Efficient Verification of Program Fragments: *Eager POR* ⋆

Patrick Metzler, Habib Saissi, Péter Bokor, Robin Hesse, and Neeraj Suri

Technische Univeristät Darmstadt,
{metzler, saissi, pbokor, hesse, suri}@deeds.informatik.tu-darmstadt.de

Abstract. Software verification of concurrent programs is hampered by an exponentially growing state space due to non-deterministic process scheduling. Partial order reduction (POR)-based verification has proven to be a powerful technique to handle large state spaces.
In this paper, we propose a novel dynamic POR algorithm, called *Eager POR* (EPOR), that requires considerably less overhead during state space exploration than existing algorithms. EPOR is based on a formal characterization of program fragments for which exploration can be scheduled in advance and dependency checks can be avoided. We show the correctness of this characterization and evaluate the performance of EPOR in comparison to existing state-of-the-art dynamic POR algorithms. Our evaluation shows substantial improvement in the runtime performance by up to 91%.

Keywords: model checking, partial order reduction, concurrent programs, formal verification

1 Introduction

Automated verification of concurrent programs is known to be a hard problem [13]. The non-determinism of scheduling results in an exponential number of possible interleavings that need to be systematically explored by a program verifier. By constraining the considered class of properties, for instance to deadlock and local state reachability, POR techniques [10] attempt to tackle this problem by reducing the number of interleavings to be explored. A dependency relation between transitions gives raise to equivalence classes of executions, referred to as *Mazurkiewicz traces* [8], such that it is sufficient for a program verifier to explore only one representative per Mazurkiewicz trace.

The effectiveness of POR approaches relies on the precision of the dependency relation. In the original POR approaches, dependencies are calculated statically leading to an inaccurate over-approximation. Dynamic partial order reduction approaches [1,3,6] tighten the precision of the dependency relation by considering only dependencies occurring at runtime, leading to a less redundant exploration.

⋆ To appear at ATVA 2016. The final publication will be available at `link.springer.com`.

Process 1:	Process 2:	Process 3:
t_1: write x	t_2: read x	t_3: read x

Fig. 1: Readers-writers benchmark with one writer and two readers.

While exploring the state space of a program, dynamic POR algorithms identify pairs of dependent transitions which additionally need to be explored in reversed order so that all Mazurkiewicz traces are covered. Such pairs of transitions constitute a *reversible race* [1]. In order to detect all reversible races of a system, a dynamic POR algorithm checks for each transition whether it constitutes a race with any previous transition in the current path. During each such race check, the algorithm needs (often multiple times) to check whether two transitions are dependent. Therefore, dependency checks constitute a large part of any dynamic POR algorithm's runtime overhead.

In this paper, we propose *Eager POR* (EPOR), an optimization of dynamic POR algorithms such as SDPOR [1] that significantly reduces the number of dependency checks. EPOR *eagerly* creates schedules to bundle dependency checks for sequences of transitions instead of checking dependencies in every visited state. These sequences, called *sections*, correspond to program fragments of one or more statements of each process. By checking races in a section only once, many additional race checks and dependency checks can be avoided. A new constraint system-based representation of Mazurkiewicz traces ensures that all reversible races inside a section are explored in both orderings. As a result, EPOR requires significantly fewer dependency checks compared to other DPOR algorithms where dependencies are checked after the execution of every transition.

Contributions. Our contributions are threefold. (1) We introduce a general optimization of POR algorithms that explores program fragments, called sections. We formally model section-based exploration by a constraint system representation of Mazurkiewicz traces and proof its correctness. (2) We present a dynamic POR algorithm called EPOR that enables efficient verification of concurrent programs against local state properties and deadlocks. EPOR shows how to extend existing POR algorithms with section-based exploration. Finally, (3) we implement and evaluate EPOR using well established benchmarks written in a simplified C-like programming language.

2 Motivating Example

As a motivating example, consider the Readers-Writers benchmark in Figure 1 (also used in [1, 3]). Process 1 writes to the shared variable x (t_1), Processes 2 and 3 read from x (t_2 and t_3). The dynamic dependencies for all states are $D = \{(t_1, t_2), (t_2, t_1), (t_1, t_3), (t_3, t_1)\}$; the operations t_2 and t_3 are commutative (do not constitute a race), while both t_1, t_2 and t_1, t_3 are non-commutative, (constitute a race).

Our approach is based on the observation that the set of all Mazurkiewicz traces of program fragments as in the Readers-writers example can be calculated without exploring any program states and checking for races between operations only once. The program of Figure 1 has 4 (Mazurkiewicz) traces and the dynamic POR algorithm SDPOR [1] explores one execution per trace. Each execution consists of 3 events, hence SDPOR performs 3 race checks per execution (each time an operation is appended to the current partial execution, a check is performed whether the current operation constitutes a race with any previous operation of the current partial execution). Each race check consists of several dependency checks (in order to decide whether e_1 and e_2 constitute a race, pairwise dependencies need to be determined for all events that occur between e_1 and e_2). In total, SDPOR performs 12 race checks and 25 dependency checks.

By exploiting the fact that all executions consist of the same operations and contain the same races, it is possible to reduce the number of race checks to 3 and the number of dependency checks to 8: after exploring an arbitrary execution of the program, we know that each execution consists of t_1, t_2, and t_3 and contains the races $(t_1, t_2), (t_1, t_3)$ (either in this or in reversed order), which can be determined using 3 race checks. We construct four partial orders $\{(t_1, t_2), (t_1, t_3)\}$, $\{(t_2, t_1), (t_1, t_3)\}$, $\{(t_1, t_2), (t_3, t_1)\}$, and $\{(t_2, t_1), (t_3, t_1)\}$, which correspond to the four traces of the program. By computing a linear extension of each partial order, we obtain an execution of each trace. In Section 3.2, we explain how to generalize this idea to systems with dynamic dependencies.

3 Constraint System-based POR

3.1 System Model

This section introduces basic notions about the system model and notations used throughout the rest of this paper.

We write $u = a_1 \dots a_n$ for the sequence consisting of the elements $a_1, \dots, a_n$ and define $range(u) := \{1, \dots, n\}$. The empty sequence is denoted by ε. Concatenation of a sequence u and a sequence v or an element t is written as $u \cdot v$ or $u \cdot t$, respectively. For $i \in range(u)$, we define $u[i] := a_i$, $l[\dots i] := a_1 \dots a_i$, and $l[i \dots] := a_i \dots a_n$. We model concurrent programs as *transition systems* $TS = (PID, S, s_0, T)$, where PID is a finite set of process identifiers, S is a finite set of states, $s_0 \in S$ is the initial state of the system, and T is a finite set of transitions such that

- each transition $t \in T$ is mapped to a unique process identifier $pid(t) \in PID$
- for all $t \in T$, $t : S \rightharpoonup S$ (transitions are partial functions from S to S), where we write $t \in enabled(s)$ if t is defined at s
- for all $s_1, \dots, s_{n+1} \in S$ and any finite sequence $t_1 \dots t_n \in T$ such that $t_i(s_i) = s_{i+1}$, $s_1 \neq s_{n+1}$ (the state graph is acyclic)
- transitions do not disable other transitions:

$$\forall t, t' \in T. \forall s, s' \in S. \, s \xrightarrow{t} s' \land t' \in enabled(s) \land t' \notin enabled(s') \Rightarrow t = t'$$

– transitions do enable only transitions from the same process: $\forall t, t' \in T. \forall s, s' \in S. s \xrightarrow{t} s' \wedge t' \notin \mathit{enabled}(s) \wedge t' \in \mathit{enabled}(s') \Rightarrow \mathit{pid}(t) = \mathit{pid}(t')$
– at most one transition per process is enabled at a given state: $\forall s \in S. \forall t, t' \in T. \mathit{pid}(t) = \mathit{pid}(t') \wedge t, t' \in \mathit{enabled}(s) \Rightarrow t = t'$

To require that transitions do not disable other transitions simplifies the presentation but is not a general limitation as distinguishing between the termination and temporary blocking of a process would obviate the need for this restriction. A similar restriction is used in [1]. Acyclicity restricts our method to terminating programs in favor of a stateless exploration.

