\diamondsuit}$ that represents the behavior of P, i.e., such that the interleavings of its set of configurations equals $\mathit{runs}(M_P)$.

Each event in $\mathcal{U}_{P,\diamondsuit}$ is defined by a canonical name of the form $e := \langle a, H\rangle$, where $a \in A$ is an action of M_P and H is a configuration of $\mathcal{U}_{P,\diamondsuit}$. Intuitively, e represents the action a after the *history* (or the causes) H. Fig. 2 (b) shows an example. Event 11 is $\langle\langle 0, \texttt{acq}\ m\rangle, \{1, 8, 9, 10\}\rangle$ and event 1 is $\langle\langle 0, \texttt{loc}\rangle, \emptyset\rangle$. Note the inductive nature of the name, and how it allows to uniquely identify each event. We define the *state of a configuration* as the state reached by *any* of its interleavings. Formally, for $C \in \mathit{conf}(\mathcal{U}_{P,\diamondsuit})$ we define $\mathit{state}(C)$ as s_0 if $C = \emptyset$ and as $\mathit{state}(\sigma)$ for some $\sigma \in \mathit{inter}(C)$ if $C \neq \emptyset$. Despite its appearance $\mathit{state}(C)$ is well-defined because *all* sequences in $\mathit{inter}(C)$ reach the *same* state, see [20] for a proof.

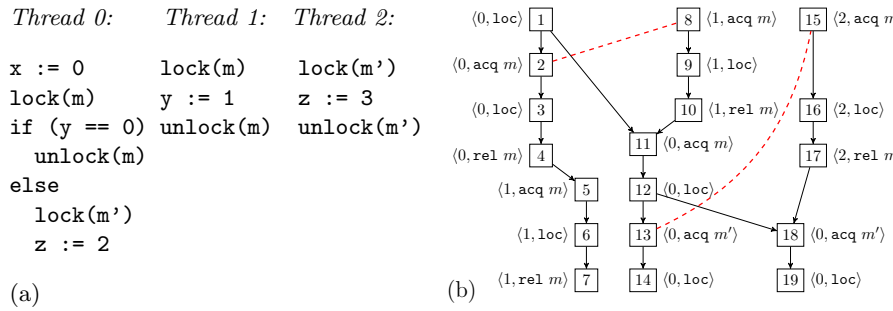


Fig. 2: (a): a program P; (b): its unfolding semantics $\mathcal{U}_{P,\diamondsuit_P}$.

Definition 1 (Unfolding). *Given a program P and some independence relation $\diamondsuit$ on $M_P := \langle \mathcal{S}, \to, A, s_0 \rangle$, the* unfolding of P under $\diamondsuit$, *denoted $\mathcal{U}_{P,\diamondsuit}$, is the* PES *over A constructed by the following fixpoint rules:*

1. *Start with a* PES $\mathcal{E} := \langle E, <, \#, h \rangle$ *equal to* $\langle \emptyset, \emptyset, \emptyset, \emptyset \rangle$.
2. *Add a new event $e := \langle a, C \rangle$ to E for any configuration $C \in conf(\mathcal{E})$ and any action $a \in A$ if a is enabled at $state(C)$ and $\neg(a \diamondsuit h(e'))$ holds for every $<$-maximal event e' in C.*
3. *For any new e in E, update $<$, $\#$, and h as follows: for every $e' \in C$, set $e' < e$; for any $e' \in E \setminus C$, set $e' \# e$ if $e \neq e'$ and $\neg(a \diamondsuit h(e'))$; set $h(e) := a$.*
4. *Repeat steps 2 and 3 until no new event can be added to E; return $\mathcal{E}$.*

Step 1 creates an empty PES with only one (empty) configuration. Step 2 inserts a new event $\langle a, C \rangle$ by finding a configuration C that enables an action a which is dependent with all causality-maximal events in C. In Fig. 2, this initially creates events 1, 8, and 15. For event $1 := \langle \langle 0, \texttt{loc} \rangle, \emptyset \rangle$, this is because action $\langle 0, \texttt{loc} \rangle$ is enabled at $state(\emptyset) = s_0$ and there is no $<$-maximal event in $\emptyset$ to consider. Similarly, the state of $C_1 := \{1, 8, 9, 10\}$ enables action $a_1 := \langle 0, \texttt{acq}\ m \rangle$, and both $h(1)$ and $h(10)$ are dependent with a_1 in $\diamondsuit_P$. As a result $\langle a_1, C_1 \rangle$ is an event (number 11). Furthermore, while $a_2 := \langle 0, \texttt{loc} \rangle$ is enabled at $state(C_2)$, with $C_2 := \{8, 9, 10\}$, a_2 is independent of $h(10)$ and $\langle a_2, C_2 \rangle$ is not an event.

After inserting an event $e := \langle a, C \rangle$, Def. 1 declares all events in C causal predecessors of e. For any event e' in E but not in $[e]$ such that $h(e')$ is dependent with a, the order of execution of e and e' yields different states. We thus set them in conflict. In Fig. 2, we set $2 \# 8$ because $h(2)$ is dependent with $h(8)$ and $2 \notin [8]$ and $8 \notin [2]$.

3 Unfolding-Based DPOR

This section presents an algorithm that exhaustively explores all deadlock states of a given program (a *deadlock* is a state where no thread is enabled).

For the rest of the paper, unless otherwise stated, we let P be a *terminating* program (i.e., $runs(M_P)$ is a finite set of finite sequences) and $\diamondsuit$ an independence on M_P. Consequently, $\mathcal{U}_{P,\diamondsuit}$ has finitely many events and configurations.

