Value-Centric Dynamic Partial Order Reduction

KRISHNENDU CHATTERJEE, IST Austria, Austria

ANDREAS PAVLOGIANNIS, EPFL, Switzerland

VIKTOR TOMAN, IST Austria, Austria

The verification of concurrent programs remains an open challenge, as thread interaction has to be accounted for, which leads to state-space explosion. Stateless model checking battles this problem by exploring traces rather than states of the program. As there are exponentially many traces, dynamic partial-order reduction (DPOR) techniques are used to partition the trace space into equivalence classes, and explore a few representatives from each class. The standard equivalence that underlies most DPOR techniques is the happens-before equivalence, however recent works have spawned a vivid interest towards coarser equivalences. The efficiency of such approaches is a product of two parameters: (i) the size of the partitioning induced by the equivalence, and (ii) the time spent by the exploration algorithm in each class of the partitioning.

In this work, we present a new equivalence, called value-happens-before and show that it has two appealing features. First, value-happens-before is always at least as coarse as the happens-before equivalence, and can be even exponentially coarser. Second, the value-happens-before partitioning is efficiently explorable when the number of threads is bounded. We present an algorithm called value-centric DPOR (VC-DPOR), which explores the underlying partitioning using polynomial time per class. Finally, we perform an experimental evaluation of VC-DPOR on various benchmarks, and compare it against other state-of-the-art approaches. Our results show that value-happens-before typically induces a significant reduction in the size of the underlying partitioning, which leads to a considerable reduction in the running time for exploring the whole partitioning.

CCS Concepts: • Theory of computation → Verification by model checking; • Software and its engineering → Formal software verification.

Additional Key Words and Phrases: concurrency, stateless model checking, partial-order reduction

ACM Reference Format:

1 INTRODUCTION

Model checking of concurrent programs. The formal analysis of concurrent programs is a key problem in program analysis and verification. Concurrency incurs a combinatorial explosion in
the behavior of the program, which makes errors hard to reproduce by testing (often identified as Heisenbugs [Musuvathi et al. 2008]). Thus, the formal analysis of concurrent program requires a systematic exploration of the state space, which is addressed by model checking [Clarke et al. 1999a]. However there are two key issues related to model-checking of concurrent programs: first, is related to the state-space explosion, and second, is related to the number of interleavings. Below we describe the main techniques to address these problems.

Stateless model checking. Model checkers typically store a large number of global states, and cannot handle realistic concurrent programs. The standard solution that is adopted to battle this problem on concurrent programs is stateless model checking [Godefroid 1996]. Stateless model-checking methods typically explore traces rather than states of the analyzed program, and only have to store a small number of traces. In such techniques, model checking is achieved by a scheduler, which drives the program execution based on the current interaction between the threads. The depth-first nature of the search enables it to be both systematic and memory-efficient. Stateless model-checking techniques have been employed successfully in several well-established tools, e.g., VeriSoft [Godefroid 1997, 2005] and CHESS [Madan Musuvathi 2007].

Partial-order Reduction (POR). While stateless model checking deals with the state-space issue, one key challenge that remains is exploring efficiently the exponential number of interleavings, which results from non-deterministic interprocess communication. There exist various techniques for reducing the number of explored interleavings, such as depth bounding and context bounding [Lal and Reps 2009; Musuvathi and Qadeer 2007]. One of the most well-studied techniques is partial-order reduction (POR) [Clarke et al. 1999b; Godefroid 1996; Peled 1993]. The main principle of POR is that two interleavings can be regarded as equal if they agree on the order of conflicting (dependent) events. In other words, POR considers certain pairs of traces to be equivalent, and the theoretical foundation of POR is the equivalence relation induced on the trace space, known as the happens-before (or Mazurkiewicz) equivalence HB [Mazurkiewicz 1987]. POR algorithms explore at least one trace from each equivalence class and guarantee a complete coverage of all behaviors that can occur in any interleaving, while exploring only a subset of the trace space. For the most interesting properties that arise in formal verification, such as safety, race freedom, absence of global deadlocks, and absence of assertion violations, POR-based algorithms make sound reports of correctness [Godefroid 1996].

Dynamic Partial-order Reduction (DPOR). Dynamic partial-order reduction (DPOR) is an on-the-fly version of POR [Flanagan and Godefroid 2005]. DPOR records conflicts that actually occur during the execution of traces, and thus is able to infer independence more frequently than static POR, which typically relies on over-approximations of conflicting events. Similar to POR, DPOR-based algorithms guarantee the exploration of at least one trace in each class of the happens-before partitioning. Recently, an optimal method for DPOR was developed [Abdulla et al. 2014] that explores exactly one trace from each happens-before equivalence class.

Efficiency of DPOR techniques. The efficiency of DPOR algorithms typically depends on two parameters, namely (i) the size of the trace-space partitioning and (ii) the time required to explore each class of the partitioning. The overall efficiency of the algorithm is a product of the two above, and there is usually a trade-off between the two, as coarser partitionings typically make the problem of moving between different classes of the partitioning computationally harder.

Beyond the Mazurkiewicz equivalence. Lately, there has been a considerable effort into going beyond Mazurkiewicz equivalence, by developing algorithms that explore partitionings of the trace space induced by equivalence relations that are coarser than Mazurkiewicz [Albert et al. 2017; Aronis
et al. 2018; Chalupa et al. 2017; Huang 2015]. Such approaches can be broadly classified as oracle-based methods that rely on NP-hard oracles such as SMT-solvers to guide the exploration, and explicit methods which avoid such computationally expensive oracles. Explicit methods include DC-DPOR [Chalupa et al. 2017] and Optimal DPOR with Observers [Aronis et al. 2018], which rely on equivalences provably coarser than the happens-before equivalence, as well as Context-sensitive DPOR [Albert et al. 2017] which sometimes might be coarser, but not always. On the other hand, oracle-based methods include MCR [Huang 2015] and SATCheck [Demsky and Lam 2015].

Value-centric DPOR. The happens-before equivalence and most coarser equivalences which admit an efficient exploration are insensitive to the values that variables take during the execution of a trace. On the other hand, it is well-understood that equivalences which are sensitive to such values can be very coarse, thereby reducing the size of the trace-space partitioning. An interesting approach with a value-centric partitioning was recently explored in [Huang 2015]. However, that approach is implicit and relies on expensive NP-oracles repeatedly for guiding the exploration of the partitioning. This NP bottleneck was identified in that work, and was subsequently only partially improved with static-analysis-based heuristics [Huang and Huang 2017]. Hence, the challenge of constructing value-centric equivalences that also admit efficient explorations has remained open. In the next section, we illustrate the benefits of such equivalences on a small example.

1.1 A Small Motivating Example

Consider the simple program given in Figure 1, which consists of two threads communicating over a global variable \( x \). We have two types of events: \( p_1 \) writes to \( x \) the value 1, whereas \( p_2 \) first writes to \( x \) the value 2, then writes to \( x \) the value 1, and finally it reads the value of \( x \) to its local variable. When we analyze this program, it becomes apparent that a model-checking algorithm can benefit if it takes into account the values written by the write events. Indeed, denote by \( w_j^i \) the \( j \)-th write event of thread \( i \), and by \( r \) the unique read event. There exist 4 Mazurkiewicz orderings.

\[
\begin{align*}
  t_1 : w_1^1 w_2^1 w_2^2 r & \quad t_2 : w_2^1 w_1^1 w_2^2 r \\
  t_3 : w_2^1 w_2^2 w_1^1 r & \quad t_4 : w_2^1 w_2^2 r w_1^1 \\
\end{align*}
\]

Hence, any algorithm that uses the Mazurkiewicz equivalence for exploring the trace space of the above program will have to explore at least 4 traces. Moreover, any sound algorithm that is insensitive to values will, in general, explore at least two traces (e.g., \( t_1 \) and \( t_3 \)), since the value read by \( r \) can, in principle, be different in both cases. This is true, for example, for DC-DPOR [Chalupa et al. 2017], which is based on the recently introduced Observation equivalence, as well as the Optimal DPOR with observers [Aronis et al. 2018] (which explores 3). On the other hand, it is clear that examining a single trace suffices for visiting all the local states of all threads. Although minimal, the above example illustrates the advantage that stateless model checking algorithms can gain by being sensitive to the values used by the events during an execution.

Fig. 1. A toy program with two threads.
1.2 Challenges and Our Contributions

Challenges. The above example illustrates that a value-centric partitioning can be coarse. To our knowledge, value-centric equivalences have been used systematically (i.e., with provable guarantees) only by implicit methods which rely on NP oracles (e.g., SMT-solvers [Huang 2015; Huang and Huang 2017]), which makes the exploration of the (reduced) partitionings computationally expensive. The challenge that arises naturally is to produce a partitioning that (a) is provably always coarser than Mazurkiewicz trace equivalence; and (b) is efficiently explorable (i.e., the time required in each step of the search is small/polynomial). In this work we address this challenge.

Our contributions. The main contribution of this work is a new value-centric equivalence, called value-happens-before \( (VHB) \). Intuitively, \( VHB \) distinguishes (arbitrarily) a thread of the program, called the root, from the other threads, called the leaves. The coarsening of the partitioning is achieved by \( VHB \) by relaxing the happens-before orderings between events that belong to the root and leaf threads. Given two traces \( t_1 \) and \( t_2 \) which have the same happens-before ordering on the events of leaf threads, \( VHB \) deems \( t_1 \) and \( t_2 \) equivalent by using a combination of (i) the values and (ii) the causally-happens-before orderings on pairs of events between the root and the leaves.

Properties of \( VHB \). We discuss two key properties of \( VHB \).

1. Soundness. The \( VHB \) equivalence is sound for reporting correctness of local-state properties. In particular, if \( t_1 \sim_{VHB} t_2 \), then every trace is guaranteed to visit the same local states in both executions. Thus, in order to report local-state-specific properties (e.g., absence of assertion violations), it is sound to explore a single representative from each class of the underlying partitioning. Global-state properties can be encoded as local properties by using a thread to monitor the global state. Due to this fact, many other recent works on DPOR focus on local-state properties only [Aronis et al. 2018; Chalupa et al. 2017; Huang 2015; Huang and Huang 2017].

2. Exponentially coarser than happens-before. The \( VHB \) is always at least as coarse as the happens-before (or Mazurkiewicz) equivalence, i.e., if two traces are \( HB \)-equivalent, then they are also \( VHB \)-equivalent. This implies that the underlying \( VHB \) partitioning is never larger than the \( HB \) partitioning. In addition, we show that there exist programs for which the \( VHB \) partitioning is exponentially smaller, thereby getting a significant reduction in one of the two factors that affect the efficiency of DPOR algorithms. Interestingly, this reduction is achieved even if there are no concurrent writes in the program.

Value-centric DPOR. We develop an efficient DPOR algorithm that explores the \( VHB \) partitioning, called VC-DPOR. This algorithm is guaranteed to visit every class of the \( VHB \) partitioning, and for a constant number of threads, the time spent in each class is polynomial. Hence, VC-DPOR explores efficiently a value-centric partitioning without relying on NP oracles. For example, in the program of Figure 1, VC-DPOR explores only one trace.

Experimental results. Finally, we make a prototype implementation of VC-DPOR and evaluate it on various classes of concurrency benchmarks. We use our implementation to assess (i) the coarseness of the \( VHB \) partitioning in practice, and (ii) the efficiency of VC-DPOR to explore such partitionings. To this end, we compare these two metrics with existing state-of-the-art explicit DPOR algorithms, namely, the Source-DPOR [Abdulla et al. 2014], Optimal-DPOR [Abdulla et al. 2014], Optimal-DPOR with observers [Aronis et al. 2018], as well as DC-DPOR [Chalupa et al. 2017]. Our results show a significant reduction in the size of the partitioning compared to the partitionings explored by existing techniques, which also typically leads to smaller running times.
2 PRELIMINARIES

2.1 Concurrent Computation Model

In this section we define the model of concurrent programs and introduce general notation. We follow a standard exposition found in the literature (e.g., [Abdulla et al. 2014; Chalupa et al. 2017]). For simplicity of presentation we do not consider locks in our model. Later, we remark how locks can be handled naturally by our approach (see Remark 4).

General notation. Given a natural number \( i \geq 1 \), we denote by \([i]\) the set \( \{1, 2, \ldots, i\} \). Given a map \( f : X \to Y \), we let \( \text{dom}(f) = X \) and \( \text{img}(f) = Y \) denote the domain and image sets of \( f \), respectively. We represent maps \( f \) as sets of tuples \( \{(x, f(x))\}_x \). Given two maps \( f_1, f_2 \), we write \( f_1 = f_2 \) to denote that \( \text{dom}(f_1) = \text{dom}(f_2) \) and for every \( x \in \text{dom}(f_1) \) we have \( f_1(x) = f_2(x) \), and we write \( f_1 \neq f_2 \) otherwise. A binary relation \( \sim \) on a set \( X \) is an equivalence relation iff \( \sim \) is reflexive, symmetric and transitive. Given an equivalence \( \sim_E \) and some \( x \in X \), we denote by \([x]_E \) the equivalence class of \( x \) under \( \sim_E \), i.e., \([x]_E = \{y \in X : x \sim_E y\}\). The quotient set \( X/E = \{[x]_E \mid x \in X\} \) of \( X \) under \( \sim_E \) is the set of all equivalence classes of \( X \) under \( \sim_E \).