For the rest of this paper, we assume that there is an arbitrary transition system $\mathit{TS} = (\mathit{PID}, S, s_0, T)$ which models a concurrent program under analysis. Where not otherwise mentioned, we refer to this transition system.

Paths in the state graph of TS correspond to (partial) executions of the program modeled by TS. We represent such paths as transition sequences $t_1 \ldots t_n$ for some $t_1, \ldots, t_n \in T$. We write $s_1 \xrightarrow{t_1 \ldots t_n} s_{n+1}$ if there exist states $s_2, \ldots, s_{n+1} \in S$ such that $s_i \xrightarrow{t_i} s_{i+1}$ for all $1 \leq i \leq n$, i.e., $t_1 \ldots t_n$ corresponds to a path in the state graph of TS. Furthermore, if $s_1 \xrightarrow{u} s_2$ for some states s_1, s_2 and a transition sequence u, we write $u(s_1)$ to denote the state s_2 and call u a feasible sequence at s_1, written $u \in \mathit{feasible}(s_1)$.

A particular occurrence of a transition in a transition sequence is called an *event*. In a transition sequence $u = t_1 \ldots t_n$ feasible at s_0, we represent an event t_i by its index i in u.

We distinguish between data dependencies and dependencies caused by the program control flow of a process. The latter is modeled by a *program order* for TS, which is a partial order $\mathit{PO} \subseteq T \times T$ such that $\forall (t_1, t_2) \in \mathit{PO}. \mathit{pid}(t_1) = \mathit{pid}(t_2)$ (PO only relates transitions of the same process) and $\forall t, t' \in T. \forall s, s' \in S. s \xrightarrow{t} s' \wedge t' \notin \mathit{enabled}(s) \wedge t' \in \mathit{enabled}(s') \Rightarrow (t, t') \in \mathit{PO}$ (transitions enable only transitions which are successors w.r.t. the program order) and $\forall (t, t') \in \mathit{PO}. \exists s, s' \in S. s \xrightarrow{t} s' \wedge t' \notin \mathit{enabled}(s) \wedge t' \in \mathit{enabled}(s')$ (two transitions are in relation w.r.t. the program order only if the first transitions enables the second transition). We write $t_1 <_{\mathit{PO}} t_2$ for $(t_1, t_2) \in \mathit{PO}$.

Dynamic data dependencies are modeled by a relation $D \subseteq T \times T \times S$ such that $\forall t_1, t_2 \in T. \forall s \in S. (t_1, t_2, s) \notin D \Rightarrow (t_1 \in \mathit{enabled}(s) \wedge t_2 \in \mathit{enabled}(s) \Rightarrow \exists s'. s \xrightarrow{t_1 t_2} s' \wedge s \xrightarrow{t_2 t_1} s')$. Furthermore, $\forall t_1, t_2 \in T. \forall s \in S. (t_1, t_2, s) \in D \Rightarrow (t_1, t_2) \notin \mathit{PO}$ (transitions in program order are not data dependent).

The combination of program order and data dependency gives rise to partial orders that characterize the Mazurkiewicz traces of TS. For transition sequences $v = t_1 \ldots t_n$ and v' feasible at some state $s = u(s_0)$, we represent the ordering induced by dynamic data dependencies as the sequence $\mathit{dep}(u, v)$, defined as the sequence that consists of the elements of $\{(i, j) : (t_i, t_j, t_1 \ldots t_{i-1}(s)) \in D \wedge i < j\}$ ordered with respect to $(i, j) < (i', j')$ if $i < i'$, or $i = i'$ and $j < j'$. We define Mazurkiewicz equivalence as $v \simeq v'$ if $\mathit{dep}(u, v) = \mathit{dep}(u, v')$.

For a given transition t and a state s, we write $\mathit{dependencies}(t, s) := \{t' : (t, t', s) \in D\}$ for the set of transitions that are dependent with t.

Process 0:	Process 1:	Process 2:
t_{00}: y := 0	t_{10}: **if** x[0] = 0	t_{20}: y := 1
t_{01}: x[y] := 1	t_{11}: **then** z := 1	

Fig. 2: A program with branchings.

As we use SDPOR as a basis to present EPOR, we adapt the corresponding definition of reversible races [1]. Two data dependent transitions t_i, t_j in some transition sequence $u = t_1 \ldots t_n$ feasible at s_0 constitute a reversible race, written $i \precsim_u j$, if there exists an equivalent sequence in which t_i and t_j are adjacent and dependent; formally, we define $i \precsim_u j \Leftrightarrow (i,j) \in \mathit{dep}(\varepsilon, u) \wedge \forall i < k < j. (i,k) \notin \mathit{dep}(\varepsilon, u) \vee (k,j) \notin \mathit{dep}(\varepsilon, u) \wedge t_j \in \mathit{enabled}(t_1 \ldots t_{i-1} t_{k_1} \ldots t_{k_m}(s_0))$, where $t_{k_1} \ldots t_{k_m}$ is the sequence $t_{i+1} \ldots t_{j-1}$ with all transitions removed that are neither data dependent nor in program order with t_j.

3.2 Exploring Programs in Sections

Requirements for Sections. As described in our motivating example (Section 2), EPOR requires only 3 instead of 12 race detections and only 8 instead of 25 dependency checks when exploring the Readers-Writers program. This reduction is possible because two conditions are met: every maximal transition sequence feasible at the initial state of Readers-Writers contains the same transitions and dependencies do not depend on states (it is possible to precisely calculate all dependencies statically).

In order to generalize our approach to arbitrary programs, we identify program fragments called *sections* where a generalization of these two conditions hold: (A) every execution of the section contains the same set of events and (B) dependencies inside the section do not change during any execution of the section (it is possible to precisely calculate all dependencies of the section with the information given at the first state of the section). Once all traces for a section are explored, EPOR performs the same race checks as SDPOR in order to find races between events before and inside the current section.

Throughout this section, we use the program of Figure 2 as an example to explain conditions (A) and (B). Here, three processes work on the shared variables x, y, and z, where x is an array of length two. The statements labeled t_{00}, t_{01}, and t_{10} constitute a section. Including t_{11} in the same section would violate condition (A) and including t_{20} would violate condition (B), as detailed below.

In order to meet condition (A), we have to ensure that no transition is enabled in one trace of a section while it is disabled in another trace of the section. For this, we define *branching transitions* as transitions which enable different program order successors depending on the state it is executed in:

$$\begin{aligned} &\mathit{branching}(t) :\Leftrightarrow \exists s, s' \in S. t \in \mathit{enabled}(s) \wedge t \in \mathit{enabled}(s') \wedge \\ &\quad (\mathit{enabled}(t(s)) \setminus \mathit{enabled}(s)) \neq (\mathit{enabled}(t(s')) \setminus \mathit{enabled}(s')). \end{aligned}$$

For the example of Figure 2, the statement t_{11} cannot be part of the same section as t_{10} because t_{10} is a branching transition and t_{11} is a program order successor of t_{10}.