Algorithm 1: Unfolding-based POR exploration. See text for definitions.

```
1  Initially, set U := ∅,
2  and call Explore(∅, ∅, ∅).
3  Procedure Explore(C, D, A)
4      Add ex(C) to U
5      if en(C) ⊆ D return
6      if A = ∅
7          Choose e from en(C) \ D
8      else
9          Choose e from A ∩ en(C)
10     Explore(C ∪ {e}, D, A \ {e})
11     if ∃J ∈ Alt(C, D ∪ {e})
12         Explore(C, D ∪ {e}, J \ C)
13     U := U ∩ Q_{C,D}
14 Function cexp(C)
15     R := ∅
16     foreach event e ∈ C of type acq
17         e_t := pt(e)
18         e_m := pm(e)
19         while ¬(e_m ≤ e_t) do
20             e_m := pm(e_m)
21             if (e_m < e_t) break
22             e_m := pm(e_m)
23             ê := ⟨h(e), [e_t] ∪ [e_m]⟩
24             Add ê to R
25     return R
```

Our POR algorithm (Alg. 1) analyzes P by exploring the configurations of $\mathcal{U}_{P,\diamond}$. It visits all $\subseteq$-maximal configurations of $\mathcal{U}_{P,\diamond}$, which correspond to the deadlock states in $\mathit{reach}(M_P)$, and organizes the exploration as a binary tree.

`Explore`(C, D, A) has a global set U that stores all events of $\mathcal{U}_{P,\diamond}$ discovered so far. The three arguments are: C, the configuration to be explored; D (for *disabled*), a set of events that shall never be visited (included in C) again; and A (for *add*), used to direct the exploration towards a configuration that conflicts with D. A call to `Explore`(C, D, A) visits all maximal configurations of $\mathcal{U}_{P,\diamond}$ which contain C and do not contain D, and the first one explored contains $C \cup A$.

The algorithm first adds $ex(C)$ to U. If C is a maximal configuration (i.e., there is no enabled event) then line 5 returns. If C is not maximal but $en(C) \subseteq D$, then all possible events that could be added to C have already been explored and this call was redundant work. In this case the algorithm also returns and we say that it has explored a *sleep-set blocked* (SSB) execution [1]. Alg. 1 next selects an event enabled at C, if possible from A (line 7 and 9) and makes a recursive call (left subtree) that explores *all* configurations that contain all events in $C \cup \{e\}$ and no event from D. Since that call visits all maximal configurations containing C and e, it remains to visit those containing C but not e. At line 11 we determine if any such configuration exists. Function `Alt` returns a set of configurations, so-called *clues*. A clue is a witness that a $\subseteq$-maximal configuration exists in $\mathcal{U}_{P,\diamond}$ which contains C and not $D \cup \{e\}$.

Definition 2 (Clue). *Let D and U be sets of events, and C a configuration such that $C \cap D = \emptyset$. A* clue *to D after C in U is a configuration $J \subseteq U$ such that $C \cup J$ is a configuration and $D \cap J = \emptyset$.*

Definition 3 (`Alt` function). *Function* `Alt` *denotes* any *function such that* $\texttt{Alt}(B, F)$ *returns a set of clues to* F *after* B *in* U*, and the set is non-empty if* $\mathcal{U}_{P,\diamond}$ *has at least one maximal configuration* C *where* $B \subseteq C$ *and* $C \cap F = \emptyset$*.*

When `Alt` returns a clue J, the clue is passed in the second recursive call (line 12) to "mark the way" (using set A) in the subsequent recursive calls at line 10, and guide the exploration towards the maximal configuration that J witnesses. Def. 3 does not identify a concrete implementation of `Alt`. It rather indicates how to implement `Alt` so that Alg. 1 terminates and is complete (see below). Different PORs in the literature can be reframed in terms of Alg. 1. SDPOR [1] uses clues that mark the way with only one event ahead ($|J \setminus C| = 1$) and can hit SSBs. Optimal DPORs [1,19] use size-varying clues that guide the exploration provably guaranteeing that any SSB will be avoided.

Alg. 1 is *optimal* when it does not explore a SSB. To make Alg. 1 optimal `Alt` needs to return clues that are *alternatives* [19], which satisfy stronger constraints. When that happens, Alg. 1 is equivalent to the DPOR in [19] and becomes optimal (see [20] for a proof).

Definition 4 (Alternative [19]). *Let* D *and* U *be sets of events and* C *a configuration such that* $C \cap D = \emptyset$*. An* alternative *to* D *after* C *in* U *is a clue* J *to* D *after* C *in* U *such that* $\forall e \in D : \exists e' \in J,\ e \# e'$*.*

Line 13 removes from U events that will not be necessary for `Alt` to find clues in the future. The events preserved, $Q_{C,D} := C \cup D \cup \#(C \cup D)$, include all events in $C \cup D$ as well as every event in U that is in conflict with some event in $C \cup D$. The preserved events will suffice to compute alternatives [19], but other non-optimal implementations of `Alt` could allow for more aggressive pruning.