Concurrent program. We consider a concurrent program \( \mathcal{H} = \{p_i\}_{i=1}^k \) of \( k \) threads communicating over shared memory, where \( k \) is some arbitrary constant. For simplicity of presentation, we neglect dynamic thread creation. We distinguish \( p_1 \) as the root thread of \( \mathcal{H} \), and refer to the remaining threads \( p_2, \ldots, p_k \) as leaf threads. The shared memory consists of a finite set \( G \) of global variables, where each variable receives values from a finite value domain \( D \). Every thread executes instructions, which we call events, and are of the following types.

1. A write event \( w \) writes a value \( v \in D \) to a global variable \( x \in G \).
2. A read event \( r \) reads the value \( v \in D \) of a global variable \( x \in G \).
3. A local (invisible) event is an event that does not access any global variable.

Although typically threads contain local events to guide the control-flow, such events are not relevant in our setting, and will thus be ignored. For simplicity of exposition, we consider that every thread is represented as an unrolled tree, which captures its unrolled control-flow, and every event is a node in this tree. In practice, each event is sufficiently identified by its thread identifier and an integer that counts how many preceding events of the same thread have been executed already. Given an event \( e \), we denote by \( p(e) \) the thread of \( e \) and by \( \text{loc}(e) \) the unique global variable that \( e \) accesses. We denote by \( E \) the set of all events, by \( W \) the set of write events, and by \( R \) the set of read events of \( \mathcal{H} \). Given a thread \( p \), we denote by \( E_p, W_p \) and \( R_p \) the set of events, read events and write events of \( p \), respectively. In addition, we let \( E_{=p} = \bigcup_{p \neq p'} E_{p'} \) and similarly for \( W_{=p} \) and \( R_{=p} \), i.e., \( E_{=p} \) denote the set of events of threads other than thread \( p \), and similarly, for \( W_{=p} \) and \( R_{=p} \). Finally, given a set \( X \subseteq E \), we let \( W(X) = X \cap W \) and \( R(X) = X \cap R \) for the set of write and read events of \( X \), respectively.

Concurrent program semantics. The semantics of \( \mathcal{H} \) are defined by means of a transition system over a state space of global states \( s = (\text{val}, L_1, \ldots, L_k) \), where \( \text{val} : G \to D \) is a value function that maps every global variable to a value, and \( L_1 \) is a local state of thread \( p_1 \), which contains the values of the local variables of each thread. The memory model considered here is sequentially consistent. Since the setting is standard, we omit here the formal setup and refer the reader to [Godefroid 2005] for details. As usual in stateless model checking, we focus our attention on state spaces \( S_\mathcal{H} \) that are acyclic (hence our focus is on bounded model checking).
Traces. A (concurrent) trace is a sequence of events \( t = e_1, \ldots, e_j \) that corresponds to a valid execution of \( \mathcal{H} \). Given a trace \( t \), we denote by \( \mathcal{E}(t) \) the set of events that appear in \( t \), and by \( \mathcal{R}(t) = \mathcal{E}(t) \cap \mathcal{R} \) (resp., \( W(t) = \mathcal{E}(t) \cap \mathcal{W} \)) the read (resp., write) events in \( t \). We let enabled\((t)\) denote the set of enabled events in the state reached after \( t \) is executed, and call \( t \) maximal if enabled\((t)\) = \( \emptyset \). We write \( \mathcal{T}_\mathcal{H} \) and \( \mathcal{T}_\mathcal{H}^{\text{max}} \) for the set of all traces and maximal traces, respectively, of \( \mathcal{H} \). Given a set of events \( A \), we denote by \( t|A \) the projection of \( t \) on \( A \), which is the unique subsequence of \( t \) that contains all events of \( A \cap \mathcal{E}(t) \), and only those.

Observation, side and value functions. Given a trace \( t \) and a read event \( r \in \mathcal{R}(t) \), the observation of \( r \) in \( t \) is the last write event \( w \) that appears before \( r \) in \( t \) such that \( \text{loc}(r) = \text{loc}(w) \). The observation function of \( t \) is a function \( O_t : \mathcal{R}(t) \rightarrow \mathcal{W}(t) \) such that \( O_t(r) \) is the observation of \( r \) in \( t \). The side function of \( t \) is a function \( S_t : \mathcal{R}(t) \cap \mathcal{R}_{p_1} \rightarrow [2] \) such that \( S_t(r) = 1 \) if \( p(O_t(r)) = p_1 \) and \( S_t(r) = 2 \) otherwise. In other words, a side function is defined for the read events of the root thread, and assigns 1 (resp., 2) to each read event if it observes a local (resp., remote) write event in the trace \( r \). The value function of \( t \) is a function \( \text{val}_t : \mathcal{E}(t) \rightarrow \mathcal{D} \) such that \( \text{val}_t(e) \) is the value of the global variable \( \text{loc}(e) \) after the prefix of \( t \) up to \( e \) has been executed. Note that since each thread is deterministic, this value is always unique and thus \( \text{val}_t \) is well-defined.

2.2 Problem and Complexity Parameters

The local-state reachability problem. The problem we address in this work is detecting erroneous local states of threads, e.g., whether a thread ever encounters an assertion violation. The underlying algorithmic problem is that of discovering every possible local state of every thread of \( \mathcal{H} \), and checking whether a bug occurs. In stateless model checking, the focal object for this task is the trace, and algorithms solve the problem by exploring different maximal traces of the trace space \( \mathcal{T}_\mathcal{H}^{\text{max}} \). DPOR techniques use an equivalence \( E \) to partition the trace space into equivalence classes, and explore the partitioning \( \mathcal{T}_\mathcal{H}^{\text{max}}/E \) instead of the whole space \( \mathcal{T}_\mathcal{H}^{\text{max}} \).

Complexity parameters. Given an equivalence \( E \) over \( \mathcal{T}_\mathcal{H}^{\text{max}} \), the efficiency of an algorithm that explores the partitioning \( \mathcal{T}_\mathcal{H}^{\text{max}}/E \) is typically a product of two factors \( O(\alpha \cdot \beta) \). The first factor \( \alpha \) is the size of the partitioning itself, i.e., \( \alpha = |\mathcal{T}_\mathcal{H}^{\text{max}}/E| \), which is typically exponentially large. As we construct coarser equivalences \( E \), \( \alpha \) decreases. The second factor \( \beta \) captures the amortized time on each explored class, and can be either polynomial (i.e., efficient) or exponential. There is a tradeoff between \( \alpha \) and \( \beta \): typically, for coarser equivalences \( E \) the algorithms spend more time to explore each class, and hence \( \alpha \) is decreased at the cost of increasing \( \beta \). Hence, the challenge is to make \( \alpha \) as small as possible without increasing \( \beta \) much.

This work. In this work, we introduce the value-happens before equivalence \( \mathcal{VHB} \) and show that the \( \mathcal{VHB} \)-partitioning is efficiently explorable. For a constant number of threads, which is typically the case, \( \beta = \text{poly}(n) \), i.e., \( \beta \) is polynomial in the length of the longest trace in \( \mathcal{T}_\mathcal{H}^{\text{max}} \). Since, on the other hand, \( \alpha \) is usually exponentially large in \( n \), we will not focus on establishing the exact dependency of \( \beta \) on \( n \). This helps to keep the exposition of the main message clear and focused.

Due to space restrictions, proofs and some experimental details appear in a technical report [Chatterjee et al. 2019].

---

\(^1\) Although the definition of side functions might appear arbitrary, we rely on this definition later for computing the \( \mathcal{VHB} \) abstraction.
2.3 Partial Orders

Here we introduce some useful notation around partial orders, which are the central objects of our algorithms in later sections.

Partial orders. Given a trace \( t \) and a set \( X \subseteq E(t) \), a (strict) partial order \( P(X) \) over \( X \) is an irreflexive, antisymmetric and transitive relation over \( X \) (i.e., \( <_{P(X)} \subseteq X \times X \)). When \( X \) is clear from the context, we will simply write \( P \) for the partial order \( P(X) \). Given two events \( e_1, e_2 \in X \), we write \( e_1 \preceq p e_2 \) to denote that \( e_1 < e_2 \) or \( e_1 = e_2 \). Given two distinct events \( e_1, e_2 \in X \), we say that \( e_1 \) and \( e_2 \) are unordered by \( P \), denoted by \( e_1 \parallel_P e_2 \), if neither \( e_1 < p e_2 \) nor \( e_2 < p e_1 \). Given a set \( Y \subseteq X \), we denote by \( P|Y \) the projection of \( P \) on the set \( Y \), i.e., \( <_{P|Y} \subseteq Y \times Y \), and for every pair of events \( e_1, e_2 \in Y \), we have that \( e_1 <_{P|Y} e_2 \) iff \( e_1 < p e_2 \). Given two partial orders \( P \) and \( Q \) over a common set \( X \), we say that \( Q \) refines \( P \), denoted by \( Q \subseteq P \), if for every pair of events \( e_1, e_2 \in X \), if \( e_1 < p e_2 \) then \( e_1 < Q e_2 \). We write \( Q \subset P \) to denote that \( Q \subseteq P \) and \( P \not\subseteq Q \). A linearization of \( P \) is a total order that refines \( P \). Note that a trace \( t \) is a partial (and, in fact, total) order over the set \( E(t) \).

Conflicting events, width and Mazurkiewicz width. Two events \( e_1, e_2 \) are called conflicting, written \( e_1 \cong e_2 \), if they access the same global variable and at least one writes to the variable. Let \( P \) be a partial order over a set \( X \). The width \( \text{width}(P) \) of \( P \) is the length of its longest antichain, i.e., it is the smallest integer \( i \) such that for every set \( Y \subseteq X \) of size \( i + 1 \), \( e_1, e_2 \in Y \) such that \( e_1 \parallel_P e_2 \). A set \( Y \subseteq X \) is called pairwise conflicting if for every pair of distinct events \( e_1, e_2 \in Y \), we have that \( e_1 \cong e_2 \). We define the Mazurkiewicz width \( \text{Mwidth}(P) \) of \( P \) as the smallest integer \( i \) such that for every pairwise conflicting set \( Y \subseteq X \) of size \( i + 1 \) there exists a pair \( e_1, e_2 \in Y \) such that \( e_1 \parallel_P e_2 \). Intuitively, \( \text{Mwidth}(P) \) is similar to \( \text{width}(P) \), with the difference that, in the first case, we focus on events that are conflicting as opposed to any events.

The thread order \( \text{TO} \). The thread order \( \text{TO} \) of \( \mathcal{H} \) is a partial order \( <_{\text{TO}} \subseteq E \times E \) that defines a fixed order between pairs of events of the same thread. For every trace \( t \in \mathcal{T}_\mathcal{H} \), we have that \( t \subseteq \text{TO}|E(t) \). Every partial order \( P \) used in this work respects the thread order.

Visible, maximal and minimal writes. Consider a partial order \( P \) over a set \( X \). Given a read event \( r \in R(X) \) we define the set of visible writes of \( r \) as

\[
\text{VisibleW}_r(P) = \{ w \in W(X) : r \cong w \text{ and } r <_P w \text{ and for each } w' \in W(X) \text{ s.t. } r \cong w', \text{ if } w <_P w' \text{ then } w' <_P r \}.
\]

In words, \( \text{VisibleW}_r(P) \) contains the write events \( w \) that conflict with \( r \) and are not “hidden” to \( r \) by \( P \), i.e., there exist linearizations \( t \) of \( P \) such that \( O_t(r) = w \) (note that here \( t \) is not necessarily an actual trace of \( \mathcal{H} \)). The set of minimal writes \( \text{MinW}_r(P) \) (resp., maximal writes \( \text{MaxW}_r(P) \)) of \( r \) contains the write events that are minimal (resp., maximal) elements in \( P|\text{VisibleW}_r(P) \).

The happens-before partial order. A trace \( t \) induces a happens-before partial order \( \Rightarrow_t \subseteq E(t) \times E(t) \), which is the smallest transitive relation on \( E(t) \) such that (i) \( \Rightarrow_t \subseteq \text{TO}|E(t) \) and (ii) \( e_1 \Rightarrow_t e_2 \) if \( e_1 <_t e_2 \) and \( e_1 \cong e_2 \).

The causally-happens-before partial order. A trace \( t \) induces a causally-happens-before partial order \( \mathrel{\llap H} \subseteq E(t) \times E(t) \), which is the smallest transitive relation on \( E(t) \) such that (i) \( \mathrel{\llap H} \subseteq \text{TO}|E(t) \) and (ii) for every read event \( r \in R(t) \), we have \( O_t(r) \mathrel{\llap H} r \). In words, \( \mathrel{\llap H} \) captures the flow of write events into read events, and is closed under composition with the thread order. Intuitively, for an event \( e \), the set of events \( e' \) that causally-happen-before \( e \) are the events that need to be present so
that $e$ is enabled. Note that $\rightarrow_t \subseteq \leftrightarrow_t$, i.e., the happens-before partial order refines the causally-happens-before partial order.