As long as sections do not contain any branching transition together with one of its program order successors, condition (A) is satisfied. To see this, assume that there exists a transition sequence u in a section such that u becomes unfeasible when transformed to u' by swapping only transitions that are not in program order relation. Let t_1 be the first transition in u that is not enabled at the corresponding state in u'. Since transitions cannot disable other transitions by definition, there exists some transition t_2 that occurs before t_1 in u and enables t_1 in u but does not enable t_1 in u'. We have $t_2 <_{PO} t_1$, hence t_2 occurs before t_1 in u' as well. Transition t_2 is enabled in both u and u' because t_1 is the first transition not enabled in u'. Since t_2 enables different transitions depending on the state it is executed in, it is a branching transition, contradiction.

A section satisfies condition (B) if all of its traces contain the same set of dependencies or, equivalently, if the dependencies inside the section can be determined at the first state of the section. This condition holds if swapping two dependent transitions inside a section does not influence whether following transitions are dependent. We characterize such a pair of dependent transitions that influences following dependencies as *hiding dependency* so that the absence of hiding dependencies implies (B):

$$\begin{aligned} t_1 \xrightarrow{*}_s t_2 :\Leftrightarrow\ & \exists s_1, s_1', s_2, s_2' \in S.\, s \xrightarrow{t_1} s_1 \xrightarrow{t_2} s_1' \wedge s \xrightarrow{t_2} s_2 \xrightarrow{t_1} s_2' \\ & \wedge\ \mathit{dependencies}(t_2, s_1') \neq \mathit{dependencies}(t_2, s_2'). \end{aligned}$$

In the example of Figure 2, the statement t_{20} cannot be in the same section as statement t_{00} because they constitute a hiding dependency: the order in which t_{00} and t_{20} are executed influences the fact whether t_{01} and t_{10} are dependent and constitute a race.

A section which contains no hiding dependency trivially satisfies condition (B). Although dependencies inside of sections have to be independent of states inside the section, dynamic information about dependencies that is known at the beginning of a section can be accounted for. Therefore, EPOR makes use of all dynamic dependency information just as SDPOR.

Implementing Section Construction. In order to implement an algorithm that relies on sections, it is desirable to determine where the next section ends with only small overhead. Therefore, we present two static checks which detect branching transitions (in order to ensure condition (A)) and hiding dependencies (in order to ensure condition (B)).

When translating a program into a transition system, we statically classify all transitions that model a branching statement as a branching transition, where a branching statement is a statement with multiple program order successors, e.g., a conditional jump, an if-then-else construct, or a loop. This over-approximates the set of all branching transitions (for example, a conditional jump with an unsatisfiable condition would still be classified as a branching transition).

We prepare the check whether two transitions form a hiding dependency by a static dependency analysis. For each transition t, we calculate the set of program variables that can influence the address which is accessed by t. For each such variable, all transitions writing to the variable are marked as potentially influencing the address of t's memory access. Two transitions with disjoint sets of address-influencing transitions do not constitute a hiding dependency.

Constructing Mazurkiewicz Traces. Once transitions and the races of a section are known (e.g., by executing an arbitrary interleaving until the end of the current section), it is possible to calculate all Mazurkiewicz traces without calculating any further program states as follows. A Mazurkiewicz trace can be calculated by constructing a directed graph with statements as nodes and an edge between two statements t and t' whenever t should occur before t' in all representatives of the Mazurkiewicz trace. If the resulting graph is acyclic, it induces a partial order that directly corresponds to a Mazurkiewicz trace and any of its linear extensions is a representative of the Mazurkiewicz trace. Otherwise, the graph contains a cycle and there exists no execution that obeys the ordering of the graph.

For the example of Figure 2, we start by calculating a Mazurkiewicz trace of the section containing t_{00}, t_{01}, and t_{10}. We calculate the Mazurkiewicz trace where t_{01} occurs before t_{10} by defining the following graph:

$$t_{00} \xrightarrow{\text{po}} t_{01} \xrightarrow{\text{dep}} t_{10}$$

The edge (t_{00}, t_{01}) represents the program order of Process 1 and the edge (t_{01}, t_{10}) represents the (only) race of the section. Because the graph is acyclic, there exists a linear extension of the induced partial order, $t_{00}t_{01}t_{10}$, and we found a Mazurkiewicz trace of the program. By swapping the direction of the edge (t_{01}, t_{10}), we obtain a graph for another Mazurkiewicz trace where the race $t_{01} \precsim_{t_{00}t_{01}t_{10}} t_{10}$ is reversed. We do not swap the edge (t_{00}, t_{01}) because it represents the program order, which is obeyed by all executions.

A linear extension of the induced partial order can be constructed in linear time w.r.t. the number of nodes by iteratively removing a minimal node (a node with no incoming edge) and all its outgoing edges [11]. If no minimal node is found, the graph is cyclic.

By calculating Mazurkiewicz traces as described, it is possible to construct representatives of all Mazurkiewicz traces "in advance", i.e., without performing any (typically expensive) program state computations.

3.3 Formal Foundations of Trace Construction

This section formalizes the notions introduced in Section 3.2 and details how EPOR constructs Mazurkiewicz traces from a given transition sequence.

Section 3.2 describes sections as program fragments and specifies two conditions (A) and (B) they have to satisfy in order to support our POR algorithm. At the transition system level, we model a section as the set of transition sequences that correspond to an execution of the program fragment of the section. We write *section*(u), where u is feasible at s_0, for the set of transition sequences that are feasible at $u(s_0)$ and include exactly those transitions that model the statements of a section. Formally, *section*(u) includes all transition sequences

$v = t_1 \ldots t_k$ that are feasible at $u(s_0)$ and satisfy (where conditions (A) and (B) have been introduced informally in Section 3.2):

(A): for each branching transition t in v, no transition in program order with t follows t in v: $\forall 1 \leq i \leq k.\, \mathit{branching}(t_i) \Rightarrow \forall i < j \leq k.\, \neg t_i <_{PO} t_j$.

(B): v contains no hiding dependency: $\forall 1 \leq i \leq k.\, \forall i < j \leq k.\, \neg t_i \overset{*}{\rightarrow}_s t_j$, where $s = t_1 \cdot \ldots \cdot t_{i-1}(s_0)$.

- maximality: There is no transition t such that $v \cdot t$ satisfies the above requirements.

For some $\mathit{section}(u)$, a POR algorithm ideally explores only a subset $\mathit{section\text{-}rep}(u) \subseteq \mathit{section}(u)$ that contains exactly one representative of each Mazurkiewicz trace of the transition sequences in $\mathit{section}(u)$. In order to formalize the generation of $\mathit{section\text{-}rep}(u)$, we introduce *trace constraint systems*. Each satisfiable trace constraint system corresponds to the fragment of a Mazurkiewicz trace. The constraints of a trace constraint system in conjunction with the program order specify the fragment's partial order of events. By swapping those constraints, it is possible to reverse races and thereby generate all transition sequences of $\mathit{section\text{-}rep}(u)$ for some u.

Formally, a trace constraint system is a tuple $c = (A, C, l)$ where

- $A = \{1, \ldots, k\}$ for some k (the variables of c).
- C is a list of pairs $(i, j) \in A \times A$ (the constraints of c).
- $l : A \rightarrow T$ is a function which labels the elements of A with transitions.

If for a given transition sequence $v = t_1 \ldots t_n$ feasible at some $s = u(s_0)$ we have $k = n$, $l(i) = t_i$ for all $1 \leq i \leq n$, and $C = \mathit{dep}(u, v)$, we call c the trace constraint system of u at s and write $c = \mathit{CS}(u, v)$.