The $\subseteq$-maximal configurations of Fig. 2 (b) are $[7] \cup [17]$, $[14]$, and $[19]$. Our algorithm starts at configuration $C = \emptyset$. After 10 recursive calls it visits $C = [7] \cup [17]$. Then it backtracks to $C = \{1\}$, calls $\texttt{Alt}(\{1\}, \{2\})$, which provides, e.g., $J = \{1, 8\}$, and visits $C = \{1, 8\}$ with $D = \{2\}$. After 6 more recursive calls it visits $C = [14]$, backtracks to $C = [12]$, calls $\texttt{Alt}([12], \{2, 13\})$, which provides, e.g., $J = \{15\}$, and after two more recursive calls it visits $C = [12] \cup \{15\}$ with $D = \{2, 13\}$. Finally, after 4 more recursive calls it visits $C = [19]$.

Finally, we focus on the correctness of Alg. 1, and prove termination and soundness of the algorithm:

Theorem 1 (Termination). *Regardless of its input, Alg. 1 always stops.*

Theorem 2 (Completeness). *Let* $\hat{C}$ *be a* $\subseteq$*-maximal configuration of* $\mathcal{U}_{P,\diamond}$*. Then Alg. 1 calls* $\texttt{Explore}(C, D, A)$ *at least once with* $C = \hat{C}$*.*

4 Complexity

This section presents complexity results about the only non-trival steps in Alg. 1: computing $ex(C)$ and the call to $\texttt{Alt}(\cdot, \cdot)$. An implementation of $\texttt{Alt}(B, F)$

that systematically returns B would satisfy Def. 3, but would also render Alg. 1 unusable (equivalent to a DFS in M_P). On the other hand the algorithm becomes optimal when `Alt` returns alternatives. Optimality comes at a cost:

Theorem 3. *Given a finite* PES $\mathcal{E}$*, some configuration* $C \in conf(\mathcal{E})$*, and a set* $D \subseteq ex(C)$*, deciding if an alternative to* D *after* C *exists in* $\mathcal{E}$ *is NP-complete.*

Theorem 3 assumes that $\mathcal{E}$ is an arbitrary PES. Assuming that $\mathcal{E}$ is the unfolding of a program P under $\diamondsuit_P$ does not reduce this complexity:

Theorem 4. *Let* P *be a program and* U *a causally-closed set of events from* $\mathcal{U}_{P,\diamondsuit_P}$*. For any configuration* $C \subseteq U$ *and any* $D \subseteq ex(C)$*, deciding if an alternative to* D *after* C *exists in* U *is NP-complete.*

These complexity results lead us to consider (in next section) new approaches that avoid the NP-hardness of computing alternatives while still retaining their capacity to prune the search.

Finally, we focus on the complexity of computing $ex(C)$, which essentially reduces to computing $cex(C)$, as computing $en(C)$ is trivial. Assuming that $\mathcal{E}$ is given, computing $cex(C)$ for some $C \in conf(\mathcal{E})$ is a linear problem. However, for any realistic implementation of Alg. 1, $\mathcal{E}$ is not available (the very goal of Alg. 1 is to find all of its events). So a useful complexity result about $cex(C)$ necessarily refers to the orignal system under analysis. When $\mathcal{E}$ is the unfolding of a Petri net [14], computing $cex(C)$ is NP-complete:

Theorem 5. *Let* N *be a Petri net,* t *a transition of* N*,* $\mathcal{E}$ *the unfolding of* N *and* C *a configuration of* $\mathcal{E}$*. Deciding if* $h^{-1}(t) \cap cex(C) = \emptyset$ *is NP-complete.*

Fortunately, computing $cex(C)$ for programs is a much simpler task. Function `cexp(`C`)`, shown in Alg. 1, computes and returns $cex(C)$ when $\mathcal{E}$ is the unfolding of some program. We explain `cexp(`C`)` in detail in Sec. 5.3. But assuming that functions `pt` and `pm` can be computed in constant time, and relation $<$ decided in $\mathcal{O}(\log |C|)$, as we will show, clearly `cexp` works in time $\mathcal{O}(n^2 \log n)$, where $n := |C|$, as both loops are bounded by the size of C.

5 New Algorithm for Computing Alternatives

This section introduces a new class of clues, called k-partial alternatives. These can arbitrarily reduce the number of redundant explorations (SSBs) performed by Alg. 1 and can be computed in polynomial time. Specialized data structures and algorithms for k-partial alternatives are also presented.

Definition 5 (k-partial alternative). *Let* U *be a set of events,* $C \subseteq U$ *a configuration,* $D \subseteq U$ *a set of events, and* $k \in \mathbb{N}$ *a number. A configuration* J *is a* k-partial alternative *to* D *after* C *if there is some* $\hat{D} \subseteq D$ *such that* $|\hat{D}| = k$ *and* J *is an alternative to* $\hat{D}$ *after* C*.*

A k-partial alternative needs to conflict with only k (instead of all) events in D. An alternative is thus an ∞-partial alternative. If we reframe SDPOR in terms of Alg. 1, it becomes an algorithm using *singleton 1-partial* alternatives. While k-partial alternatives are a very simple concept, most of their simplicity stems from the fact that they are expressed within the elegant framework of PES semantics. Defining the same concept on top of sequential semantics (often used in the POR literature [11,10,23,1,2,9]), would have required much more complex device.