We refer to Figure 2 for an illustration of the $\rightarrow_t$ and $\leftrightarrow_t$ partial orders.

![Fig. 2. A trace $t$ of three threads.](image)

**3 THE VALUE-HAPPENS-BEFORE EQUIVALENCE**

In this section we introduce our new equivalence between traces, called value-happens-before, and prove some of its properties. We start with the happens-before equivalence, which has been used by DPOR algorithms in the literature.

**The happens-before equivalence.** Two traces $t_1, t_2 \in T_H$ are called happens-before-equivalent (commonly referred to as Mazurkiewicz equivalent), written $t_1 \sim_H t_2$, if the following hold.

1. $E(t_1) = E(t_2)$, i.e., they consist of the same set of events.
2. $\rightarrow t_1 = \rightarrow t_2$, i.e., their happens-before partial orders are equal.

**The value-happens-before equivalence.** Two traces $t_1, t_2 \in T_H$ are called value-happens-before-equivalent, written $t_1 \sim_{VHB} t_2$, if the following hold.

1. $E(t_1) = E(t_2)$, $val_{t_1} = val_{t_2}$, and $S_{t_1} = S_{t_2}$, i.e., they consist of the same set of events, and their value functions and side functions are equal.
2. $\leftrightarrow t_1 | R = \leftrightarrow t_2 | R$, i.e., $\leftrightarrow t_i$ agree on the read events.
3. $\rightarrow t_1 | E_{\neq p_1} = \rightarrow t_2 | E_{\neq p_1}$, i.e., $\rightarrow t_i$ agree on the events of the leaf threads.

**Remark 1 (Soundness).** Since every thread of $H$ is deterministic, for any two traces $t_1, t_2 \in T_H$ such that $E(t_1) = E(t_2)$ and $val_{t_1} = val_{t_2}$, the local states of each thread after executing $t_1$ and $t_2$ agree. It follows that any algorithm that explores every class of $T_H^{max} / VHB$ discovers every local state of every thread, and thus $VHB$ is a sound equivalence for local-state reachability.

**Exponential coarseness.** Here we provide two toy examples which illustrate different cases where $VHB$ can be exponentially coarser than $HB$, i.e., $T_H / VHB$ can have exponentially more classes than $T_H / HB$.

**Many operations on one variable.** First, consider the program shown in Figure 3a which consists of two threads $p_1$ and $p_2$, with $p_1$ being the root thread. This program has a single global variable $x$, and the threads perform operations on $x$ repeatedly. We assume a salient write event $w(x, 0)$ that writes
the initial value of \( x \). Consider any two traces \( t_1, t_2 \) that consist of the \( i \geq 0 \) first \( w(x) \) events of \( p_1 \) and \( j \geq 0 \) first \( r(x) \) events of \( p_2 \) (hence \( E(t_1) = E(t_2) \)). Since each \( w(x) \) writes the same value, we have \( \text{val}_1(r) = \text{val}_2(r) \) for every read event \( r \) in \( p_2 \). Moreover, since the root thread \( p_1 \) has no read events, we trivially have \( S_{t_1} = S_{t_2} \). Since all read events are on thread \( p_2 \), we have \( \leftrightarrow_{t_1}[R = \leftrightarrow_{t_1}[R = \top]\top[R(t_1)] \top[R(t_1)]. \) Finally, since we only have one leaf thread, \( \rightarrow_{t_1}[E_{\neq p_1} = \rightarrow_{t_2}[E_{\neq p_1} = \top\top[E_{\neq p_1}(t_1)] \top[E_{\neq p_1}(t_1)] \top[E_{\neq p_1}(t_1)]. \) We conclude that \( t_1 \sim_{\mathcal{VHB}} t_2, \) and thus given \( i \geq 0 \) and \( j \geq 0 \) there exists a single class of \( \sim_{\mathcal{VHB}} \) that contains the first \( i \) and first \( j \) events of \( p_1 \) and \( p_2 \), respectively. Thus \( |\mathcal{T}_{\mathcal{H}}/\mathcal{VHB}| = \Omega(n^2). \) On the other hand, given the first \( i \geq 0 \) and \( j \geq 0 \) events of threads \( p_1 \) and \( p_2 \), respectively, there exist \( \frac{(i+j)!}{i!j!} \) different ways to order them without violating the thread order. Observe that every such reordering induces a different happens-before relation. Using Stirling’s approximation, we obtain

\[
|\mathcal{T}_{\mathcal{H}}/\mathcal{H}\mathcal{B}| \geq \frac{(2 \cdot n)!}{(n!)^2} \geq \sqrt{\frac{2 \cdot \pi \cdot 2 \cdot n \cdot (2 \cdot n/e)^{2n}}{\sqrt{2 \cdot \pi \cdot n \cdot (n/e)^n}}} = \Omega\left(\frac{4^n}{\sqrt{n}}\right)
\]

**Few operations on many variables.** Now consider the example program shown in Figure 3b which consists of two threads \( p_1 \) and \( p_2 \), with \( p_1 \) being the root thread. We assume a salient write event \( w(x_1, 0) \) that writes the initial value of \( x_1 \). Consider any two traces \( t_1, t_2 \) that consist of the \( i \geq 0 \) first \( w(x) \) events of \( p_1 \) and \( j \geq 0 \) first \( r(x) \) events of \( p_2 \) (hence \( E(t_1) = E(t_2) \)). Since each \( w(x_1, 0) \) writes the same value, we have \( \text{val}_1(r) = \text{val}_2(r) \) for every read event \( r \) in \( p_2 \). Moreover, since the root thread \( p_1 \) has no read events, we trivially have \( S_{t_1} = S_{t_2} \). Since all read events are on thread \( p_2 \), we have \( \leftrightarrow_{t_1}[R = \leftrightarrow_{t_1}[R = \top]\top[R(t_1)] \top[R(t_1)]. \) Finally, since we only have one leaf thread, \( \rightarrow_{t_1}[E_{\neq p_1} = \rightarrow_{t_2}[E_{\neq p_1} = \top\top[E_{\neq p_1}(t_1)] \top[E_{\neq p_1}(t_1)] \top[E_{\neq p_1}(t_1)]. \) We conclude that \( t_1 \sim_{\mathcal{VHB}} t_2, \) and thus given \( i \geq 0 \) and \( j \geq 0 \) there exists a single class of \( \sim_{\mathcal{VHB}} \) that contains the first \( i \) and first \( j \) events of \( p_1 \) and \( p_2 \), respectively. Thus \( |\mathcal{T}_{\mathcal{H}}/\mathcal{VHB}| = \Omega(n^2). \) On the other hand, given the first \( i \) read events of \( p_2 \) and \( 2 \cdot i \) write events of \( p_1 \), there exist at least \( 2^i \) different observation functions that map each read event \( r \) to one of the two write events that \( r \) observes. Hence \( |\mathcal{T}_{\mathcal{H}}/\mathcal{H}\mathcal{B}| = \Omega(n^2). \)

**Theorem 3.1.** \( \mathcal{VHB} \) is sound for local-state reachability. Also, \( \mathcal{VHB} \) is at least as coarse as \( \mathcal{H}\mathcal{B}, \) and there exist programs where \( \mathcal{VHB} \) is exponentially coarser.

### 4 Closed Annotated Partial Orders

In this section we develop the core algorithmic concepts that will be used in the enumerative exploration of the \( \mathcal{VHB} \). We introduce **annotated partial orders**, which are traditional partial orders over events, with additional constraints. We formulate the question of the realizability of an annotated partial order \( \mathcal{P} \), which asks for a witness trace \( t \) that linearizes \( \mathcal{P} \) and satisfies the
constraints. We develop the notion of closure of annotated partial orders, and show that (i) an annotated partial order is realizable if and only if its closure exists, and (ii) deciding whether the closure exists can be done efficiently. This leads to an efficient procedure for deciding realizability.

4.1 Annotated Partial Orders

Here we introduce the notion of annotated partial orders, which is a central concept of our work. We build some definitions and notation, and provide some intuition around them.

Annotated Partial Orders. An annotated partial order is a tuple $\mathcal{P} = (X_1, X_2, P, \text{val}, S, \text{GoodW})$ where the following hold.

1. $X_1, X_2$ are sets of events such that $X_1 \cap X_2 = \emptyset$.
2. $P$ is a partial order over the set $X = X_1 \cup X_2$.
3. $\text{val}: X \rightarrow \mathcal{D}$ is a value function.
4. $S: \mathcal{R}(X_1) \rightarrow [2]$ is a side function.
5. $\text{GoodW}: \mathcal{R}(X) \rightarrow 2^{\mathcal{W}(X)}$ is a good-writes function such that $w \in \text{GoodW}(r)$ only if $r \triangleright w$ and $\text{val}(r) = \text{val}(w)$ and, if $r \in X_1$ then $w \in X_{S(r)}$.
6. $\text{width}(P|X_1) = \text{Mwidth}(P|X_2) = 1$.

We let the bad-writes function be $\text{BadW}(r) = \{w \in \mathcal{W}(X) \setminus \text{GoodW}(r) : r \triangleright w\}$. We call $\mathcal{P}$ consistent if for every thread $p$, we have that $\tau_p = \text{TO}[(X \cap E_p)]$ is a local trace of thread $p$ that occurs if every event $e$ of $\tau_p$ reads/writes the value $\text{val}(e)$. Hereinafter we only consider consistent annotated partial orders.

The realizability problem for annotated partial orders. Consider an annotated partial order $\mathcal{P} = (X_1, X_2, P, \text{val}, S, \text{GoodW})$. A trace $t$ is a linearization of $\mathcal{P}$ if (i) $t \subseteq P$ and (ii) for every read event $r \in \mathcal{R}(X_1 \cup X_2)$ we have that $O_t(r) \in \text{GoodW}(r)$. In words, $t$ must be a linearization of the partial order $P$ with the additional constraint that the observation function of $t$ must agree with the good-writes function $\text{GoodW}$ of $\mathcal{P}$. We call $\mathcal{P}$ realizable if it has a linearization. The associated realizability problem takes as input an annotated partial order $\mathcal{P}$ and asks whether $\mathcal{P}$ is realizable.

Remark 2 (Realizability to valid traces.). If $t$ is a linearization of some consistent annotated partial order $\mathcal{P}$ then $t$ is a valid (i.e., actual) trace of $\mathcal{H}$. This holds because of the following observations.

1. Since $t$ is a linearization of $\mathcal{P}$, we have $O_t(r) \in \text{GoodW}(r)$ for every read event $r \in \mathcal{R}(t)$.
2. Due to the previous item and the consistency of $\mathcal{P}$, for every thread $p$ we have that $\tau_p = \text{TO}[(X \cap E_p)]$ is a valid local trace of $p$.

Intuition. An annotated partial order $\mathcal{P}$ contains a partial order $P$ over a set $X = X_1 \cup X_2$ of events and the value of each event of $X$. Intuitively, the consistency of $\mathcal{P}$ states that we obtain the set of events $X$ if we execute each thread and force every read event in this execution to observe the value of a write event according to the good-writes function. In the next section, our VC-DPOR algorithm uses annotated partial orders to represent different classes of the $\mathcal{VHB}$ equivalence in order to guide the trace-space exploration. The set $X_1$ (resp., $X_2$) will contain the events of the root thread (resp., leaf threads). We will see that if VC-DPOR constructs two annotated partial orders $\mathcal{P}$ and $\mathcal{Q}$ during the exploration, then any two linearizations $t_1$ and $t_2$ of $\mathcal{P}$ and $\mathcal{Q}$, respectively, will satisfy that $t_1 \not\sim_{\mathcal{VHB}} t_2$, and hence $\mathcal{P}$ and $\mathcal{Q}$ represent different classes of the $\mathcal{VHB}$ partitioning.
Closed annotated partial orders. Consider an annotated partial order $\mathcal{P} = (X_1, X_2, P, \text{val}, S, \text{GoodW})$ and let $X = X_1 \cup X_2$. We say that $\mathcal{P}$ is closed if the following conditions hold for every read event $r \in \mathcal{R}(X)$.

1. There exists a write event $w \in \text{GoodW}(r) \cap \text{MinW}_P(r)$ such that $w <_P r$.
2. $\text{MaxW}_P(r) \cap \text{GoodW}(r) \neq \emptyset$.
3. For every write event $w' \in \text{BadW}(r) \cap \text{MinW}_P(r)$ such that $w' <_P r$ there exists a write event $w \in \text{GoodW}(r) \cap \text{VisibleW}_P(r)$ such that $w' <_P w$.

Our motivation behind this definition becomes clear from the following lemma, which states that closed annotated partial orders are realizable.

**Lemma 4.1.** If $\mathcal{P}$ is closed then it is realizable and a witness can be constructed in $O(\text{poly}(n))$ time.

In particular, the witness trace of $\mathcal{P}$ is constructed by the following process.

1. Create a partial order $\mathcal{Q}$ as follows.
   - (a) For every pair of events $e_1, e_2$ with $e_1 <_P e_2$, we have $e_1 <_Q e_2$.
   - (b) For every pair of events $e_1, e_2$ with $e_1 \in X_i$ for each $i \in [2]$, if $e_2 \not<_P e_1$ then $e_1 <_Q e_2$.
2. Create $t$ by linearizing $\mathcal{Q}$ arbitrarily.