Given a state $u(s_0)$ for some transition sequence u, one can construct a transition sequence v from $\mathit{section}(u)$ by starting with $v = \varepsilon$ and iteratively adding transitions enabled at $u \cdot v(s_0)$ until adding another transition would violate one of the conditions (A) and (B). All remaining transition sequences of $\mathit{section\text{-}rep}(u)$ can subsequently be constructed by the use of trace constraint systems as follows. First, the trace constraint system $\mathit{CS}(u, v)$ that corresponds to the trace of v is constructed. Subsequently, all trace constraint systems which are equal to $\mathit{CS}(u, v)$ except for one or more swapped constraints are constructed. The set of these constraint systems is called $\mathit{traces}(u)$ and defined as

$$
\begin{aligned}
\mathit{traces}(u) := \{(\mathit{range}(v), C, l) : {} & \forall i \in \mathit{range}(v).\, l(i) = v[i] \\
& \wedge \mathit{range}(C) = \mathit{range}(\mathit{dep}(u, v)) \\
& \wedge \forall i \in \mathit{range}(C).\, (C[i] = \mathit{dep}(u, v)[i] \\
& \vee \exists \alpha_1, \alpha_2 \in \mathit{range}(v).\, (C[i] = (\alpha_2, \alpha_1) \wedge \mathit{dep}(u, v)[i] = (\alpha_1, \alpha_2)))\} \\
& \text{for some } v \in \mathit{section}(u).
\end{aligned}
$$

A solution v of a trace constraint system $c = (A, C, l)$, written $v \in \mathit{solutions}(c)$, is a transition sequence that (1) contains exactly the transitions that occur in the image of l and (2) obeys the constraints in C and

(3) respects the program order for the transitions they contain. Formally, we require for v that the following holds.

– There exists an injective (1-to-1) function $\sigma : A \rightarrow A$ such that $\forall(\alpha_1, \alpha_2) \in A. (\sigma(\alpha_1), \sigma(\alpha_2)) \in C \Rightarrow \alpha_1 \geq \alpha_2$ (σ respects the constraints C) and $\forall\alpha_1, \alpha_2 \in A. (l(\sigma(\alpha_1)) <_{PO} l(\sigma(\alpha_2)) \Rightarrow \alpha_1 \geq \alpha_2$ (σ respects the program order PO).
– $v = l(\sigma(1)) \cdots l(\sigma(n))$

We call c *satisfiable* if a solution of c exists. A solution of a satisfiable c can be constructed in linear time w.r.t. the number of transitions that are contained in c. For example, create a linear extension of the partial order induced by the union of the constraints of c and the program order for the transitions occurring in c. If this union contains cycles, c is not satisfiable, which is easily detected by a linear extension algorithm.

Using the notion of $traces(u)$, one can construct $section\text{-}rep(u)$ as a set that contains exactly one solution of each satisfiable trace constraint system in $traces(u)$. As each trace constraint system in $traces(u)$ is unique, only one representative of each trace of $section(u)$ is constructed, enabling an optimal POR exploration. Correctness of section-based exploration is provided by the following theorem; given two transition sequences v_1, v_2 in $section(u)$, there exists a constraint system c in $traces(u)$ whose solutions are equivalent to v_2.

Theorem 1 (Correctness of section-based exploration). $\forall u \in feasible(s_0). \forall v \in section(u). \exists c \in traces(u). \forall w \in solutions(c). w \simeq v$

Proof. Let $u \in feasible(s_0), v_1, v_2 \in section(u)$. Because of condition (A) in the definition of $section()$, v_1 and v_2 contain the same events (1). Because of condition (B) in the definition of $section()$, the same data dependencies appear in v_1 and v_2 ($D|_{dom(v_1)} = D|_{dom(v_2)}$) (2). Let $traces(u)$ be calculated on the basis of $CS(v_1)$; by definition, all constraint systems in $traces(u)$ contain exactly the transitions of $dom(v_1)$ and contain exactly one constraint for each data dependency in $D|_{dom(v_1)}$. Additionally, there exists a constraint system in $traces(u)$ for every ordering of races in $dom(v_1)$. Hence, and because of (1) and (2), there exists some $c \in traces(u)$ whose constraints correspond to the ordering of races in v_2. By the definition of $solutions()$, all transition sequences $w \in solutions(c)$ are linear extensions of the partial order induced by the constraints of c and the program order for $dom(v_1)$. Hence, $w \simeq v_2$.

3.4 The Algorithm: *Eager POR*

This section presents our algorithm EPOR. It is an extension of the SDPOR algorithm [1]. Instead of exploring single transitions at each recursive call, EPOR creates schedules for sections of the transition system under analysis. If no schedule is currently present, EPOR creates new schedules for all transition sequences in the section starting at the current state. If a schedule is present, it is used to guide the exploration. Checks for races inside a section are only performed once when schedules are created; checks for races between an event before the current section and an event inside the current section are still performed at every recursive call in order to ensure correctness.

As EPOR is based on SDPOR, we repeat basic definitions from SDPOR's pseudo code [1]. Let u be a transition sequence feasible at the initial state s_0. The next transition of a process p at some state $u(s_0)$ is denoted by $next_u(p)$ and $u \cdot p$ denotes $u \cdot next_u(p)$. For two processes p_1, p_2 with $t_1 = next_u(p_1), t_2 = next_u(p_2)$, we write $u \vDash p_1 \Diamond p_2$ to denote that t_1 and t_2 are independent, i.e., $(t_1, t_2, u(s_0)) \notin D$ and $(t_1, t_2) \notin PO$. Overloading the notation $enabled()$, we define $enabled(u) = \{p : \exists t \in enabled(u(s_0)).\, pid(t) = p\}$. For $v \in feasible(u(s_0))$, define $p \in I_u(v) \Leftrightarrow \exists v'.\, u \cdot v \simeq u \cdot p \cdot v'$. For event e in u, $pre(u, e)$ denotes the prefix of u up to but not including e and $notdep(u, e)$ denotes the subsequence of u that contains all events that occur after e in u but are not dependent with e in u.

The main routine $\mathsf{Explore}(u, \mathit{sec\text{-}start})$ takes as arguments a transition sequence u that identifies the current state of the transition system and an integer *sec-start* that identifies the index in u at which the last section of u starts. The initial call is $\mathsf{Explore}(\varepsilon, 0)$ so that the exploration starts at the initial state. EPOR uses three global variables *sleep*, *backtrack*, and *schedule*, which map a transition sequence to a set of processes. For some transition sequence u feasible at the initial state, $sleep(u)$ corresponds to the sleep set at state $u(s_0)$; $backtrack(u)$ holds processes whose transitions need to be explored at state $u(s_0)$ in order to reverse races between two events of different sections; $schedule(u)$ holds processes which are scheduled at state $u(s_0)$ in order to explore a section.

At some call $\mathsf{Explore}(u, \mathit{sec\text{-}start})$, EPOR first checks whether a deadlock is reached or u is sleep set-blocked (line 4). Subsequently, if no schedule for the current state is present, the subroutine Fill_Schedule calculates $section\text{-}rep(u)$ (as described in Section 3.3) and corresponding schedules (lines 6–8).

The loop in lines 10–15 explores any transitions of processes that are scheduled for the current state in order to explore a section. The subroutine Race_Detection checks whether there are reversible races between an event before the start of the current section (as specified in variable *sec-start*) and an event inside the current section. This avoids race checks between two events that are both inside the current section. For every reversible race that is found, the reversed race is scheduled for later exploration just as in the SDPOR algorithm.

Finally, the loop in lines 16–21 explores any transitions of processes that have been scheduled for the current state in order to reverse a race. Before the race check, the marker for the start of the current section is updated so that all reversible races in the current transition sequence are found.

Correctness. EPOR is correct in the sense that it explores a representative of every Mazurkiewicz trace that starts at s_0 and ends at a deadlock, which is expressed by the following theorem.