We compute k-partial alternatives using a *comb* data structure:

Definition 6 (Comb). *Let A be a set. An* A-comb *c of size $n \in \mathbb{N}$ is an ordered collection of* spikes *$\langle s_1, \ldots, s_n \rangle$, where each spike $s_i \in A^*$ is a sequence of elements over A. Furthermore, a* combination *over c is any tuple $\langle a_1, \ldots, a_n \rangle$ where $a_i \in s_i$ is an element of the spike.*

It is possible to compute k-partial alternatives (and by extension optimal alternatives) to D after C in U using a comb, as follows:

1. Select k (or $|D|$, whichever is smaller) arbitrary events $e_1, \ldots, e_k$ from D.
2. Build a U-comb $\langle s_1, \ldots, s_k \rangle$ of size k, where spike s_i contains all events in U in conflict with e_i.
3. Remove from s_i any event $\hat{e}$ such that either $[\hat{e}] \cup C$ is not a configuration or $[\hat{e}] \cap D \neq \emptyset$.
4. Find combinations $\langle e'_1, \ldots, e'_k \rangle$ in the comb satisfying $\neg(e'_i \# e'_j)$ for $i \neq j$.
5. For any such combination the set $J := [e'_1] \cup \ldots \cup [e'_k]$ is a k-partial alternative.

Step 3 guarantees that J is a clue. Steps 1 and 2 guarantee that it will conflict with at least k events from D. It is straightforward to prove that the procedure will find a k-partial alternative to D after C in U when an ∞-partial alternative to D after C exists in U. It can thus be used to implement Def. 3.

Steps 2, 3, and 4 require to decide whether a given pair of events is in conflict. Similarly, step 3 requires to decide if two events are causally related. Efficiently computing k-partial alternatives thus reduces to efficiently computing causality and conflict between events.

5.1 Computing Causality and Conflict for PES events

In this section we introduce an efficient data structure for deciding whether two events in the unfolding of a program are causally related or in conflict.

As in Sec. 3, let P be a program, M_P its LTS semantics, and $\diamondsuit_P$ its independence relation (defined in Sec. 2). Additionally, let $\mathcal{E}$ denote the PES $\mathcal{U}_{P,\diamondsuit_P}$ of P extended with a new event $\bot$ that causally precedes every event in $\mathcal{U}_{P,\diamondsuit_P}$.

The unfolding $\mathcal{E}$ represents the dependency of actions in M_P through the causality and conflict relations between events. By definition of $\diamondsuit_P$ we know that for any two events $e, e' \in \mathcal{E}$:

- If e and e' are events from the same thread, then they are either causally related or in conflict.

– If e and e' are lock/unlock operations on the same variable, then similarly they are either causally related or in conflict.

This means that the causality/conflict relations between all events of one thread can be tracked using a tree. For every thread of the program we define and maintain a so-called *thread tree*. Each event of the thread has a corresponding node in the tree. A tree node n is the parent of another tree node n' iff the event associated with n is the immediate causal predecessor of the event associated with n'. That is, the ancestor relation of the tree encodes the causality relations of events in the thread, and the branching of the tree represents conflict. Given two events e, e' of the same thread we have that $e < e'$ iff $\neg(e \# e')$ iff the tree node of e is an ancestor of the tree node of e'.

We apply the same idea to track causality/conflict between `acq` and `rel` events. For every lock $l \in \mathcal{L}$ we maintain a separate *lock tree*, containing a node for each event labelled by either $\langle \texttt{acq}, l \rangle$ or $\langle \texttt{rel}, l \rangle$. As before, the ancestor relation in a lock tree encodes the causality relations of all events represented in that tree. Events of type `acq`/`rel` have tree nodes in both their lock and thread trees. Events for `loc` actions are associated to only one node in the thread tree.

This idea gives a procedure to decide a causality/conflict query for two events when they belong to the same thread or modify the same lock. But we still need to decide causality and conflict for other events, e.g., `loc` events of different threads. Again by construction of $\diamondsuit_P$, the only source of conflict/causality for events are the causality/conflict relations between the causal predecessors of the two. These relations can be summarized by keeping two mappings for each event:

Definition 7. *Let $e \in E$ be an event of $\mathcal{E}$. We define the* thread mapping *$tmax \colon E \times \mathbb{N} \to E$ as the only function that maps every pair $\langle e, i \rangle$ to the unique $<$-maximal event from thread i in $[e]$, or $\perp$ if $[e]$ contains no event from thread i. Similarly, the* lock mapping *$lmax \colon E \times \mathcal{L} \to E$ maps every pair $\langle e, l \rangle$ to the unique $<$-maximal event $e' \in [e]$ such that $h(e')$ is an action of the form $\langle \texttt{acq}, l \rangle$ or $\langle \texttt{rel}, l \rangle$, or $\perp$ if no such event exists in $[e]$.*

The information stored by the thread and lock mappings enables us to decide causality and conflict queries for arbitrary pairs of events:

Theorem 6. *Let $e, e' \in \mathcal{E}$ be two arbitrary events from resp. threads i and i', with $i \neq i'$. Then $e < e'$ holds iff $e \leqslant tmax(e', i)$. And $e \# e'$ holds iff there is some $l \in \mathcal{L}$ such that $lmax(e, l) \# lmax(e', l)$.*

As a consequence of Theorem 6, deciding whether two events are related by causality or conflict reduces to deciding whether two nodes from the *same* lock or thread tree are ancestors.

5.2 Computing Causality and Conflict for Tree Nodes

This section presents an efficient algorithm to decide if two nodes of a tree are ancestors. The algorithm is similar to a search in a skip list [18].

Let $\langle N, \lessdot, r\rangle$ denote a tree, where N is a set of *nodes*, $\lessdot \subseteq N \times N$ is the *parent relation*, and $r \in N$ is the root. Let $d(n)$ be the depth of each node in the tree, with $d(r) = 0$. A node n is an *ancestor* of n' if it belongs to the only path from r to n'. Finally, for a node $n \in N$ and some integer $g \in \mathbb{N}$ such that $g \leq d(n)$ let $q(n, g)$ denote the unique ancestor n' of n such that $d(n') = g$.