The above construction is guaranteed to produce a valid witness trace for $\mathcal{P}$. The consistency of annotated partial orders guarantees that $t$ is a valid trace of the concurrent program $\mathcal{H}$ (see Remark 2). We provide an illustration of this construction later in Figure 5.

We now introduce the notion of closures. Intuitively, the closure of an annotated partial order $\mathcal{P}$ strengthens $\mathcal{P}$ by introducing the smallest set of event orderings such that the resulting annotated partial order $\mathcal{Q}$ is closed. The intuition behind the closure is the following: whenever a rule forces some ordering, any trace that witnesses the realizability of $\mathcal{P}$ also linearizes $\mathcal{Q}$. In some cases this operation results to cyclic orderings, and thus the closure does not exist. We also show that obtaining the closure or deciding that it does not exist can be done in polynomial time. Thus, in combination with Lemma 4.1, we obtain an efficient algorithm for deciding whether $\mathcal{P}$ is realizable, by deciding whether it has a closure.

**Closure of annotated partial orders.** Consider an annotated partial order $\mathcal{P} = (X_1, X_2, P, \text{val}, S, \text{GoodW})$. We say that an annotated partial order $\mathcal{Q} = (X_1, X_2, Q, \text{val}, S, \text{GoodW})$ is a closure of $\mathcal{P}$ if (i) $Q \subset P$, (ii) $Q$ is closed, and (iii) for any partial order $K$ with $Q \subset K \subset P$, we have that the annotated partial order $(X_1, X_2, K, \text{val}, S, \text{GoodW})$ is not closed. As the following lemma states, $\mathcal{P}$ can have at most one closure.

**Lemma 4.2.** There exists at most one weakest partial order $Q$ such that $Q \subset P$ and $(X_1, X_2, Q, \text{val}, S, \text{GoodW})$ is closed.

**Feasible annotated partial orders.** In light of Lemma 4.2, we define the closure of $\mathcal{P}$ as the unique annotated partial order $Q$ that is a closure of $\mathcal{P}$, if such $Q$ exists, and $\bot$ otherwise. We call $\mathcal{P}$ feasible if its closure is not $\bot$. We have the following lemma.

**Lemma 4.3.** $\mathcal{P}$ is realizable if and only if it is feasible.
Intuitively, Lemma 4.3 states that the closure rules give the weakest strengthening of $\mathcal{P}$ that is met by any linearization of $\mathcal{P}$. If that strengthening can be made (i.e., $\mathcal{P}$ is feasible), then $\mathcal{P}$ has a linearization. Hence, to decide whether $\mathcal{P}$ is realizable, it suffices to decide whether it is feasible, by computing its closure. In the next section we show that this computation can be done efficiently.

4.2 Computing the Closure

We now turn our attention to computing the closure of annotated partial orders, which will provide us with a way of solving the realizability problem.

**Algorithm** Closure. Consider an annotated partial order $\mathcal{P} = (X_1, X_2, P, \text{val}, S, \text{GoodW})$ and let $X = X_1 \cup X_2$. The algorithm Closure either computes the closure of $\mathcal{P}$, or concludes that $\mathcal{P}$ is not feasible, and returns $\perp$. Intuitively, the algorithm maintains a partial order $Q$, initially identical to $P$. The algorithm iterates over every read event $r$ and tests whether $r$ violates Item 1, Item 2 or Item 3 of the definition of closed annotated partial orders. When it discovers that $r$ violates one such closure rule, Closure calls one of the closure methods Rule1($r$), Rule2($r$), Rule3($r$), for violation of Item 1, Item 2 and Item 3 of the definition, respectively. In turn, each of these methods inserts a new ordering $e_1 \rightarrow e_2$ in $Q$, with the guarantee that if $\mathcal{P}$ has a closure $\mathcal{F} = (X_1, X_2, F, \text{val}, S, \text{GoodW})$, then $e_1 <_F e_2$. Hence, $e_1 \rightarrow e_2$ is a necessary ordering in the closure of $\mathcal{P}$. Finally, when the algorithm discovers that all closure rules are satisfied by every read event in $Q$, it returns the annotated partial order $(X_1, X_2, Q, \text{val}, S, \text{GoodW})$, which, due to Lemma 4.2, is guaranteed to be the closure of $\mathcal{P}$. We refer to Algorithm 1 for a formal description.

We now provide some intuition behind each of the closure methods. Given an event $e \in X$, we let $I_\mathcal{P}(e) = i$ such that $e \in X_i$. Given two events $e_1, e_2 \in X$, we say that $e_2$ is local to $e_1$ if $I_\mathcal{P}(e_1) = I_\mathcal{P}(e_2)$, i.e., $e_1$ and $e_2$ belong to the same set $X_i$. If $e_2$ is not local to $e_1$, then it is remote to $e_1$.

1. **Rule1($r$).** This rule is called when Item 1 of closure is violated, i.e., there exists no write event $w \in \text{GoodW}(r) \cap \text{MinW}_Q(r)$ such that $w <_Q r$. Observe that in this case there is no write event that is (i) local to $r$, (ii) good for $r$ and (iii) visible to $r$. To make $r$ respect this rule, the algorithm finds the first write event $w$ that is (i) good for $r$ and (ii) visible to $r$, and orders $w \rightarrow r$ in $Q$. See Figure 4a provides an illustration.

2. **Rule2($r$).** This rule is violated when $\text{MaxW}_Q(r) \cap \text{GoodW}(r) = \emptyset$, i.e., every maximal write event is bad for $r$. To make $r$ respect this rule, the algorithm finds the unique maximal write event $w$ that is remote to $r$ and orders $r \rightarrow w$ in $Q$. Rule2($r$) is called only if $r$ does not violate Item 1 of closure, which guarantees that $w$ exists. Figure 4b provides an illustration.

3. **Rule3($r$).** This rule is violated when there exists a write event $\overline{w} \in \text{BadW}(r) \cap \text{MinW}_Q(r)$ such that (i) $\overline{w} <_Q r$, and (ii) there exists no write event $w' \in \text{GoodW}(r) \cap \text{VisibleW}_Q(r)$ such that $\overline{w} <_Q w'$. To make $r$ respect this rule, the algorithm determines a maximal write event $w$ that is (i) remote to $\overline{w}$ and (ii) a good write for $r$, and orders $\overline{w} \rightarrow w$ in $Q$. Rule3($r$) is called only if $r$ does not violate either Item 1 or Item 2 of closure, which guarantees that $w$ exists. Figure 4c provides an illustration, depending on whether $\overline{w}$ is local or remote to $r$.

We have the following lemma regarding the correctness and complexity of Closure.

**Lemma 4.4.** Closure correctly computes the closure of $\mathcal{P}$ and requires $O(\text{poly}(n))$ time.

4.3 Realizing Annotated Partial Orders

Finally, we address the question of realizability of annotated partial orders. Lemma 4.3 implies that in order to decide whether an annotated partial order is realizable, it suffices to compute its closure,
Fig. 4. Illustration of the three closure operations Rule1(r) (a), Rule2(r) (b) and Rule3(r) (c). We follow the convention that barred and unbarred write events (\(\overline{w}\) and \(w\)) are bad writes and good writes for \(r\), respectively. In each case, the dashed edge shows the new order introduced by the algorithm in \(Q\).

Algorithm 1: Closure(\(P\))

\[
\begin{align*}
Q & \leftarrow P \quad \text{// We will strengthen } Q \text{ during the closure computation} \\
\text{Flag} & \leftarrow \text{True} \\
\text{while Flag do} \\
\text{foreach } r \in \mathcal{R}(X_1 \cup X_2) \text{ do} \\
& \quad \text{// Iterate over the reads} \\
& \quad \text{if } r \text{ violates Item 1 of closure then} \\
& \quad \quad \text{Call Rule1}(r) \quad \text{// Strengthen } Q \text{ to remove violation} \\
& \quad \quad \text{Flag} \leftarrow \text{True} \quad \text{// Repeat as new violations might have appeared} \\
& \quad \text{if } r \text{ violates Item 2 of closure then} \\
& \quad \quad \text{Call Rule2}(r) \quad \text{// Strengthen } Q \text{ to remove violation} \\
& \quad \quad \text{Flag} \leftarrow \text{True} \quad \text{// Repeat as new violations might have appeared} \\
& \quad \text{if } r \text{ violates Item 3 of closure then} \\
& \quad \quad \text{Call Rule3}(r) \quad \text{// Strengthen } Q \text{ to remove violation} \\
& \quad \quad \text{Flag} \leftarrow \text{True} \quad \text{// Repeat as new violations might have appeared} \\
\text{end} \\
\text{end} \\
\text{return } (X_1, X_2, Q, \text{val}, S, \text{GoodW}) \quad \text{// The closure of } P
\end{align*}
\]

Algorithm 2: Rule1(r)

\[
\begin{align*}
Y & \leftarrow \text{GoodW}(r) \cap \text{VisibleW}_Q(r) \\
\text{if } Y = \emptyset \text{ then return } \perp \\
w & \leftarrow \text{min}_Q(Y) \quad \text{// Since Rule 1 is violated, min}_Q(Y) \text{ is unique} \\
\text{Insert } w \rightarrow r \text{ in } Q
\end{align*}
\]

Algorithm 3: Rule2(r)

\[
\begin{align*}
w & \leftarrow \text{the unique event in MaxW}_Q(r) \cap X_{3-I_P}(r) \quad \text{// } w \text{ exists since Item 1 of closure holds} \\
\text{Insert } r \rightarrow w \text{ in } Q
\end{align*}
\]

and Lemma 4.4 states that the closure can be computed efficiently. Together, these two lemmas yield a simple algorithm for solving the realizability problem.
Algorithm Realize. We describe a simple algorithm Realize that decides whether an annotated partial order $P$ is realizable. The algorithms runs in two steps.

1. Use Lemma 4.4 to compute the closure of $P$. If the closure is $\perp$, report that $P$ is not realizable. Otherwise, the closure is an annotated partial order $Q$.
2. Use Lemma 4.1 to obtain a witness trace $t$ that linearizes $Q$. Report that $P$ is linearizable, and $t$ is the witness trace.

We conclude the results of this section with the following theorem.

**Theorem 4.5.** Let $P$ be an annotated partial order of $n$ events. Deciding whether $P$ is realizable requires $O(\text{poly}(n))$ time. If $P$ is realizable, a witness trace can be produced in $O(\text{poly}(n))$ time.

**Example on the realizability of annotated partial orders.** We illustrate Realize on a simple example in Figure 5 with an annotated partial order $P = (X_1, X_2, P, \text{val}, S, \text{GoodW})$, which we assume to be consistent. We have a concurrent program $H$ of two threads. To represent $P$, we make the following conventions. We have three global variables $x, y, z$, and a unique read event per variable. Event subscripts denote the variable accessed by the corresponding event. For each variable, we have a unique read event, and barred and unbarred events denote the good and bad write events, respectively, for that read event. Since we have specified the good-writes for each read event, the value function $val$ is not important for this example. Note also that $S(r_z) = 2$ (resp., $S(r_z) = 1$) since the good writes of $r_x$ (resp., $r_z$) are remote (resp., local) to the read event. The partial order $P$ of $\mathcal{P}$ consists of the thread orders of each thread, shown in solid lines in Figure 5a. The dashed edges of Figure 5a show the strengthening of $P$ performed by the algorithm $\text{Closure}$ (Algorithm 1). The numbers above the dashed edges denote both the order in which these orderings are added and the closure rule that is responsible for the corresponding ordering. In particular, algorithm $\text{Closure}$ performs the following steps.

1. Initially there are no dashed edges, and $r_x$ violates Item 1 of closure, as there is no good write event for $r_x$ that is ordered before $r_x$. Rule1 inserts an ordering $w_x \rightarrow r_x$ (dashed edge 1).
2. After the previous step, $r_y$ violates Item 2 of closure, as at this point, $r_y$ has only one maximal write event $\overline{w_y}$, which is bad for $r_y$. Rule2 inserts an ordering $r_y \rightarrow \overline{w_y}$ (dashed edge 2).
3. After the previous step, $r_z$ violates Item 3 of closure, as at this point, $r_z$ has a bad minimal write event $\overline{w_z}$ that is ordered before $r_z$ but not before any good write event. Rule3 inserts an ordering $\overline{w_z} \rightarrow w_z$ (dashed edge 3).

At this point no closure rule is violated, and $\text{Closure}$ returns the closure $Q = (X_1, X_2, Q, \text{val}, S, \text{GoodW})$ of $P$ where $P$ has been strengthened to $Q$ with the dashed edges. Observe that $Q$ has Mazurkiewicz width 2 (and not 1), as there still exist pairs of conflicting events that are unordered, both on variable $y$ and variable $z$. For example, there exist two write events on variable $y$ that are unordered, and hence there exist some linearizations that are “bad” in the sense that the read event $r_y$ does not observe the good write event $w_y$. Nevertheless, Lemma 4.1 guarantees
that the corresponding annotated partial order is linearizable to a valid trace, which is shown in Figure 5b. We make two final remarks for this example.