Theorem 2 (Correctness of EPOR). $\forall u \in feasible(s_0).\, \forall w \in feasible(u(s_0)).$ $\exists v.\, v \simeq w \wedge$ *Explore($u, length(u)$) calls Explore($v, \cdot$), i.e., v is explored*

Proof. By ind. on the ordering $\propto$ where $u_1 \propto u_2$ if $\mathsf{Explore}(u_1, \cdot)$ returned before $\mathsf{Explore}(u_2, \cdot)$ (as in [1]). Base case: trivial, as $feasible(u(s_0)) = \varnothing$. Inductive step:

```
1  initially: Explore(ε, 0)
2  global variables:
       sleep, backtrack, schedule = λu.∅
3  Explore(u, sec-start):
4  if (enabled(u) \ sleep(u)) = ∅ then
5    return
6  if schedule(u) = ∅ then
7    sec-start := length(u)
8    Fill_Schedule(u)
9  Done := ∅
10 while ∃p ∈ (schedule(u) \ Done) do
11   Race_Detection(u, sec-start, p)
12   sleep(u) := {p′ ∈ sleep(u) : u ⊨ p◊p′}
13   Explore(u · p, sec-start)
14   add p to Done
15   add p to sleep(u)
16 while ∃p ∈ (backtrack(u) \ sleep(u)) do
17   sec-start := length(u)
18   Race_Detection(u, sec-start, p)
19   sleep(u) := {p′ ∈ sleep(u) : u ⊨ p◊p′}
20   Explore(u · p, sec-start)
21   add p to sleep(u)
22
23 Fill_Schedule(u):
24 foreach v ∈ section-rep(u) do
25   foreach prefix v′ = e_1 ... e_n of v do
26     add pid(e_n) to schedule(u · v′)
27     sleep(u · v′) := {p′ ∈ sleep(u · v′) : u ⊨
           p◊p′}
28
29 Race_Detection(u, sec-start, p):
30 foreach e ∈ u[... sec-start] with
       e ≾_{u·p} next_u(p) do
31   u′ := pre(u, e)
32   v := notdep(u, e) · p
33   if I_{u′}(v) ∩ backtrack(u′) = ∅ then
34     add some p′ ∈ I_{u′}(v) to backtrack(u′)
```

Fig. 3: The EPOR algorithm.

By [1], it is sufficient to prove that $sleep(u)$ is a source set for $feasible(u)$. Indirectly assume that $\exists w \in feasible(u(s_0)).\, \forall p \in sleep(u).\, \forall v, w'.\, u \cdot w \cdot v \not\simeq u \cdot p \cdot w'$. Then there exists a race $i \precsim_{u \cdot p \cdot w'} j$ that distinguishes $u \cdot w \cdot v$ and $u \cdot p \cdot w'$. Case (1): i and j belong to different sections. EPOR in lines 11 and 18 performs the same backtracking as SDPOR, hence $\exists q \in sleep(u).\, q \in I_u(notdep(u \cdot p \cdot w', p))$. By the induction hypothesis, $\exists v_1, v_2.\, u \cdot w \cdot v_1 \simeq u \cdot q \cdot v_2$. ↯. Case (2): i and j belong to the same section $section(u')$ f.s. u'. By the definition of Fill_Schedule, $section\text{-}rep(u')$ is explored. By Theorem 1, $section\text{-}rep(u')$ contains a representative of every trace in $section(u')$. Hence, $\exists q \in sleep(u).\, q \in I_u(u \cdot w)$. ↯.

4 Implementation and Evaluation

We implemented EPOR and SDPOR in the Python programming language and ran it on multiple benchmark programs that are written in a simple imperative programming language where processes communicate over shared memory. We used sequential consistency as a memory model, which corresponds to total program orders. Two events are data dependent if one of the events writes to a memory location the other event either reads from or writes to. All experiments were run on 8 Intel i7-4790 CPUs at 3.60GHz with 16 GB main memory.

We use the runtime and the number of dependency checks as main metrics for the comparison of EPOR and SDPOR. A dependency check determines whether two events are in the dynamic dependency relation of the current transition system and is often performed several times in order to determine whether two events constitute a reversible race. The complete results can be found in ap-

Table 1: Comparison of EPOR and SDPOR on four well-known benchmarks.

Benchmark	Algorithm	Time(s)	Traces	Dep. Checks	Speedup(%)
Readers-Writers (9)	SDPOR	0.668	256	60885	—
Readers-Writers (9)	EPOR	0.400	256	3204	40.1
Readers-Writers (20)	SDPOR	6874.472	524288	1570045995	—
Readers-Writers (20)	EPOR	2728.742	524288	17827145	60.3
Indexer (12)	SDPOR	0.413	8	27072	—
Indexer (12)	EPOR	0.284	8	19325	31.2
Indexer (16)	SDPOR	13060.033	32768	1345407904	—
Indexer (16)	EPOR	7998.984	32805	466384458	38.8
Last Zero (6)	SDPOR	0.911	96	66384	—
Last Zero (6)	EPOR	0.724	96	29570	20.5
Last Zero (16)	SDPOR	*not terminating*			
Last Zero (16)	EPOR	18408.671	262144	7232899654	—
Shared Pointer (50)	SDPOR	32.529	101	14074966	—
Shared Pointer (50)	EPOR	17.398	101	11459539	46.5
Shared Pointer (100)	SDPOR	238.968	201	192707828	—
Shared Pointer (100)	EPOR	170.762	201	154590222	28.5

pendix A.A missing runtime indicates that the corresponding algorithm did not terminate for the given benchmark configuration within 35000 seconds (~ 9.7 hours) or required more than 16 GB of memory.

In Table 1, we present results for four benchmarks which have previously been used to evaluate dynamic POR algorithms. The Readers-Writers, Indexer, and Last Zero benchmarks are used in [1] to evaluate SDPOR; the Shared Pointer benchmark is borrowed from [6]. The Readers-Writers (N) benchmark contains a single writer and $N-1$ readers. The Indexer (N) benchmark consists of N processes that write to a shared hash table. It is the only benchmark presented here that contains hiding dependencies. The scheduling of an execution influences the control flow behaviour. The parameter of the Indexer benchmark specifies the number of processes. The Last Zero (N) benchmark consists of $N-1$ processes that update a shared array and an additional process that reads the same array. Again, the scheduling of an execution influences the control flow behaviour. The Shared Pointer (N) benchmark consists of two equal processes which execute a loop N times, followed by an update of the respective other's process pointer.

In all four benchmarks, EPOR shows a speed-up over SDPOR for the highest parameter. The number of dependency checks is always lower for EPOR than for SDPOR (except for Indexer (11), where no races occur), while the number of explored maximal transition sequences is equal between EPOR and SDPOR for all configurations.

```
Process PID:
  x[(PID+1)%l] := x[PID]
```

(a) Ring

```
Process PID:
  if x[PID] == 0 then
    x[(PID+1)%l] := 1
  if x[PID] == 0 then
    x[(PID+1)%l] := 1
```

(b) Branching

```
Process PID:
  x[(PID+1)%l] := x[PID]
  x[(PID+1)%l] := x[PID]
```

(c) Ring Extended

Fig. 4: Three artificial benchmarks (x is a global array of length l, a is a local variable. Each program statement is executed atomically.)

Table 2: Comparison of EPOR and SDPOR on two simple benchmarks.