Given two *distinct* nodes $n, n' \in N$, we need to efficiently decide whether n is an ancestor of n'. The key idea is that if $d(n) = d(n')$, then the answer is clearly negative; and if the depths are different and w.l.o.g. $d(n) < d(n')$, then we have that n is an ancestor of n' iff nodes n and $n'' := q(n', d(n))$ are the same node.

To find n'' from n', a linear traversal of the branch starting from n' would be expensive for deep trees. Instead, we propose to use a data structure similar to a skip list. Each node stores a pointer to the parent node *and* also a number of pointers to ancestor nodes at distances $s^1, s^2, s^3, \ldots$, where $s \in \mathbb{N}$ is a user-defined *step*. The number of pointers stored at a node n is equal to the number of trailing zeros in the s-ary representation of $d(n)$. For instance, for $s := 2$ a node at depth 4 stores 2 pointers (apart from the pointer to the parent) pointing to the nodes at depth $4 - s^1 = 2$ and depth $4 - s^2 = 0$. Similarly a node at depth 12 stores a pointer to the ancestor (at depth 11) and pointers to the ancestors at depths 10 and 8. With this algorithm computing $q(n, g)$ requires traversing $\log(d(n) - g)$ nodes of the tree.

5.3 Computing Conflicting Extensions

We now explain how function `cexp`(C) in Alg. 1 works. A call to `cexp`(C) constructs and returns all events in $cex(C)$. The function works only when the PES being explored is the unfolding of a program P under the independence $\diamondsuit_P$.

Owing to the properties of $\mathcal{U}_{P,\diamondsuit_P}$, all events in $cex(C)$ are labelled by `acq` actions. Broadly speaking, this is because only the actions from different threads that are co-enabled *and* are dependent create conflicts in $\mathcal{U}_{P,\diamondsuit_P}$. And this is only possible for `acq` statements. For the same reason, an event labelled by $a := \langle i, \langle \texttt{acq}, l\rangle\rangle$ exists in $cex(C)$ iff there is some event $e \in C$ such that $h(e) = a$.

Function `cexp` exploits these facts and the lock tree introduced in Sec. 5.1 to compute $cex(C)$. Intuitively, it finds every event e labelled by an $\langle \texttt{acq}, l\rangle$ statement and tries to "execute" it before the $\langle \texttt{rel}, l\rangle$ that happened before e (if there is one). If it can, it creates a new event $\hat{e}$ with the same label as e.

Function `pt`(e) returns the only immediate causal predecessor of event e in its own thread. For an `acq`/`rel` event e, function `pm`(e) returns the parent node of event e in its lock tree (or $\perp$ if e is the root). So for an `acq` event it returns a `rel` event, and for a `rel` event it returns an `acq` event.

6 Experimental Evaluation

We implemented QPOR in a new tool called Dpu (*Dynamic Program Unfolder*, available at `https://github.com/cesaro/dpu/releases/tag/v0.5.2`). Dpu is a stateless model checker for C programs with POSIX threading. It uses the

LLVM infrastructure to parse, instrument, and JIT-compile the program, which is assumed to be data-deterministic. It implements k-partial alternatives (k is an input), optimal POR, and context-switch bounding [6].

Dpu does not use data-races as a source of thread interference for POR. It will not explore two execution orders for the two instructions that exhibit a data-race. However, it can be instructed to detect and report data races found during the POR exploration. When requested, this detection happens for a user-provided percentage of the executions explored by POR.

6.1 Comparison to SDPOR

In this section we investigate the following experimental questions: (a) How does QPOR compare against SDPOR? (b) For which values of k do k-partial alternatives yield optimal exploration?

We use realistic programs that expose complex thread synchronization patterns including a job dispatcher, a multiple-producer multiple-consumer scheme, parallel computation of π, and a thread pool. Complex synchronizations patterns are frequent in these examples, including nested and intertwined critical sections or conditional interactions between threads based on the processed data, and provide means to highlight the differences between POR approaches and drive improvement. Each program contains between 2 and 8 assertions, often ensuring invariants of the used data structures. All programs are safe and have between 90 and 200 lines of code. We also considered the SV-COMP'17 benchmarks, but almost all of them contain very simple synchronization patterns, not representative of more complex concurrent algorithms. On these benchmarks QPOR and SDPOR perform an almost identical exploration, both timeout on exactly the same instances, and both find exactly the same bugs.

In Table 1, we present a comparison between Dpu and Nidhugg [2], an efficient implementation of SDPOR for multithreaded C programs. We run k-partial alternatives with $k \in \{1, 2, 3\}$ and optimal alternatives. The number of SSB executions dramatically decreases as k increases. With $k = 3$ almost no instance produces SSBs (except MPC(4,5)) and optimality is achieved with $k = 4$. Programs with simple synchronization patterns, e.g., the Pi benchmark, are explored optimally both with $k = 1$ and by SDPOR, while more complex synchronization patterns require $k > 1$.

Overall, if the benchmark exhibits many SSBs, the run time reduces as k increases, and optimal exploration is the fastest option. However, when the benchmark contains few SSBs (cf., Mpat, Pi, Poke), k-partial alternatives can be slightly faster than optimal POR, an observation inline with previous literature [1]. Code profiling revealed that when the comb is large and contains many solutions, both optimal and non-optimal POR will easily find them, but optimal POR spends additional time constructing a larger comb. This suggests that optimal POR would profit from a lazy comb construction algorithm.