1) Not every linearization of Q produces a valid witness trace for the realizability of Q, as some linearizations violate the additional constraints that every read event must observe a write event that is good for the read event. Hence, the challenge is to find a correct witness.

2) Q has more than one witness of realizability. Figure 5b shows one such witness q, as constructed by Lemma 4.1. It is easy to verify that q is a valid witness. Due to Remark 2, the consistency of P guarantees that q is a valid trace of the program H.

Fig. 5. Figure 5a shows an annotated partial order P and its closure (dashed edges). (b) A witness trace that linearizes P.

5 VALUE-CENTRIC DYNAMIC PARTIAL ORDER REDUCTION

We now present our algorithm VC-DPOR for exploring the partitioning \( T^\text{max}_H \setminus \mathcal{VHB} \). Intuitively, the algorithm manipulates annotated partial orders of the form \( P = (X_1, X_2, P, S, \text{GoodW}) \), where \( X_1 \subseteq E_{P_1} \) and \( X_2 \subseteq E_{P_2} \), i.e., \( X_1 \) (resp., \( X_2 \)) contains events of the root thread (resp., leaf threads). We first introduce some useful concepts and then proceed with the main algorithm.

Trace extensions and inevitable sets. Given a trace t, an extension of t is a trace t’ such that t is a prefix of t’. We say that t’ is a maximal extension of t if t’ is an extension of t and t’ is maximal. A set of events X is inevitable for t if for every maximal extension t’ of t we have \( X \subseteq E(t’) \). A write extension of t, denoted by WExtend(t), is any arbitrary largest extension t’ of t such that \( E(t’) \setminus E(t) \subseteq \mathcal{W} \). In words, we obtain each t’ by extending t arbitrarily until (but not included) the next read event of each thread. Note that for every such write extension t’ of t, for every thread p, the local trace t’ \mid E(p) is unique, and the set E(t’) is inevitable for t. Let P be a closed annotated partial order over a set X. A set of events Y is inevitable for P if for every linearization t of P and every maximal extension t’ of t, we have that Y \( \subseteq E(t’) \).

Leaf refinement and minimal annotated partial orders. Consider two partial orders P, Q over a set X. We say that Q leaf-refines P, denoted by Q \( \sqsubseteq P \) if for every pair of events \( e_1, e_2 \in X \cap E_{P_1} \), if \( e_1 \prec e_2 \) and \( e_1 <_P e_2 \) then \( e_1 <_Q e_2 \). In words, Q leaf-refines P if Q agrees with P on the order of every pair of conflicting events that belong to leaf threads. Consider an annotated partial order \( P = (X_1, X_2, P, S, \text{GoodW}) \). We call P minimal if for every closed annotated partial order
We describe Extend. We rely on an algorithm called Extend to extend annotated partial orders with new events. Hence P does not contain any unnecessary orderings, given these two constraints. Observe that if P is minimal and K is the closure of P then K is also minimal. Afterwards, our algorithm VC-DPOR will use minimal annotated partial orders to represent different classes of the VHBB partitioning.

Algorithm Extend(P, X', val', S', GoodW'). Let P = (X_1, X_2, P, val, S, GoodW) be a minimal, closed annotated partial order, and X = X_1 ∪ X_2. Consider

1. a set X' with (i) X' \ X ⊆ W or |X' \ X| = 1 and (ii) X' is inevitable for P,
2. a value function val' over X' such that val ⊆ val',
3. a side function S' over X' such that S ⊆ S', and
4. a good-writes set GoodW' over X' such that GoodW ⊆ GoodW'.

We describe Extend for the special case where |X' \ X| = 1. When |X' \ X| = q > 1, Extend calls itself recursively for every annotated partial order of its output set on a sequence of sets Y_1, . . . , Y_q where Y_q = X', Y_0 = X and |Y_{i+1} \ Y_i| = 1. Let X' \ X = {e}.

1. If p(e) = p_1 (i.e., e belongs to the root thread), the algorithm simply constructs a partial order K over the set X' such that K|X = P and e' ≺_K e for every event e ∈ X' such that e' ≺_{TO} e. Afterwards, the algorithm constructs the annotated partial order K = (X'_1, X'_2, K, val', S', GoodW') and returns the singleton set A_ω = {Closure(K)}.
2. If p(e) ≠ p_1 (i.e., e belongs to the leaf threads), the algorithm first constructs a partial order K as in the previous item. Afterwards, it creates a new partial order K_i for every possible ordering of e with all events e' ∈ X_2 such that e ≺_{TO} e'. Finally, the algorithm constructs the annotated partial orders A = K_i = (X'_i, X'_2, K_i, val', S', GoodW'), and returns the set A = {Closure(K_i) : K_i ∈ A and Closure(K_i) ≠ ⊥}.

Causally-happens-before maps, guarding reads and candidate writes. A causally-happens-before (CHB) map is a map C : R → H → R ∪ {⊥, } such that for each read event r ∈ dom(C) and thread p ∈ H we have that C(r)(p) ∈ Rp ∪ {⊥, }. In words, C maps read events to functions that map every thread p ∈ H to a read event of p, or to some initial values {⊥, }. Given a trace t and an event e ∈ E(t), we define the guarding read Guard_t(e) of e in t as the last read event of p(e) that happens before e in t, and Guard_t(e) = ⊥ if no such read event exists. Formally,

\[\text{Guard}_t(e) = \max\{r ∈ R(t[p(e)]) : r ≺_{TO} e\}\]
where we take the maximum of the empty set to be $\perp$. Given a trace $t$, a CHB map $C$ and a read event $r \in \text{enabled}(t)$, we define the candidate write set $M^C_t(r)$ of $r$ in $t$ given $C$ as follows:

$$M^C_t(r) = \{ w \in \mathcal{W}(t) : r \bowtie w \text{ and either Guard}_t(w) = \perp \text{ and } C(r(p(w))) = \perp \text{ or Guard}_t(w) \neq \perp \text{ and also } C(r(p(w))) \in \{\perp, \top\} \text{ or } C(r(p(w))) <_{\text{TO}} \text{Guard}_t(w) \}$$

We refer to Figure 6 for an illustration of the above notation. Intuitively, $C(r(p))$ encodes the prefix of the local trace of thread $p$ that contains write events which have already been considered by the algorithm as good writes for $r$. Instead of the whole prefix, we store the last read of that prefix. The two special values $\perp \perp$ and $\perp$ encode the empty prefix, and the prefix before the first read. The guarding read of a write $w$ is the last local read event the same thread that appears before $w$ in the execution so far. Hence, if the guarding read of $w$ appears before $C(r(p))$, we know that $w$ has been considered as a good write for $r$. The candidate write set for $r$ contains writes that are considered as good writes for $r$ in the current recursive step.

<table>
<thead>
<tr>
<th>$\tau_1$</th>
<th>$\tau_2$</th>
<th>$\tau_3$</th>
<th>$\tau_4$</th>
<th>$\tau_5$</th>
<th>$\tau_6$</th>
<th>$\tau_7$</th>
<th>$\tau_8$</th>
<th>$\tau_9$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$w_x$</td>
<td>$w_y$</td>
<td>$w_x$</td>
<td>$r_y$</td>
<td>$w_x$</td>
<td>$r_y$</td>
<td>$w_x$</td>
<td>$r_x$</td>
<td>$w_x$</td>
</tr>
</tbody>
</table>

(a) A trace $t$. Threads $p_1$ and $p_3$ have enabled events $r_1^3$ and $r_3^3$ (not shown), which access the variable $x$. The candidate write sets of the read events $r_1^3$ and $r_3^3$ given the causally-happens-before map $C$.

(b) The candidate write sets of the read events $r_1^3$ and $r_3^3$ given the causally-happens-before maps (Figure 6b). We denote by $e_i$ the $i$-th event of $t$.

**Algorithm 5: VC-DPOR($\mathcal{P} = (X_1, X_2, P, \text{val}, S, \text{GoodW}), C$)**

**Input:** A minimal closed annotated partial order $\mathcal{P}$, a CHB map $C$.

```plaintext
$1$ $t' \leftarrow \text{Realize}(\mathcal{P})$ // $\mathcal{P}$ is closed hence realizable
$2$ $t \leftarrow \text{WExtend}(t')$ // Extend $t'$ until before the next read of each thread
$3$ foreach $Q \in \text{Extend}(\mathcal{P}, E(t), \text{val}_t, S, \text{GoodW})$ do
   $4$ $C_Q \leftarrow C$ // Extensions of $\mathcal{P}$ to $E(t)$
   $5$ $\text{ExtendRoot}(Q, t, C_Q)$ // Create a copy of the CHB $C$
   $6$ foreach $p \in \mathcal{H} \setminus \{p_1\}$ do
      $7$ $\text{ExtendLeaf}(Q, t, C_Q, p)$ // Process the root thread
   $8$ end
$9$ end
```

**Algorithm VC-DPOR.** We are now ready to describe our main algorithm VC-DPOR for the enumerative exploration of the partitioning $\mathcal{T}_\mathcal{H}/\mathcal{VHB}$. The algorithm takes as input a minimal closed
annotated partial order $\mathcal{P}$ and a CHB map $\mathcal{C}$. First, VC-DPOR calls Realize to obtain a linearization $t'$ of $\mathcal{P}$ and constructs the write-extension $t$ of $t'$ which reveals new write events in $t$. Afterwards, the algorithm extends $\mathcal{P}$ to the set $\mathcal{E}(t)$ by calling Extend. Recall that Extend returns a set of minimal closed annotated partial orders. For every annotated partial order $Q$ returned by Extend, the algorithm calls ExtendRoot to process the read event of the root thread $p_1$ that is enabled in $t$. Finally, the algorithm calls ExtendLeaf for every leaf thread $p \neq p_1$ to process the read event of $p$ that is enabled in $t$. For the initial call, we construct an empty annotated partial order $\mathcal{P}$ and an initial CHB map $\mathcal{C}$ that for every read event $r \in \mathcal{R}$ and thread $p \in \mathcal{H}$ maps $\mathcal{C}(r)(p) = \{\bot\}$.

Algorithm 6: ExtendRoot($Q = (X_1, X_2, Q, val, S, GoodW), t, C_Q$)

<table>
<thead>
<tr>
<th>Input: A minimal closed annotated partial order $Q$, a trace $t$, a CHB map $C_Q$.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 $r \leftarrow$ enabled($t, p_1$) // The next enabled event in $p_1$ is a read</td>
</tr>
<tr>
<td>2 $Y_1 \leftarrow M^C_{Q_1}(r) \cap W_{p_1}$ // The set of local candidate writes of $r$</td>
</tr>
<tr>
<td>3 $Y_2 \leftarrow M^C_{Q_2}(r) \cap W_{p_1}$ // The set of remote candidate writes of $r$</td>
</tr>
<tr>
<td>4 foreach $i \in [2]$ do // $i = 1$ ($i = 2$) reads from local (remote) writes</td>
</tr>
<tr>
<td>5 $S_r \leftarrow S \cup {(r, i)}$ // The new side function</td>
</tr>
<tr>
<td>6 $D_r \leftarrow {\text{val}_i(w) : w \in Y_i}$ // The set of values of candidate writes of $r$</td>
</tr>
<tr>
<td>7 foreach $v \in D_r$ do // Every value $v$ that $r$ may read</td>
</tr>
<tr>
<td>8 $\text{val}_r \leftarrow \text{val}_i \cup {(r, v)}$ // The new value function</td>
</tr>
<tr>
<td>9 $\text{GoodW}_r \leftarrow \text{GoodW} \cup {(r, {w \in Y_i : \text{val}_i(w) = v})} // The new good-writes function</td>
</tr>
<tr>
<td>10 $\mathcal{K} \leftarrow \text{Extend}(Q_1, X_1) \cup \text{Extend}(Q_2, X_2) \cup {r, \text{val}_r, S_r, \text{GoodW}_r}$ // Returns one element</td>
</tr>
<tr>
<td>11 if $\mathcal{K} \neq \bot$ then // Extension is successful</td>
</tr>
<tr>
<td>12 Call VC-DPOR($\mathcal{K}, C_Q$) // Recurse</td>
</tr>
<tr>
<td>13 end</td>
</tr>
<tr>
<td>14 end</td>
</tr>
<tr>
<td>$C_Q(r) \leftarrow {(p, \text{max}_i((\mathcal{R}(t)[p])) : p \in \mathcal{H}}$ // The last read of each thread in $t$</td>
</tr>
</tbody>
</table>

Algorithm ExtendRoot. The algorithm takes as input a minimal closed annotated partial order $Q$, a trace $t$ and a CHB map $C_Q$, and attempts all possible extensions of $Q$ with the read event $r$ of $p_1$ that is enabled in $t$ to all possible values that are written in $t$. The algorithm first constructs two sets $Y_1$ and $Y_2$ which hold the local and remote, respectively, write events of $t$ that are candidate writes for $r$ according to the CHB map $C_Q$. Then, it iterates over the local ($i = 1$) and remote ($i = 2$) write choices for $r$ in $Y_i$. Finally, the algorithm (i) collects all possible values that $r$ may read from the set $Y_i$, (ii) constructs the appropriate new side function, value function and good-writes function, and (iii) calls Extend on these new parameters in order to establish the respective extension for $r$. For every such case, Extend returns a new minimal, closed annotated partial order $\mathcal{K}$ which is passed recursively to VC-DPOR.