Benchmark	Algorithm	Time(s)	Traces	Dep. Checks	Speedup(%)
Ring (17)	SDPOR	5984.174	131070	734642101	—
Ring (17)	EPOR	538.031	131070	2096753	91.0
Ring (19)	SDPOR		*not terminating*		
Ring (19)	EPOR	2884.695	524286	8653144	—
Branching (5)	SDPOR	1.180	311	145186	—
Branching (5)	EPOR	1.045	311	114640	11.4
Branching (11)	SDPOR	19068.490	318363	2200202598	—
Branching (11)	EPOR	8220.448	318978	1343673801	56.9

In order to investigate the performance of EPOR in special cases, we have designed two artificial benchmarks Ring and Branching, which are depicted in Figure 4b and 4a. They loosely resemble the communication of processes which communicate in a ring, for example as in a ring election protocol. Every line is executed atomically. The Branching benchmark consists of two branching statements and two assignments; whether the assignments are executed depends on the scheduling of a particular execution. In the Ring benchmark, each process likewise communicates with its next process, but without control flow branchings. The Ring benchmark is similar to the Readers-Writers benchmark, but shows a higher number of dependencies, as each process is both reading and writing. Selected results for these two benchmarks are depicted in Table 2.

For the Ring and Branching benchmarks, EPOR requires considerably less dependency checks than SDPOR for all configurations. The number of explored traces is equal for EPOR and SDPOR except for the Branching benchmark with 9 to 11 processes. The speed-up of EPOR over SDPOR is very prominent for the Ring benchmark; SDPOR does not terminate for 19 processes. Equally significantly, EPOR requires several orders of magnitude less dependency checks than SDPOR. For the Branching benchmark, EPOR still shows a considerable speed-up over SDPOR, however, the saving in terms of dependency checks is lower than for the Ring benchmark.

Table 3: Comparison of EPOR, EPOR-SH (short sections), and SDPOR on the Ring Extended benchmark.

Benchmark	Algorithm	Time(s)	Traces	Dep. Checks	Unsat. TCS	Speedup(%)
Ring Extended (6)	SDPOR	70.729	38466	7537485	0	—
Ring Extended (6)	EPOR	3412.561	38466	144095	16738750	-4724.8
Ring Extended (6)	EPOR-SH	72.869	38466	6747840	126	-3.0
Ring Extended (8)	SDPOR	6552.194	1548546	806537903	0	—
Ring Extended (8)	EPOR			*not terminating*		
Ring Extended (8)	EPOR-SH	5061.882	1548546	720212287	510	22.7

Less Unsatisfiable Trace Constraint Systems. Interestingly, EPOR shows a much higher runtime overhead than SDPOR for a slightly changed Ring benchmark as depicted in Figure 4c (Ring Extended). Here, each process repeats its assignment so that the program order is not empty as opposed to the Ring benchmark.

As will be detailed later, EPOR (in its original form) does not scale as well for this benchmark as for the benchmarks previously presented. We explain this by the fact that EPOR generates at most 2 unsatisfiable trace constraint systems for the previous benchmarks while the number of unsatisfiable trace constraint systems for the Ring Extended benchmark increases with the number of processes. These additional unsatisfiable constraint systems occur due to the dependency structure of the Ring Extended benchmark. Each process consists of two transitions, which model its two assignments. Each of these transitions depends on both transitions of the previous process and additionally on both transitions of the next process. Consequently, when combining the constraints of a trace constraint system for the Ring Extended benchmark with the program order between the two transitions of each process, a cycle occurs with considerably higher probability than it is the case for the Ring benchmark.

For program fragments with dense dependencies as in the Ring Extended benchmark, we propose an alternative definition of sections in order to reduce the generation of unsatisfiable trace constraint systems. Specifically, sections are shortened so that no trace constraint systems are generated whose constraints show cycles due to a combination with the program order. We call these adapted sections *short sections*. Cycles due to the program order can be avoided by permitting only one dependent transition per process inside a single short section. Formally, we define short sections by adding the following constraint to the definition of sections given in Section 3.3) such that all transition sequences $v = t_1 \ldots t_k \in \mathit{section}(u)$ additionally satisfy $\forall 1 \leq i, j, m, n \leq k.\, (i, j) \in \mathit{dep}(u, v) \wedge (m, n) \in \mathit{dep}(u, v) \wedge \mathit{pid}(t_i) = \mathit{pid}(t_m) \Rightarrow i = m$.

We have implemented the EPOR algorithm with short sections instead of sections, denoted by EPOR-SH, and compare it with EPOR and SDPOR on the Ring Extended benchmark. The observed numbers are shown in Table 3. For 6 processes, EPOR-SH still shows a considerable number of unsatisfiable constraint systems but reduces this number by more than 99% in comparison to EPOR with original sections. While EPOR is more than 47 times slower than SDPOR for 6 processes and does not terminate for 8 processes, EPOR-SH is only slightly slower than SDPOR for 6 processes and more than 22% faster than SDPOR for 8 processes. Hence, the overhead of generating the remaining unsatisfiable trace constraint systems is still small enough so that EPOR-SH outperforms SDPOR. Appendix A shows the performance of EPOR-SH on our remaining benchmarks.

In order to increase the robustness of EPOR, it is perceivable to dynamically adapt the section length to the dependency structure of the program. Additionally, we expect that the number of generated unsatisfiable trace constraint systems can be reduced by exploiting information about the infeasibility of a constraint system to prevent the generation of further trace constraint systems that contain the same cycle (with or without program order). Such optimizations would further improve the performance of EPOR and EPOR-SH.

5 Related Work

Static POR techniques use a static approximation of dependencies [2, 5, 10, 12]. While both static and dynamic POR algorithms can be augmented with section-based exploration as in EPOR, we focus on dynamic dependency calculation, which drastically increases the state space reduction for, e.g., Indexer benchmark.

Dynamic POR has been introduced by Flanagan and Godefroid [3]. Their algorithm DPOR computes a *persistent set* of transitions to explore in every visited state. Like many POR algorithms, DPOR has been combined with the *sleep set* technique [4]. For every visited state, the corresponding sleep set contains transitions whose exploration would be redundant and is avoided.

Abdulla, Aronis, Jonsson, and Sagonas have proposed two model checking algorithms based on DPOR [1], named SDPOR and ODPOR, replacing persistent sets with *source sets*. In some cases, the source set of a state is smaller than the smallest persistent set of this state, which improves the state graph reduction. EPOR uses source sets in order to reverse races between sections but avoids redundant race checks and source set calculations inside of sections.

The ODPOR algorithm is an extension of SDPOR that can increase the amount of state space reduction for certain benchmarks, however adding runtime overhead that is not always compensated by a higher state space reduction. In fact, for many benchmarks, SDPOR is faster than ODPOR due to less runtime overhead [1]. Consequently, we compare our algorithm EPOR to SDPOR instead of ODPOR in order to investigate whether even the lower runtime overhead of SDPOR can be reduced.

CDPOR by Gueta, Flanagan, Yahav, and Sagiv [6] handles sequences of transitions, similar to EPOR and unlike DPOR, SDPOR, and ODPOR. However, CDPOR explores only transitions of a single process at once, while EPOR handles transition sequences of all processes and of varying length.

POR approaches for relaxed memory models have been proposed, e.g., [15]. Our system model is able to handle systems with relaxed memory models by defining the program order accordingly. Symbolic model checking (both bounded and unbounded) using POR has been addressed, e.g., in [7,14]. We present EPOR as an improvement of dependency calculation in concrete-state dynamic POR algorithms. However, we see no fundamental difficulty in using it in symbolic POR algorithms as well.

6 Conclusion

We present section-based exploration, a dynamic POR approach that eagerly creates schedules for program fragments. In comparison to known dynamic POR algorithms, it avoids redundant race and dependency checks. We introduce trace constraint systems as a formalization of section-based exploration and prove its correctness. While our approach does not depend on a particular POR algorithm, we implement section-based exploration in EPOR and compare it to SDPOR. Our results show that EPOR is able to reduce the runtime overhead by up to 91% and increase the tractable program size.