Dpu is faster than Nidhugg in the majority of the benchmarks because it can greatly reduce the number of SSBs. In the cases where both tools explore the same set of executions, Dpu is in general faster than Nidhugg because

Benchmark			Dpu (k=1)		Dpu (k=2)		Dpu (k=3)		Dpu (optimal)		Nidhugg		
Name	Th	Confs	Time	SSB	Time	SSB	Time	SSB	Time	Mem	Time	Mem	SSB
Disp(5,2)	8	137	0.8	1K	0.4	43	0.4	0	**0.4**	37	1.2	33	2K
Disp(5,3)	9	2K	5.4	11K	1.3	595	1.0	1	**1.0**	37	10.8	33	13K
Disp(5,4)	10	15K	58.5	105K	16.4	6K	10.3	213	**10.3**	87	109	33	115K
Disp(5,5)	11	151K	TO	-	476	53K	280	2K	**257**	729	TO	33	-
Disp(5,6)	12	?	TO	-	TO	-	TO	-	TO	1131	TO	33	-
Mpat(4)	9	384	0.5	0	N/A		N/A		**0.5**	37	0.6	33	0
Mpat(5)	11	4K	2.4	0	N/A		N/A		2.7	37	**1.8**	33	0
Mpat(6)	13	46K	50.6	0	N/A		N/A		73.2	214	**21.5**	33	0
Mpat(7)	15	645K	TO	-	TO	-	TO	-	TO	660	**359**	33	0
Mpat(8)	17	?	TO	-	TO	-	TO	-	TO	689	TO	33	-
MPC(2,5)	8	60	0.6	560	0.4	0			**0.4**	38	2.0	34	3K
MPC(3,5)	9	3K	26.5	50K	3.0	3K	1.7	0	**1.7**	38	70.7	34	90K
MPC(4,5)	10	314K	TO	-	TO	-	391	30K	**296**	239	TO	33	-
MPC(5,5)	11	?	TO	-	TO	-	TO	-	TO	834	TO	34	-
Pi(5)	6	120	**0.4**	0	N/A		N/A		0.5	39	19.6	35	0
Pi(6)	7	720	**0.7**	0	N/A		N/A		0.7	39	123	35	0
Pi(7)	8	5K	**3.5**	0	N/A		N/A		4.0	45	TO	34	-
Pi(8)	9	40K	48.1	0	N/A		N/A		**42.9**	246	TO	34	-
Pol(7,3)	14	3K	48.5	72K	2.9	1K	**1.9**	6	1.9	39	74.1	33	90K
Pol(8,3)	15	4K	153	214K	5.5	3K	**3.0**	10	3.0	52	251	33	274K
Pol(9,3)	16	5K	464	592K	9.5	5K	**4.8**	15	4.8	73	TO	33	-
Pol(10,3)	17	7K	TO	-	17.2	9K	**6.8**	21	7.1	99	TO	33	-
Pol(11,3)	18	10K	TO	-	27.2	12K	**9.7**	28	10.6	138	TO	33	-
Pol(12,3)	19	12K	TO	-	46.3	20K	**13.5**	36	16.4	184	TO	33	-

Table 1: Comparing QPOR and SDPOR. Machine: Linux, Intel Xeon 2.4GHz. TO: timeout after 8 min. Columns are: Th: nr. of threads; Confs: maximal configurations; Time in seconds, Memory in MB; SSB: Sleep-set blocked executions. N/A: analysis with lower k yielded 0 SSBs.

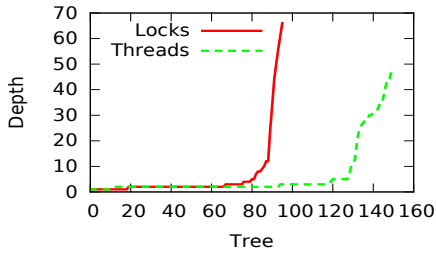
it JIT-compiles the program, while Nidhugg interprets it. All the benchmark in Table 1 are data-race free, but Nidhugg cannot be instructed to ignore data-races and will attempt to revert them. Dpu was run with data-race detection disabled. Enabling it will incur in approximatively 10% overhead. In contrast with previous observations [1,2], the results in Table 1 show that SSBs can dramatically slow down the execution of SDPOR.

6.2 Evaluation of the Tree-based Algorithms

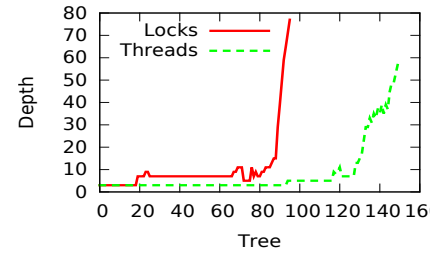
We now evaluate the efficiency of our tree-based algorithms from Sec. 5 answering: (a) What are the average/maximal depths of the thread/lock sequential trees? (b) What is the average depth difference on causality/conflict queries? (c) What is the best step for branch skip lists? We do not compare our algorithms against others because to the best of our knowledge none is available (other than a naive implementation of the mathematical definition of causality/conflict).

We run Dpu with an optimal exploration over 15 selected programs from Table 1, with 380 to 204K maximal configurations in the unfolding. In total, the 15 unfoldings contain 246 trees (150 thread trees and 96 lock trees) with 5.2M nodes. Fig. 3 shows the average depth of the nodes in each tree (subfigure a) and the maximum depth of the trees (subfigure b), for each of the 246 trees.