Algorithm ExtendLeaf. The algorithm ExtendLeaf takes as input a minimal closed partial order $Q$, a trace $t$, a CHB map $C_Q$, and a thread $p \in \mathcal{H} \setminus \{p_1\}$. Similarly to ExtendRoot, ExtendLeaf attempts all possible extensions of $Q$ with the read event $r$ of $p$ that is enabled in $t$ to all possible values that are written in $t$. The main difference compared to ExtendRoot is that since $r$ belongs to a leaf thread, Extend returns a set of minimal, closed annotated partial orders (as opposed to just one) which result from all possible orderings of $r$ with the write events of $X_2$ that are conflicting with $r$. Then ExtendLeaf makes a recursive call to VC-DPOR for each such annotated partial order.

The following theorem states the main result of this paper.
We conclude with two remarks on space usage and the way lock events can be handled.

Remark 3 (Space complexity). To make our presentation simpler so far, VC-DPOR and ExtendLeaf iterate over the set of annotated partial orders returned by Extend, which can be exponentially large. An efficient variant of VC-DPOR shall explore these sets recursively, instead of computing all elements of each set imperatively. This results in polynomial space complexity for VC-DPOR.

Remark 4 (Handling locks). For simplicity of presentation, so far we have neglected locks. However, lock events can be incorporated naturally, as follows.

(1) Each lock-release event is a write event, writing an arbitrary value.

(2) Each lock-acquire event is a read event. Given two lock-acquire events $r_1, r_2$ the algorithm maintains that $\text{GoodW}(r_1) \cap \text{GoodW}(r_2) = \emptyset$.

VC-DPOR running example. Figure 7 illustrates the main aspects of VC-DPOR (Algorithms 5, 6, and 7) on a small example. We start with an empty annotated partial order $P$ and a CHB map $C$ that is empty (i.e., $C(r)(p) = \{\perp\}$ for every read event $r \in R$ and thread $p \in P$). The initial trace obtained in Line 1 of Algorithm 5 is $t' = \epsilon$. Its write-extension $t$ in Line 2 contains the three writes of $P_1$ and the first write of $P_2$. Next, Line 3 returns an annotated partial order $Q_a$ that corresponds to the thread order $\text{TO}(E(t))$. In $t$, the root thread $P_1$ has an enabled event (which is always a read), so ExtendRoot (Algorithm 6) is called on $Q_a$ and the (empty) CHB map $C_{Q_a}$.

The enabled read in Line 1 is $r^4_{P_1}$, its local candidate write (computed in Line 2) is $w^3_{P_1}$, and its remote candidate write (computed in Line 3) is $w^3_{P_2}$. This holds because $C_{Q_a}(r)(P_1) = \{(P_1, \perp), (P_2, \perp)\}$, which allows any write event to be observed. For the local (Line 4, $i = 1$) candidate $w^3_{P_1}$, first the side function is updated with $\{(r^4_{P_1}, 1)\}$ in Line 5. Then in Line 6, the only considered value is 1. Thus, in Line 8 the value function is updated with $\{(r^4_{P_1}, 1)\}$, and in Line 9 the good-writes
Thread $p_1$ : Thread $p_2$:
1. $w(y, 1)$  1. $w(x, 1)$
2. $w(y, 2)$  2. $r(x)$
3. $w(x, 1)$  3. $w(y, 2)$
4. $r(x)$  4. $r(y)$

(a) A concurrent program $\mathcal{H}$. (b) The VC-DPOR exploration tree.

Fig. 7. A program with two threads (Figure 7a) and the corresponding VC-DPOR exploration (Figure 7b).

function is updated with $\{(r^4_{p_1}, \{w^3_{p_1}\})\}$. Then, such an update is successfully realized in Line 10 by Extend, where the partial order is extended with $r^4_{p_1}$ and afterwards it is closed using algorithm Closure (Algorithm 1). Thus VC-DPOR (Algorithm 5) is recursively called on the corresponding annotated partial order $\mathcal{K}_a$ (and the empty CHB map $C_{\mathcal{Q}_a}$), and we proceed to the child $b$ of $a$.

In node $b$, no new event is added during the write-extension (Line 2), as $r^4_{p_1}$ is the last event of $p_1$, and in Line 3 we obtain $Q_b$. The only thread with an enabled read event is $p_2$, so ExtendLeaf (Algorithm 7) is called on $Q_b$ and $p_2$ (and empty CHB map $C_{\mathcal{Q}_b}$). The enabled read $r^5_{p_2}$ has candidate writes $w^3_{p_1}$ and $w^3_{p_2}$, both of which write the same value (c.f. Line 2), and hence the algorithm will allow $r^5_{p_2}$ to observe either. This is an example of the value-centric gains we obtain in this work.

In Line 4 the value function is updated with $\{(r^2_{p_2}, 1)\}$, and in Line 5 the good-writes function is updated with $\{(r^2_{p_2}, \{w^3_{p_1}, w^3_{p_2}\})\}$. The realization of this update happens in Line 6 by Extend, where the partial order is extended with $r^2_{p_2}$ and then closed using algorithm Closure (Algorithm 1). One annotated partial order $\mathcal{K}_b$ is returned and it is the argument of the further VC-DPOR call (with an empty CHB map $C_{\mathcal{Q}_b}$), we proceed to the child $c$ of $b$. In node $c$, the write-extension adds the event $w^3_{p_2}$, which, in similar steps as before, will lead to nodes $d$ and $e$.

Next, the recursion backtracks to the call of ExtendRoot in the node $a$ (†). The second iteration ($i = 2$) of the loop in Line 4 proceeds, where the remote candidate write $w^3_{p_2}$ is considered for $r^4_{p_1}$.

In a similar fashion, the descendants $f$, $g$, and $h$ are created and $h$ concludes with a maximal trace.

Finally, the recursion backtracks to the node $a$ again, where ExtendRoot (†) concludes with updating the CHB map as follows: $C_{\mathcal{Q}_a}(r^4_{p_1}) = \{(p_1, \bot), (p_2, \bot)\}$. The control-flow comes back to the initial VC-DPOR call (from Line 5), where the annotated partial order $\mathcal{Q}_a$ with the (now updated) CHB map $C_{\mathcal{Q}_a}$ is considered. The thread $p_2$ has an enabled read ($r^5_{p_2}$) in $t$, hence ExtendLeaf is called on $Q_a, C_{\mathcal{Q}_a}$, and $p_2$. Eventually, the descendants $i$, $j$, and $k$ are created and the exploration concludes. Note that in each of $i, j, k$, the thread $p_1$ has an enabled read $r^4_{p_1}$. However, note that Guard$_i(w^3_{p_1}) = Guard_i(w^3_{p_2}) = \bot$ and in all those nodes we have $C(r^4_{p_1})(p_1) = \{(p_1, \bot), (p_2, \bot)\}$, and thus $w^3_{p_1}$ and $w^3_{p_2}$ are never considered as candidate writes for $r^4_{p_1}$. This illustrates how VC-DPOR never explores the same class of $\mathcal{VHFB}$ twice.
6 EXPERIMENTS

We have seen in Theorem 3.1 that $\mathcal{VHB}$ is a coarse partitioning that can be explored efficiently by VC-DPOR. In this section we present an experimental evaluation of VC-DPOR on various classes of concurrent benchmarks, to assess

(1) the reduction of the trace-space partitioning achieved by $\mathcal{VHB}$, and
(2) the efficiency with which this partitioning is explored by VC-DPOR.

Implementation and experiments. To address the above questions, we have made a prototype implementation of VC-DPOR in the stateless model checker Nidhugg [Abdulla et al. 2015], which works on LLVM IR\(^2\). We have tested VC-DPOR on benchmarks coming in four classes:

(1) The TACAS Software Verification Competition (SV-COMP).
(2) Mutual-exclusion algorithms from the literature.
(3) Multi-threaded dynamic-programming algorithms that use memoization.
(4) Individual benchmarks that exercise various concurrency patterns.

Each benchmark comes with a scaling parameter, which is either the number of threads, or an unroll bound on all loops of the benchmark (often the unroll bound also controls the number of threads that are spawned.) We have compared our algorithm with three other state-of-the-art DPOR algorithms that are implemented in Nidhugg, namely Source [Abdulla et al. 2014], Optimal [Abdulla et al. 2014] and Optimal\(^*\) (“optimal with observers”) [Aronis et al. 2018], as well as our own implementation of DC-DPOR [Chalupa et al. 2017]. For our experiments, we have used a Linux machine with Intel(R) Xeon(R) CPU E5-1650 v3 @ 3.50GHz and 128GB of RAM. We have run Nidhugg with Clang and LLVM version 3.8. In all cases, we report the number of maximal traces and the total running time of each algorithm, subject to a timeout of 4 hours, indicated by “-“.

Implementation details. Here we clarify some details regarding our implementation.

(1) In our theory so far, we have neglected dynamic thread creation for simplicity of presentation. In practice, all our benchmarks spawn threads dynamically. This situation is handled straightforwardly, by including in our partial orders the orderings that are naturally induced by spawn and join events.

(2) The root thread is chosen as the first thread that is spawned from the main thread. We make this choice instead of the main thread as in many benchmarks, the main thread mainly spawns worker threads and performs only a few concurrent operations.

(3) In our presentation of $\text{Extend}(\mathcal{P}, X', \text{val}', S', \text{GoodW}')$, given $X' \setminus X = \{e\}$ such that $e$ belongs to a leaf thread, we consider all possible orderings of $e$ with conflicting events from all leaf threads. In our implementation, we relax this in two ways. Given a write event $e_w$, we say it is never-good if it does not belong to $\text{GoodW}'(r)$ for any read event $r$. Further, given $e_w$ and an annotated partial order $\mathcal{K}$, we say that $e_w$ is unobservable in $\mathcal{K}$, if for every linearization of $\mathcal{K}$ no read event can observe $e_w$. Given two unordered conflicting write events from leaf threads, we do not order them if (i) both are never-good, or (ii) at least one is unobservable.

Value-centric gains. As a preliminary experimental step, we explore the gains of our value-centric technique on small variants of the simple benchmark fib_bench from SV-COMP. This benchmark consists of a main thread and two worker threads, and two global variables $x$ and $y$. The first worker thread enters a loop in which it performs the update $x \leftarrow x + y$. Similarly, the second worker thread enters a loop in which it performs the update $y \leftarrow y + x$. To explore the sensitivity

\(^2\)Code accessible at \url{https://github.com/ViToSVK/nidhugg/tree/valuecentric_stable}
of our value-centric DPOR to values, we have created three variants $\text{fib\_bench}_1$, $\text{fib\_bench}_2$, $\text{fib\_bench}_3$ of the main benchmark. In variant $\text{fib\_bench}_i$ each worker thread performs the addition modulo $i$. Hence, the first and the second worker performs the update $x \leftarrow (x + y) \mod i$ and $y \leftarrow (y + x) \mod i$, respectively. For smaller values of $i$, we expect more write events to write the same value, and thus VC-DPOR to benefit both in terms of the traces explored and the running time. Although simple, this experiment serves the purpose of quantifying the value-centric gains of VC-DPOR in a controlled benchmark. Figure 8 depicts the obtained results for the three variants of $\text{fib\_bench}$, where Modulo = $\infty$ represents the original benchmark (i.e., without the modulo operation). We see that indeed, as $i$ gets smaller, VC-DPOR benefits significantly in both number of traces and running time. Moreover, this benefit gets amplified with higher unroll bounds.

**Benchmarks from SV-COMP.** Here we present experiments on benchmarks from SV-COMP (along the industrial benchmark parker) (Table 1). We have replaced all assertions with simple read events. This way we ensure a fair comparison among all algorithms in exploring the trace-space of each benchmark, as an assertion violation would halt the search. We have verified that all assertion violations present in these benchmarks are detected by all algorithms before this modification. The scaling parameter in each case controls the size of the input benchmark in terms of loop unrolls.

**Dynamic-programming benchmarks.** Here we present experiments on various multi-threaded dynamic-programming algorithms (Table 2). For efficiency, these algorithms use memoization to avoid recomputing instances that correspond to the same sub-problem. The benchmarks consist of three or four threads. In each case, all-but-one threads are performing the dynamic programming computation, and one thread reads a flag signaling that the computation is finished, as well as the result of the computation. Each benchmark name contains either the substring "td" or the substring "bu", denoting that the dynamic programming table is computed top-down or bottom-up, respectively. The scaling parameter of each benchmark controls the different sizes of the input problem. The dynamic programming problems we use as benchmarks are the following.