References

1. Abdulla, P.A., Aronis, S., Jonsson, B., Sagonas, K.F.: Optimal dynamic partial order reduction. In: POPL. pp. 373–384. ACM (2014)
2. Bokor, P., Kinder, J., Serafini, M., Suri, N.: Supporting domain-specific state space reductions through local partial-order reduction. In: ASE. pp. 113–122. IEEE (2011)
3. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: POPL. pp. 110–121. ACM (2005)
4. Godefroid, P.: Using partial orders to improve automatic verification methods. In: CAV. LNCS, vol. 531, pp. 176–185. Springer (1990), http://dx.doi.org/10.1007/BFb0023731
5. Godefroid, P., Pirottin, D.: Refining dependencies improves partial-order verification methods (ext. abstr.). In: CAV. LNCS, vol. 697, pp. 438–449. Springer (1993)
6. Gueta, G., Flanagan, C., Yahav, E., Sagiv, M.: Cartesian partial-order reduction. In: SPIN. LNCS, vol. 4595, pp. 95–112. Springer (2007)
7. Kahlon, V., Wang, C., Gupta, A.: Monotonic partial order reduction: An optimal symbolic partial order reduction technique. In: CAV. LNCS, vol. 5643, pp. 398–413. Springer (2009)
8. Mazurkiewicz, A.W.: Trace theory. In: Advances in Petri Nets. LNCS, vol. 255, pp. 279–324. Springer (1986)
9. Metzler, P., Saissi, H., Bokor, P., Hesse, R., Suri, N.: Efficient verification of program fragments: Eager POR. Tech. rep., Technische Universität Darmstadt (2016)
10. Peled, D.: All from one, one for all: on model checking using representatives. In: CAV. LNCS, vol. 697, pp. 409–423. Springer (1993), http://dx.doi.org/10.1007/3-540-56922-7_34
11. Pruesse, G., Ruskey, F.: Generating linear extensions fast. SIAM 23(2), 373–386 (1994), http://dx.doi.org/10.1137/S0097539791202647

12. Valmari, A.: Stubborn sets for reduced state space generation. In: Applications and Theory of Petri Nets. LNCS, vol. 483, pp. 491–515. Springer (1989)
13. Valmari, A.: The state explosion problem. In: Lectures on Petri Nets I. LNCS, vol. 1491, pp. 429–528. Springer (1996), http://dx.doi.org/10.1007/3-540-65306-6_21
14. Wachter, B., Kroening, D., Ouaknine, J.: Verifying multi-threaded software with impact. In: FMCAD. pp. 210–217. IEEE (2013), http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=6679412
15. Zhang, N., Kusano, M., Wang, C.: Dynamic partial order reduction for relaxed memory models. In: PLDI. pp. 250–259. ACM (2015), http://doi.acm.org/10.1145/2737924.2737956

A Supplementary Appendix: Detailed Benchmark Results

The following table shows our complete experiment results for detailed reference. All benchmarks are parametric, where the parameter specifies the number of processes, except for the Shared Pointer benchmark, where it specifies the number of loop iterations. EPOR and EPOR-SH refer to our algorithm with sections as defined in Section 3.3 and short sections as defined in 4. Column *Unsat. TCS* refers to the number of unsatisfiable trace constraint systems generated by EPOR and EPOR-SH; column *Speedup* refers to the percentage-wise time saving over SDPOR.

Benchmark	Algorithm	Time(s)	Traces	Dep. Checks	Race Checks	Unsat. TCS	Speedup(%)
Readers-Writers (2)	SDPOR	0.001	2	3	2	0	0
Readers-Writers (2)	EPOR-SH	0.001	2	2	1	0	0.0
Readers-Writers (2)	EPOR	0.001	2	2	1	0	0.0
Readers-Writers (3)	SDPOR	0.002	4	28	12	0	0
Readers-Writers (3)	EPOR-SH	0.002	4	10	3	0	0.0
Readers-Writers (3)	EPOR	0.002	4	10	3	0	0.0
Readers-Writers (4)	SDPOR	0.005	8	148	47	0	0
Readers-Writers (4)	EPOR-SH	0.005	8	33	6	0	0.0
Readers-Writers (4)	EPOR	0.005	8	33	6	0	0.0
Readers-Writers (5)	SDPOR	0.015	16	607	153	0	0
Readers-Writers (5)	EPOR-SH	0.012	16	92	10	0	20.0
Readers-Writers (5)	EPOR	0.012	16	92	10	0	20.0
Readers-Writers (6)	SDPOR	0.041	32	2155	449	0	0
Readers-Writers (6)	EPOR-SH	0.030	32	236	15	0	26.8
Readers-Writers (6)	EPOR	0.030	32	236	15	0	26.8
Readers-Writers (7)	SDPOR	0.109	64	6969	1233	0	0
Readers-Writers (7)	EPOR-SH	0.072	64	578	21	0	33.9
Readers-Writers (7)	EPOR	0.072	64	578	21	0	33.9
Readers-Writers (8)	SDPOR	0.274	128	21107	3233	0	0
Readers-Writers (8)	EPOR-SH	0.172	128	1375	28	0	37.2
Readers-Writers (8)	EPOR	0.170	128	1375	28	0	38.0
Readers-Writers (9)	SDPOR	0.668	256	60885	8193	0	0
Readers-Writers (9)	EPOR-SH	0.403	256	3204	36	0	39.7
Readers-Writers (9)	EPOR	0.400	256	3204	36	0	40.1
Readers-Writers (10)	SDPOR	1.627	512	169111	20225	0	0
Readers-Writers (10)	EPOR-SH	0.934	512	7346	45	0	42.6
Readers-Writers (10)	EPOR	0.936	512	7346	45	0	42.5
Readers-Writers (11)	SDPOR	3.907	1024	455705	48897	0	0
Readers-Writers (11)	EPOR-SH	2.145	1024	16618	55	0	45.1
Readers-Writers (11)	EPOR	2.125	1024	16618	55	0	45.6