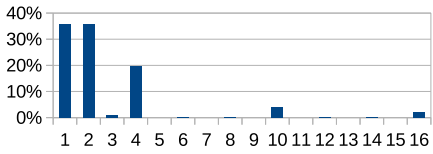
While the average depth of a node is 22.7, as much as 80% of the trees have a maximum depth of less than 8 nodes, and 90% of them less than 16 nodes. The



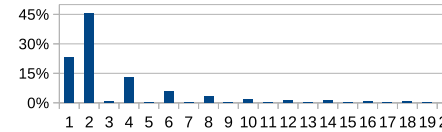
(a) Average depth of the tree nodes



(b) Maximum depth of the trees



(c) Depth-distance frequency on causality queries



(d) Depth-distance frequency on conflict queries

Fig. 3: (a), (b): depths of trees; (c), (d): frequency of depth distances

average of 22.7 is however larger because deeper trees contain proportionally more nodes. The depth of the deepest node of every tree was between 3 and 77.

We next evaluate depth differences in the causality and conflict queries over these trees. Fig. 3 (a) and (b) respectively show the frequency of various depth distances associated to causality and conflict queries made by optimal POR.

Surprisingly, depth differences are very small for both causality and conflict queries. When deciding causality between events, as much as 92% of the queries were for tree nodes separated by a distance between 1 and 4, and 70% had a difference of 1 or 2 nodes. This means that optimal POR, and specifically the procedure that adds $ex(C)$ to the unfolding (which is the main source of causality queries), systematically performs causality queries which are trivial with the proposed data structures. The situation is similar for checking conflicts: 82% of the queries are about tree nodes whose depth difference is between 1 and 4.

These experiments show that most queries on the causality trees require very short walks, which strongly drives to use the data structure proposed in Sec. 5. Finally, we chose a (rather arbitrary) skip step of 4. We observed that other values do not significantly impact the run time/memory consumption for most benchmarks, since the depth difference on causality/conflict requests is very low.

6.3 Evaluation Against the State-of-the-art on System Code

We now evaluate the scalability and applicability of Dpu on five multithreaded programs in two Debian packages: *blktrace* [5], a block layer I/O tracing mechanism, and *mafft* [12], a tool for multiple alignment of amino acid or nucleotide sequences. The code size of these utilities ranges from 2K to 40K LOC, and *mafft* is parametric in the number of threads.

We compared DPU against MAPLE [24], a state-of-the-art testing tool for multithreaded programs, as the top ranked verification tools from SV-COMP'17 are still unable to cope with such large and complex multithreaded code. Unfortunately we could not compare against NIDHUGG because it cannot deal with the (abundant) C-library calls in these programs.

Table 2 presents our experimental results. We use DPU with optimal exploration and the modified version of MAPLE used in [22]. To test the effectiveness of both approaches in state space coverage and bug finding, we introduce bugs in 4 of the benchmarks (ADD,DND,MDL,PLA). For the safe benchmark BLK, we perform exhaustive state-space exploration using MAPLE's DFS mode. On this benchmark, DPU outperfors MAPLE by several orders of magnitude: DPU explores up to 20K executions covering the entire state space in 10s, while MAPLE only explores up to 108 executions in 8 min.

Benchmark			DPU			MAPLE		
Name	LOC	Th	Time	Ex	R	Time	Ex	R
ADD(2)	40K	3	24.3	2	U	2.7	2	S
ADD(4)	40K	5	25.5	24	U	34.5	24	U
ADD(6)	40K	7	48.1	720	U	TO	316	U
ADD(8)	40K	9	TO	14K	U	TO	329	U
ADD(10)	40K	11	TO	14K	U	TO	295	U
BLK(5)	2K	2	0.9	1	S	4.6	1	S
BLK(15)	2K	2	0.9	5	S	23.3	5	S
BLK(18)	2K	2	1.0	180	S	TO	105	S
BLK(20)	2K	2	1.5	1147	S	TO	106	S
BLK(22)	2K	2	2.6	5424	S	TO	108	S
BLK(24)	2K	2	10.0	20K	S	TO	105	S
DND(2,4)	16K	3	11.1	80	U	122	80	U
DND(4,2)	16K	5	11.8	96	S	151	96	S
DND(4,4)	16K	5	TO	13K	U	TO	360	U
DND(6,2)	16K	7	149.3	4320	S	TO	388	S
MDL(1,4)	38K	7	26.1	1	U	1.4	1	U
MDL(2,2)	38K	5	29.2	9	U	13.3	9	U
MDL(2,3)	38K	5	46.2	576	U	TO	304	U
MDL(3,2)	38K	7	31.1	256	U	402	256	U
MDL(4,3)	38K	9	TO	14K	U	TO	329	U
PLA(1,5)	41K	2	22.8	1	U	1.7	1	U
PLA(2,4)	41K	3	37.2	80	U	142.4	80	U
PLA(4,3)	41K	5	160.5	1368	U	TO	266	U
PLA(6,3)	41K	7	TO	4580	U	TO	269	U

Table 2: Comparing DPU with Maple (same machine). LOC: lines of code; Execs: nr. of executions; R: safe or unsafe. Other columns as before. Timeout: 8 min.

For the remaining benchmarks, we use the random scheduler of MAPLE, considered to be the best baseline for bug finding [22]. First, we run DPU to retrieve a bound on the number of random executions to answer whether both tools are able to find the bug within the same number of executions. MAPLE found bugs in all buggy programs (except for one variant in ADD) even though DPU greatly outperforms and is able to achieve much more state space coverage.