- **rod\_cut** computes, given one rod of a given length and prices for rods of shorter lengths, the maximum profit achievable by cutting the given rod.
- **lis** computes, given an array of non-repeating integers, the length of the longest increasing subsequence (not necessarily contiguous) in the array.
- **coin\_all** computes, given an unlimited supply of coins of given denominations, the total number of distinct ways to get a desired change.
<p>|</p>
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>VC-DPOR</th>
<th>Source</th>
<th>Optimal</th>
<th>Optimal</th>
<th>DC-DPOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>parker(6)</td>
<td>38670</td>
<td>1100917</td>
<td>1100917</td>
<td>1025567</td>
<td>985807</td>
</tr>
<tr>
<td>parker(7)</td>
<td>52465</td>
<td>1735432</td>
<td>1735432</td>
<td>1613807</td>
<td>155437</td>
</tr>
<tr>
<td>parker(8)</td>
<td>68360</td>
<td>2576147</td>
<td>2576147</td>
<td>2395947</td>
<td>2307467</td>
</tr>
<tr>
<td>27_Boop(6)</td>
<td>248212</td>
<td>3507969</td>
<td>3507969</td>
<td>4750426</td>
<td>1468774</td>
</tr>
<tr>
<td>27_Boop(7)</td>
<td>420033</td>
<td>-</td>
<td>-1014616</td>
<td>2874202</td>
<td>-</td>
</tr>
<tr>
<td>27_Boop(8)</td>
<td>677780</td>
<td>-</td>
<td>-2000352</td>
<td>5268064</td>
<td>-</td>
</tr>
<tr>
<td>30_Fun_Point(6)</td>
<td>5040</td>
<td>665280</td>
<td>665280</td>
<td>665280</td>
<td>665280</td>
</tr>
<tr>
<td>30_Fun_Point(7)</td>
<td>40320</td>
<td>1729728</td>
<td>1729728</td>
<td>1729728</td>
<td>1729728</td>
</tr>
<tr>
<td>30_Fun_Point(8)</td>
<td>362880</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>45_monabsex(5)</td>
<td>600</td>
<td>14400</td>
<td>14400</td>
<td>9745</td>
<td>6197</td>
</tr>
<tr>
<td>45_monabsex(6)</td>
<td>13152</td>
<td>518400</td>
<td>518400</td>
<td>291546</td>
<td>180126</td>
</tr>
<tr>
<td>45_monabsex(7)</td>
<td>423360</td>
<td>2540160</td>
<td>2540160</td>
<td>11170405</td>
<td>7073803</td>
</tr>
<tr>
<td>46_monabsex(5)</td>
<td>1064</td>
<td>14400</td>
<td>14400</td>
<td>5566</td>
<td>2653</td>
</tr>
<tr>
<td>46_monabsex(6)</td>
<td>21371</td>
<td>518400</td>
<td>518400</td>
<td>1571717</td>
<td>62864</td>
</tr>
<tr>
<td>46_monabsex(7)</td>
<td>6219486</td>
<td>25401600</td>
<td>25401600</td>
<td>60575874</td>
<td>2057588</td>
</tr>
<tr>
<td>fk2012_true(3)</td>
<td>12400</td>
<td>42144</td>
<td>42144</td>
<td>42144</td>
<td>33886</td>
</tr>
<tr>
<td>fk2012_true(4)</td>
<td>252586</td>
<td>1217826</td>
<td>1217826</td>
<td>1217826</td>
<td>888400</td>
</tr>
<tr>
<td>fk2012_true(5)</td>
<td>3757922</td>
<td>24580886</td>
<td>24580886</td>
<td>16494444</td>
<td>-</td>
</tr>
<tr>
<td>fkp2013_true(5)</td>
<td>17751</td>
<td>86400</td>
<td>86400</td>
<td>48591</td>
<td>25626</td>
</tr>
<tr>
<td>fkp2013_true(6)</td>
<td>519977</td>
<td>3628800</td>
<td>3628800</td>
<td>1672915</td>
<td>786499</td>
</tr>
<tr>
<td>fkp2013_true(7)</td>
<td>20043857</td>
<td>-</td>
<td>-</td>
<td>3242410</td>
<td>-</td>
</tr>
<tr>
<td>nondet-array(4)</td>
<td>404</td>
<td>2616</td>
<td>2616</td>
<td>688</td>
<td>592</td>
</tr>
<tr>
<td>nondet-array(5)</td>
<td>10804</td>
<td>128760</td>
<td>128760</td>
<td>18865</td>
<td>15449</td>
</tr>
<tr>
<td>nondet-array(6)</td>
<td>430000</td>
<td>9854640</td>
<td>9854640</td>
<td>711276</td>
<td>571476</td>
</tr>
<tr>
<td>pthread-det(7)</td>
<td>327782</td>
<td>4027216</td>
<td>4027216</td>
<td>4027216</td>
<td>829168</td>
</tr>
<tr>
<td>pthread-det(8)</td>
<td>2457572</td>
<td>43976774</td>
<td>43976774</td>
<td>49964234</td>
<td>-</td>
</tr>
<tr>
<td>pthread-det(9)</td>
<td>15858126</td>
<td>-</td>
<td>-</td>
<td>59287740</td>
<td>-</td>
</tr>
<tr>
<td>reorder(5)</td>
<td>1016</td>
<td>1755360</td>
<td>1755360</td>
<td>68206</td>
<td>4978</td>
</tr>
<tr>
<td>reorder(6)</td>
<td>247684</td>
<td>-</td>
<td>-</td>
<td>437725</td>
<td>-</td>
</tr>
<tr>
<td>reorder(9)</td>
<td>1446716</td>
<td>-</td>
<td>-</td>
<td>1792290</td>
<td>220m35s</td>
</tr>
<tr>
<td>skull_true(3)</td>
<td>3532</td>
<td>617706</td>
<td>617706</td>
<td>436413</td>
<td>172931</td>
</tr>
<tr>
<td>skull_true(4)</td>
<td>8990</td>
<td>2732933</td>
<td>2732933</td>
<td>1840022</td>
<td>656100</td>
</tr>
<tr>
<td>skull_true(5)</td>
<td>19981</td>
<td>9488043</td>
<td>9488043</td>
<td>6076898</td>
<td>1988798</td>
</tr>
<tr>
<td>sigma_false(7)</td>
<td>12509</td>
<td>135135</td>
<td>135135</td>
<td>30952</td>
<td>30952</td>
</tr>
<tr>
<td>sigma_false(8)</td>
<td>133736</td>
<td>2027025</td>
<td>2027025</td>
<td>325488</td>
<td>325488</td>
</tr>
<tr>
<td>sigma_false(9)</td>
<td>1625040</td>
<td>-</td>
<td>-</td>
<td>3845724</td>
<td>3845724</td>
</tr>
<tr>
<td>check_bad_arr(5)</td>
<td>40146</td>
<td>12838</td>
<td>12838</td>
<td>10989</td>
<td>6689</td>
</tr>
<tr>
<td>check_bad_arr(6)</td>
<td>187473</td>
<td>357368</td>
<td>357368</td>
<td>3070979</td>
<td>187377</td>
</tr>
<tr>
<td>check_bad_arr(7)</td>
<td>1856332</td>
<td>8245810</td>
<td>8245810</td>
<td>6943293</td>
<td>4069592</td>
</tr>
<tr>
<td>32_pthread(1)</td>
<td>20</td>
<td>246</td>
<td>24</td>
<td>246</td>
<td>20</td>
</tr>
<tr>
<td>32_pthread(2)</td>
<td>1470</td>
<td>1890</td>
<td>1890</td>
<td>1806</td>
<td>1470</td>
</tr>
<tr>
<td>32_pthread(3)</td>
<td>226800</td>
<td>302400</td>
<td>302400</td>
<td>280800</td>
<td>226800</td>
</tr>
<tr>
<td>fkp2014_true(2)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>fkp2014_true(3)</td>
<td>10986</td>
<td>1098</td>
<td>1098</td>
<td>1098</td>
<td>1098</td>
</tr>
<tr>
<td>fkp2014_true(4)</td>
<td>207024</td>
<td>207024</td>
<td>207024</td>
<td>207024</td>
<td>207024</td>
</tr>
<tr>
<td>singleton(8)</td>
<td>2</td>
<td>40320</td>
<td>40320</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>singleton(9)</td>
<td>2</td>
<td>362880</td>
<td>362880</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>singleton(10)</td>
<td>2</td>
<td>362880</td>
<td>362880</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>stack_true(9)</td>
<td>48620</td>
<td>48620</td>
<td>48620</td>
<td>48620</td>
<td>48620</td>
</tr>
<tr>
<td>stack_true(10)</td>
<td>184756</td>
<td>184756</td>
<td>184756</td>
<td>184756</td>
<td>184756</td>
</tr>
<tr>
<td>stack_true(11)</td>
<td>705432</td>
<td>705432</td>
<td>705432</td>
<td>705432</td>
<td>705432</td>
</tr>
<tr>
<td>ticket_lock(2)</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>ticket_lock(3)</td>
<td>204</td>
<td>204</td>
<td>204</td>
<td>204</td>
<td>204</td>
</tr>
<tr>
<td>ticket_lock(4)</td>
<td>41100</td>
<td>41400</td>
<td>41400</td>
<td>41400</td>
<td>41400</td>
</tr>
</tbody>
</table>

- **coin_min** computes, given an unlimited supply of coins of given denominations, the minimum number of coins required to get a desired change.
- **bin_nocon** computes the number of binary strings of a given length that do not contain the substring '11'.

Table 1. Experimental comparison on SV-COMP benchmarks.

---

- **Proc. ACM Program. Lang., Vol. 3, No. OOPSLA, Article 124. Publication date: October 2019.**
Table 2. Experimental comparison on dynamic-programming benchmarks.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Maximal Traces</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>VC-DPOR</td>
<td>Source Optimal</td>
</tr>
<tr>
<td></td>
<td>VC-DPOR</td>
<td>Source Optimal</td>
</tr>
<tr>
<td>rod_cut_td3(7)</td>
<td>4324</td>
<td>102128</td>
</tr>
<tr>
<td>rod_cut_td3(8)</td>
<td>14744</td>
<td>508646</td>
</tr>
<tr>
<td>rod_cut_td3(9)</td>
<td>50320</td>
<td>2574772</td>
</tr>
<tr>
<td>rod_cut_td4(3)</td>
<td>1478</td>
<td>91592</td>
</tr>
<tr>
<td>rod_cut_td4(4)</td>
<td>21358</td>
<td>2459640</td>
</tr>
<tr>
<td>rod_cut_td4(5)</td>
<td>433371</td>
<td></td>
</tr>
<tr>
<td>rod_cut_td4(6)</td>
<td>19933</td>
<td>183516</td>
</tr>
<tr>
<td>rod_cut_td4(7)</td>
<td>99622</td>
<td>1101084</td>
</tr>
<tr>
<td>rod_cut_td4(8)</td>
<td>498061</td>
<td>6606492</td>
</tr>
<tr>
<td>rod_cut_bu3(6)</td>
<td>1901</td>
<td>33912</td>
</tr>
<tr>
<td>rod_cut_bu3(7)</td>
<td>74541</td>
<td>2246424</td>
</tr>
<tr>
<td>rod_cut_bu3(8)</td>
<td>3007476</td>
<td></td>
</tr>
<tr>
<td>lis_bu3(8)</td>
<td>118812</td>
<td>1744064</td>
</tr>
<tr>
<td>lis_bu3(9)</td>
<td>368400</td>
<td>7001792</td>
</tr>
<tr>
<td>lis_bu3(10)</td>
<td>1137</td>
<td>18522</td>
</tr>
<tr>
<td>coin_all_td3(9)</td>
<td>4015</td>
<td>566214</td>
</tr>
<tr>
<td>coin_all_td3(10)</td>
<td>9052</td>
<td>2444048</td>
</tr>
<tr>
<td>coin_all_td3(11)</td>
<td>637859</td>
<td></td>
</tr>
<tr>
<td>coin_all_td4(2)</td>
<td>1137</td>
<td>18522</td>
</tr>
<tr>
<td>coin_all_td4(3)</td>
<td>29931</td>
<td>1024002</td>
</tr>
<tr>
<td>coin_all_td4(5)</td>
<td>1222278</td>
<td></td>
</tr>
<tr>
<td>coin_min_td3(9)</td>
<td>46335</td>
<td>1902262</td>
</tr>
<tr>
<td>coin_min_td3(10)</td>
<td>154662</td>
<td></td>
</tr>
<tr>
<td>coin_min_td3(11)</td>
<td>1312252</td>
<td></td>
</tr>
<tr>
<td>coin_min_td4(2)</td>
<td>9012</td>
<td>1470312</td>
</tr>
<tr>
<td>coin_min_td4(3)</td>
<td>68966</td>
<td></td>
</tr>
<tr>
<td>coin_min_td4(5)</td>
<td>379086</td>
<td></td>
</tr>
<tr>
<td>coin_min_td4(9)</td>
<td>46335</td>
<td>1902262</td>
</tr>
<tr>
<td>bin_nocon_td3(7)</td>
<td>1137</td>
<td>18522</td>
</tr>
<tr>
<td>bin_nocon_td3(8)</td>
<td>44802</td>
<td></td>
</tr>
<tr>
<td>bin_nocon_td3(11)</td>
<td>9912</td>
<td>1470312</td>
</tr>
<tr>
<td>bin_nocon_bu3(6)</td>
<td>52500</td>
<td>773122</td>
</tr>
<tr>
<td>bin_nocon_bu3(7)</td>
<td>262500</td>
<td>5411854</td>
</tr>
<tr>
<td>bin_nocon_bu4(3)</td>
<td>1312500</td>
<td></td>
</tr>
</tbody>
</table>

Mutual-exclusion benchmarks. Here we present experiments on various mutual-exclusion algorithms from the literature (Table 3). In particular, we use the two-thread solutions of Dijkstra [Dijkstra 1983], Kessels [Kessels 1982], Tsay [Tsay 1998], Peterson [Peterson 1981], Peterson-Fischer [Peterson and Fischer 1977], Szymanski [Szymanski 1988], Dekker [Knuth 1966], as well as various solutions of Correia-Ramalhete [Correia and Ramalhete 2016]. In addition, we use the two-thread and three-thread versions of Burns’s algorithm [Burns and Lynch 1980]. These protocols exercise a wide range of communication patterns, based, e.g., on the number of shared variables and the number of sequentially consistent stores/loads required to enter/leave the critical section. In all these benchmarks, each thread executes the corresponding protocol to enter a (empty) critical section a number of times, the latter controlled by the scaling parameter.