6.4 Profiling a Stateless POR

In order to understand the cost of each component of the algorithm, we profile DPU on a selection of 7 programs from Table 1. DPU spends between 30% and 90% of the run time executing the program (65% in average). The remaining time is spent computing alternatives, distributed as follows: adding events to the event structure (15% to 30%), building the spikes of a new comb (1% to 50%), searching for solutions in the comb (less than 5%), and computing conflicting extensions (less than 5%). Counterintuitively, building the *comb* is more expensive than exploring it, even in the optimal case. Filling the spikes seems to be more memory-intensive than exploring the comb, which exploits data locality.

7 Conclusion

We have shown that computing alternatives in an optimal DPOR exploration is NP-complete. To mitigate this problem, we introduced a new approach to compute alternatives in polynomial time, approximating the optimal exploration with a user-defined constant. Experiments conducted on benchmarks including Debian packages show that our implementation outperforms current verification tools and uses appropriate data structures. Our profiling results show that running the program is often more expensive than computing alternatives. Hence, efforts in reducing the number of redundant executions, even if significantly costly, are likely to reduce the overall execution time.

References

1. Abdulla, P., Aronis, S., Jonsson, B., Sagonas, K.: Optimal dynamic partial order reduction. In: The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'14). ACM, ACM (2014)
2. Abdulla, P.A., Aronis, S., Atig, M.F., Jonsson, B., Leonardsson, C., Sagonas, K.: Stateless model checking for TSO and PSO. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). pp. 353–367. Springer (2015)
3. Abdulla, P.A., Aronis, S., Jonsson, B., Sagonas, K.: Source sets: A foundation for optimal dynamic partial order reduction **64**(4), 25:1–25:49
4. Abdulla, P.A., Aronis, S., Jonsson, B., Sagonas, K.: Comparing source sets and persistent sets for partial order reduction. In: Models, Algorithms, Logics and Tools - Essays Dedicated to Kim Guldstrand Larsen on the Occasion of His 60th Birthday. pp. 516–536 (2017)
5. blktrace: http://brick.kernel.dk/snaps/
6. Coons, K.E., Musuvathi, M., McKinley, K.S.: Bounded partial-order reduction. In: OOPSLA. pp. 833–848 (2013)
7. Esparza, J.: A false history of true concurrency: From Petri to tools. In: Pol, J.v.d., Weber, M. (eds.) Proc. SPIN, LNCS, vol. 6349, pp. 180–186. Springer (2010)
8. Esparza, J., Heljanko, K.: Unfoldings – A Partial-Order Approach to Model Checking. EATCS Monographs in Theoretical Computer Science, Springer (2008)
9. Farzan, A., Holzer, A., Razavi, N., Veith, H.: Con2colic Testing. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. pp. 37–47. ESEC/FSE 2013, ACM, New York, NY, USA (2013)
10. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: Principles of Programming Languages (POPL). pp. 110–121. ACM (2005). https://doi.org/10.1145/1040305.1040315
11. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem, LNCS, vol. 1032. Springer (1996)
12. MAFFT: http://mafft.cbrc.jp/alignment/software/
13. Mazurkiewicz, A.: Trace theory. In: Petri Nets: Applications and Relationships to Other Models of Concurrency, LNCS, vol. 255, pp. 278–324. Springer (1987)
14. McMillan, K.L.: Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In: Bochmann, G.v., Probst, D.K. (eds.) Proc. CAV'92. LNCS, vol. 663, pp. 164–177. Springer (1993)

15. Nguyen, H.T.T., Rodríguez, C., Sousa, M., Coti, C., Petrucci, L.: Quasi-optimal partial order reduction. CoRR **abs/1802.03950** (2018), http://arxiv.org/abs/1802.03950
16. Nielsen, M., Plotkin, G., Winskel, G.: Petri nets, event structures and domains, part I. Theoretical Computer Science **13**(1), 85–108 (1981)
17. Nielsen, M., Plotkin, G.D., Winskel, G.: Petri nets, event structures and domains. In: Proc. of the International Symposium on Semantics of Concurrent Computation. LNCS, vol. 70, pp. 266–284. Springer (1979)
18. Pugh, W.: Skip lists: A probabilistic alternative to balanced trees. In: Algorithms and Data Structures, Workshop WADS '89, Ottawa, Canada, August 17-19, 1989, Proceedings. pp. 437–449 (1989)
19. Rodríguez, C., Sousa, M., Sharma, S., Kroening, D.: Unfolding-based partial order reduction. In: Proc. CONCUR. pp. 456–469 (2015)
20. Rodríguez, C., Sousa, M., Sharma, S., Kroening, D.: Unfolding-based partial order reduction. CoRR **abs/1507.00980** (2015), http://arxiv.org/abs/1507.00980
21. Sousa, M., Rodríguez, C., D'Silva, V., Kroening, D.: Abstract interpretation with unfoldings. CoRR **abs/1705.00595** (2017), https://arxiv.org/abs/1705.00595
22. Thomson, P., Donaldson, A.F., Betts, A.: Concurrency testing using controlled schedulers: An empirical study. TOPC **2**(4), 23:1–23:37 (2016)
23. Yang, Y., Chen, X., Gopalakrishnan, G., Kirby, R.M.: Efficient stateful dynamic partial order reduction. In: Model Checking Software (SPIN), LNCS, vol. 5156, pp. 288–305. Springer (2008)
24. Yu, J., Narayanasamy, S., Pereira, C., Pokam, G.: Maple: A coverage-driven testing tool for multithreaded programs. In: OOPSLA. pp. 485–502 (2012)