Individual benchmarks. Here we present experiments on individual benchmarks (Table 4): eratosthenes consists of two threads computing the sieve of Eratosthenes in parallel; redundant_co consists of three threads, two of which repeatedly write to a variable and one reads from it; float_read consists of several threads, each writing once to a variable, and one reading from it (adapted from [Aronis et al. 2018])); opt_lock consists of three threads in an optimistic-lock scheme. The scaling parameter controls the size in terms of loop unrolls.
Summary. For the sake of completeness, we refer to Table 5 for some statistics on our benchmark set. Entries marked with “U” denote that the corresponding parameter is controlled by the unroll bound of the respective benchmarks. In a variety of cases, the \(VHB\) partitioning is significantly coarser than each of the partitionings constructed by the other algorithms. This coarseness makes VC-DPOR more efficient in its exploration than the alternatives. We note that in some cases, \(VHB\) offers little-to-no reduction, and then VC-DPOR becomes slower than the alternatives, due to the overhead incurred in constructing \(VHB\). For example, for the benchmark reorder_5 of Table 1, the partitioning reduction achieved by VC-DPOR is large enough compared to Source, Optimal
Table 4. Experimental comparison on individual benchmarks.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>VC-DPOR</th>
<th>Source</th>
<th>Optimal</th>
<th>Optimal*</th>
<th>DC-DPOR</th>
<th>VC-DPOR</th>
<th>Source</th>
<th>Optimal</th>
<th>Optimal*</th>
<th>DC-DPOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>eratosthenes(5)</td>
<td>3500</td>
<td>1527736</td>
<td>27585</td>
<td>19991</td>
<td>16.92s</td>
<td>18m37s</td>
<td>20m39s</td>
<td>41.4s</td>
<td>1m29s</td>
<td></td>
</tr>
<tr>
<td>eratosthenes(7)</td>
<td>29320</td>
<td>-</td>
<td>253792</td>
<td>189653</td>
<td>3m37s</td>
<td>-</td>
<td>-</td>
<td>9m29s</td>
<td>19m41s</td>
<td></td>
</tr>
<tr>
<td>eratosthenes(8)</td>
<td>110380</td>
<td>938756</td>
<td>710551</td>
<td>-</td>
<td>11m29s</td>
<td>-</td>
<td>-</td>
<td>42m27s</td>
<td>1h4m</td>
<td></td>
</tr>
<tr>
<td>redundant_co(2)</td>
<td>11</td>
<td>1969110</td>
<td>1969110</td>
<td>27858</td>
<td>16.92s</td>
<td>18m37s</td>
<td>20m39s</td>
<td>41.14s</td>
<td>1m29s</td>
<td></td>
</tr>
<tr>
<td>redundant_co(8)</td>
<td>35</td>
<td>-</td>
<td>1118305</td>
<td>35937</td>
<td>0.09s</td>
<td>-</td>
<td>-</td>
<td>13m24s</td>
<td>0.97s</td>
<td></td>
</tr>
<tr>
<td>redundant_co(9)</td>
<td>39</td>
<td>-</td>
<td>1778221</td>
<td>50653</td>
<td>0.07s</td>
<td>-</td>
<td>-</td>
<td>23m49s</td>
<td>1.35s</td>
<td></td>
</tr>
<tr>
<td>float_read(9)</td>
<td>9</td>
<td>3628800</td>
<td>3628800</td>
<td>2305</td>
<td>0.05s</td>
<td>-</td>
<td>-</td>
<td>26m30s</td>
<td>26m38s</td>
<td></td>
</tr>
<tr>
<td>float_read(15)</td>
<td>15</td>
<td>-</td>
<td>245761</td>
<td>16</td>
<td>0.65s</td>
<td>-</td>
<td>-</td>
<td>3m52s</td>
<td>0.97s</td>
<td></td>
</tr>
<tr>
<td>float_read(16)</td>
<td>16</td>
<td>-</td>
<td>524289</td>
<td>17</td>
<td>1.42s</td>
<td>-</td>
<td>-</td>
<td>9m25s</td>
<td>1.44s</td>
<td></td>
</tr>
<tr>
<td>redundnat_co(2)</td>
<td>2497</td>
<td>69252</td>
<td>69252</td>
<td>11982</td>
<td>1.50s</td>
<td>X2Tv6</td>
<td>15.10s</td>
<td>15.53s</td>
<td>3.25s</td>
<td></td>
</tr>
<tr>
<td>redundnat_co(3)</td>
<td>80805</td>
<td>15036174</td>
<td>15036174</td>
<td>416850</td>
<td>52.13s</td>
<td>X2Tv3</td>
<td>1h5m</td>
<td>1h9m</td>
<td>2m9s</td>
<td></td>
</tr>
<tr>
<td>redundnat_co(9)</td>
<td>2543298</td>
<td>-</td>
<td>14038926</td>
<td>6743831</td>
<td>37m41s</td>
<td>X2Tv2</td>
<td>1h27m</td>
<td>1h2m</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5. Benchmark statistics.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Loc</th>
<th>Var</th>
<th>Locks</th>
<th>Threads</th>
<th>Benchmark</th>
<th>Loc</th>
<th>Var</th>
<th>Locks</th>
<th>Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>parker</td>
<td>134</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td>48_ticket_lock</td>
<td>52</td>
<td>3</td>
<td>1</td>
<td>U</td>
</tr>
<tr>
<td>27_Boop</td>
<td>74</td>
<td>4</td>
<td>0</td>
<td>4</td>
<td>rod_cut_td3</td>
<td>50</td>
<td>51</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>30_Fun_Point</td>
<td>67</td>
<td>1</td>
<td>1</td>
<td>U</td>
<td>rod_cut_td4</td>
<td>62</td>
<td>51</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>45_monabsex</td>
<td>24</td>
<td>1</td>
<td>0</td>
<td>U</td>
<td>rod_cut_bus3</td>
<td>36</td>
<td>51</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>46_monabsex</td>
<td>22</td>
<td>2</td>
<td>0</td>
<td>U</td>
<td>rod_cut_bus4</td>
<td>37</td>
<td>51</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>fkt2012_true</td>
<td>100</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>lis_bus4</td>
<td>48</td>
<td>51</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>fkt2013_true</td>
<td>26</td>
<td>1</td>
<td>0</td>
<td>U</td>
<td>lis_bus4</td>
<td>48</td>
<td>51</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>nondet-de</td>
<td>56</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>coin_all_td3</td>
<td>25</td>
<td>51</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>reorder-5</td>
<td>1227</td>
<td>4</td>
<td>0</td>
<td>U</td>
<td>coin_min_td3</td>
<td>46</td>
<td>51</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>scull_true</td>
<td>389</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>coin_min_td4</td>
<td>52</td>
<td>51</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>sigma_false</td>
<td>36</td>
<td>1</td>
<td>0</td>
<td>U</td>
<td>bin_nocon_td3</td>
<td>43</td>
<td>101</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>check_bad_arr</td>
<td>33</td>
<td>1</td>
<td>0</td>
<td>U</td>
<td>bin_nocon_bus3</td>
<td>53</td>
<td>101</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>32_pthread5</td>
<td>87</td>
<td>4</td>
<td>1</td>
<td>U</td>
<td>tay</td>
<td>54</td>
<td>3</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>fkt2014_true</td>
<td>36</td>
<td>2</td>
<td>1</td>
<td>U</td>
<td>peter_fisch</td>
<td>59</td>
<td>3</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>singleton</td>
<td>43</td>
<td>1</td>
<td>0</td>
<td>U</td>
<td>peterson</td>
<td>68</td>
<td>4</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>stack_true</td>
<td>104</td>
<td>1</td>
<td>2</td>
<td>U</td>
<td>lamport</td>
<td>83</td>
<td>5</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

and Optimal* that makes VC-DPOR significantly faster than DC-DPOR. However, although the partitioning of VC-DPOR is smaller than DC-DPOR, the corresponding reduction is not large enough to make VC-DPOR faster than DC-DPOR in this benchmark (in general, VC-DPOR has a larger polynomial overhead than DC-DPOR.) Similarly, for the benchmark X2Tv9 of Table 3, the reduction of the VHB partitioning is quite small, and although Source is the slowest algorithm in theory, its more lightweight nature makes it faster in practice for this benchmark. Finally, we also identify benchmarks such as stack_true and 48_ticket_lock where there is no trace reduction at all, and are better handled by existing methods. We note that our approach is fairly different from the literature, and our implementation of VC-DPOR still largely unoptimized. We identify potential for improving the performance of VC-DPOR by improving the closure computation, as well as reducing (or eliminating) the number of non-maximal traces explored by the algorithm.

7 RELATED WORK AND CONCLUSIONS

The formal analysis of concurrent programs is a major challenge in verification, and has been a subject of extensive research [Cadiou and Lévy 1973; Clarke et al. 1986; Farzan and Kincaid 2012; Farzan and Madhusudan 2009; Lal and Reps 2009; Lipton 1975; Petri 1962]. Since it is hard to reproduce bugs by testing due to scheduling nondeterminism, systematic state space exploration by model checking is an important approach for the problem [Alglave et al. 2013; Andrews et al. 2004; Clarke et al. 1999a; Godefroid 2005; Musuvathi and Qadeer 2007]. In this direction, stateless
model checking has been employed to combat state-space explosion [Godefroid 1996, 1997, 2005; Madan Musuvathi 2007].

To deal with the exponential number of interleavings faced by the early model checking [Godefroid 1997], several reduction techniques have been proposed such as POR and context bounding [Musuvathi and Qadeer 2007; Peled 1993]. Several POR methods, based on persistent set [Clarke et al. 1999b; Godefroid 1996; Valmari 1991] and sleep set techniques [Godefroid 1997], have been studied. DPOR techniques were first proposed in [Flanagan and Godefroid 2005], and several variants and improvements have been made since [Lauterburg et al. 2010; Saarikivi et al. 2012; Sen and Agha 2006, 2007; Tasharofi et al. 2012]. In [Abdulla et al. 2014], source sets and wakeup trees were developed to make DPOR optimal, and the underlying computational problems were further studied in [Nguyen et al. 2018]. Besides the present work, further improvements over optimal DPOR have been made in [Aronis et al. 2018; Chalupa et al. 2017], as well as with maximal causal models [Huang 2015; Huang and Huang 2017]. Other techniques such as unfoldings have also been explored [Kähkönen et al. 2012; McMillan 1995; Rodriguez et al. 2015]. Techniques for POR have also been applied to relaxed memory models [Abdulla et al. 2015; Demsky and Lam 2015; Huang and Huang 2016; Kokologiannakis et al. 2017; Wang et al. 2008] and message passing programs [Godefroid 1996; Godefroid et al. 1995; Katz and Peled 1992].

In this work, we have introduced a new equivalence on traces, called the value-happens-before equivalence $V^B\text{HB}$, which considers the values of trace events in order to determine whether two traces are equivalent. We have shown that $V^B\text{HB}$ is coarser than the standard happens-before equivalence, which is the theoretical foundation of the majority of DPOR algorithms. In fact, this coarsening occurs even when there are no concurrent write events. In addition, we have developed an algorithm VC-DPOR that relies on $V^B\text{HB}$ to partition the trace space into equivalence classes and explore each class efficiently. Our experiments show that, in a variety of benchmarks, $V^B\text{HB}$ indeed produces smaller partitionings than those explored by alternative, state-of-the-art methods, which often leads to a large reduction in running times.

ACKNOWLEDGMENTS

The authors would also like to thank anonymous referees for their valuable comments and helpful suggestions. This work is supported by the Austrian Science Fund (FWF) NFN grants S11407-N23 (RiSE/SHiNE) and S11402-N23 (RiSE/SHiNE), by the Vienna Science and Technology Fund (WWTF) Project ICT15-003, and by the Austrian Science Fund (FWF) Schrödinger grant J-4220.

REFERENCES


Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas. 2015. Stateless Model Checking for TSO and PSO. In TACAS.


Doron Peled. 1993. All from One, One for All: On Model Checking Using Representatives. In *CAV*.