Continued on next page

Continued from previous page

Benchmark	Algorithm	Time(s)	Traces	Dep. Checks	Race Checks	Unsat. TCS	Speedup(%)
Readers-Writers (12)	SDPOR	9.231	2048	1197851	116225	0	0
Readers-Writers (12)	EPOR-SH	4.853	2048	37165	66	0	47.4
Readers-Writers (12)	EPOR	4.799	2048	37165	66	0	48.0
Readers-Writers (13)	SDPOR	21.675	4096	3083805	272385	0	0
Readers-Writers (13)	EPOR-SH	10.840	4096	82300	78	0	50.0
Readers-Writers (13)	EPOR	10.741	4096	82300	78	0	50.4
Readers-Writers (14)	SDPOR	50.985	8192	7799839	630785	0	0
Readers-Writers (14)	EPOR-SH	24.221	8192	180696	91	0	52.5
Readers-Writers (14)	EPOR	24.299	8192	180696	91	0	52.3
Readers-Writers (15)	SDPOR	116.479	16384	19429409	1445889	0	0
Readers-Writers (15)	EPOR-SH	54.318	16384	393794	105	0	53.4
Readers-Writers (15)	EPOR	54.015	16384	393794	105	0	53.6
Readers-Writers (16)	SDPOR	268.414	32768	47759395	3284993	0	0
Readers-Writers (16)	EPOR-SH	121.130	32768	852667	120	0	54.9
Readers-Writers (16)	EPOR	119.901	32768	852667	120	0	55.3
Readers-Writers (17)	SDPOR	608.308	65536	116031525	7405569	0	0
Readers-Writers (17)	EPOR-SH	264.130	65536	1835844	136	0	56.6
Readers-Writers (17)	EPOR	262.993	65536	1835844	136	0	56.8
Readers-Writers (18)	SDPOR	1361.840	131072	278986791	16580609	0	0
Readers-Writers (18)	EPOR-SH	582.379	131072	3933150	153	0	57.2
Readers-Writers (18)	EPOR	579.521	131072	3933150	153	0	57.4
Readers-Writers (19)	SDPOR	3076.191	262144	664600617	36896769	0	0
Readers-Writers (19)	EPOR-SH	1264.264	262144	8389770	171	0	58.9
Readers-Writers (19)	EPOR	1256.383	262144	8389770	171	0	59.2
Readers-Writers (20)	SDPOR	6874.472	524288	1570045995	81657857	0	0
Readers-Writers (20)	EPOR-SH	2738.353	524288	17827145	190	0	60.2
Readers-Writers (20)	EPOR	2728.742	524288	17827145	190	0	60.3
Indexer (11)	SDPOR	0.015	1	880	946	0	0
Indexer (11)	EPOR-SH	0.025	1	880	946	0	-66.7
Indexer (11)	EPOR	0.026	1	880	946	0	-73.3
Indexer (12)	SDPOR	0.413	8	27072	12825	0	0
Indexer (12)	EPOR-SH	0.274	8	19325	7961	0	33.7
Indexer (12)	EPOR	0.284	8	19325	7961	0	31.2
Indexer (13)	SDPOR	4.181	64	485600	106214	0	0
Indexer (13)	EPOR-SH	3.367	64	239590	74980	0	19.5
Indexer (13)	EPOR	3.506	64	239590	74980	0	16.1
Indexer (14)	SDPOR	49.120	512	5279831	1177634	0	0
Indexer (14)	EPOR-SH	42.644	512	2812237	795788	0	13.2
Indexer (14)	EPOR	44.144	512	2812237	795788	0	10.1
Indexer (15)	SDPOR	766.280	4096	79436769	16007293	0	0
Indexer (15)	EPOR-SH	556.283	4096	35103635	9347279	0	27.4
Indexer (15)	EPOR	576.093	4096	35103635	9347279	0	24.8
Indexer (16)	SDPOR	13060.033	32768	1345407904	251890633	0	0
Indexer (16)	EPOR-SH	7485.608	32805	466384458	116349641	0	42.7
Indexer (16)	EPOR	7998.984	32805	466384458	116349641	0	38.8
Last Zero (2)	SDPOR	0.002	2	9	13	0	0
Last Zero (2)	EPOR-SH	0.003	2	13	13	0	-50.0
Last Zero (2)	EPOR	0.003	2	8	10	0	-50.0
Last Zero (3)	SDPOR	0.013	6	197	128	0	0
Last Zero (3)	EPOR-SH	0.024	6	125	116	0	-84.6
Last Zero (3)	EPOR	0.012	6	80	84	0	7.7
Last Zero (4)	SDPOR	0.068	16	2065	709	0	0
Last Zero (4)	EPOR-SH	0.044	16	800	579	0	35.3
Last Zero (4)	EPOR	0.070	16	676	479	0	-2.9
Last Zero (5)	SDPOR	0.255	40	13613	2791	0	0
Last Zero (5)	EPOR-SH	0.173	40	4279	2371	0	32.2
Last Zero (5)	EPOR	0.195	40	4976	2120	0	23.5
Last Zero (6)	SDPOR	0.911	96	66384	10275	0	0
Last Zero (6)	EPOR-SH	0.633	96	19645	8480	0	30.5
Last Zero (6)	EPOR	0.724	96	29570	7885	0	20.5
Last Zero (7)	SDPOR	3.018	224	274999	33881	0	0
Last Zero (7)	EPOR-SH	2.142	224	79578	27720	0	29.0
Last Zero (7)	EPOR	2.517	224	147844	26234	0	16.6

Continued on next page

Continued from previous page

Benchmark	Algorithm	Time(s)	Traces	Dep. Checks	Race Checks	Unsat. TCS	Speedup(%)
Last Zero (8)	SDPOR	9.206	512	1109904	97439	0	0
Last Zero (8)	EPOR-SH	6.975	512	294877	85185	0	24.2
Last Zero (8)	EPOR	8.339	512	647298	80647	0	9.4
Last Zero (9)	SDPOR	22.350	1152	3836659	306046	0	0
Last Zero (9)	EPOR-SH	33.547	1152	1464128	314042	0	-50.1
Last Zero (9)	EPOR	33.950	1152	2884130	310058	0	-51.9
Last Zero (10)	SDPOR	108.007	2560	15149844	1160330	0	0
Last Zero (10)	EPOR-SH	94.648	2560	5405445	923038	0	12.4
Last Zero (10)	EPOR	95.582	2578	11544604	1015493	0	11.5
Last Zero (11)	SDPOR	264.036	5632	51558504	3325567	0	0
Last Zero (11)	EPOR-SH	197.799	5632	16019928	2410338	0	25.1
Last Zero (11)	EPOR	257.922	5632	40368624	2649056	0	2.3
Last Zero (12)	SDPOR	821.374	12288	175535648	9951180	0	0
Last Zero (12)	EPOR-SH	480.859	12288	41678637	5885987	0	41.5
Last Zero (12)	EPOR	705.437	12288	125302898	5950551	0	14.1
Last Zero (13)	SDPOR	2160.776	26624	565002531	29044732	0	0
Last Zero (13)	EPOR-SH	1361.417	26624	111575184	14917085	0	37.0
Last Zero (13)	EPOR	1441.852	26624	347226642	11989526	0	33.3
Last Zero (14)	SDPOR	8138.822	57344	1744754931	78289802	0	0
Last Zero (14)	EPOR-SH	3372.409	57344	300987594	37479306	0	58.6
Last Zero (14)	EPOR	3421.276	57344	1005154306	29966707	0	58.0
Last Zero (15)	SDPOR	17441.597	122880	4019531983	230194076	0	0
Last Zero (15)	EPOR-SH	6026.374	122880	514821851	93547034	0	65.4
Last Zero (15)	EPOR	6703.371	122880	1896719286	73740996	0	61.6
Last Zero (16)	SDPOR						
Last Zero (16)	EPOR-SH	19144.029	262144	1934932782	239409835	0	—
Last Zero (16)	EPOR	18408.671	262144	7232899654	179027187	0	—
Shared Pointer (10)	SDPOR	0.480	21	80395	32777	0	0
Shared Pointer (10)	EPOR-SH	0.896	21	61207	33655	0	-86.7
Shared Pointer (10)	EPOR	0.535	21	60546	33025	0	-11.5
Shared Pointer (20)	SDPOR	2.123	41	661981	225737	0	0
Shared Pointer (20)	EPOR-SH	4.226	41	528044	229295	0	-99.1
Shared Pointer (20)	EPOR	2.968	41	525351	226835	0	-39.8
Shared Pointer (30)	SDPOR	7.837	61	2374011	722897	0	0
Shared Pointer (30)	EPOR-SH	14.770	61	1932212	730935	0	-88.5
Shared Pointer (30)	EPOR	8.047	61	1923801	725445	0	-2.7
Shared Pointer (40)	SDPOR	17.013	81	6201931	1668257	0	0
Shared Pointer (40)	EPOR-SH	37.533	81	5060976	1682575	0	-120.6
Shared Pointer (40)	EPOR	13.508	81	5042257	1672855	0	20.6
Shared Pointer (50)	SDPOR	32.529	101	14074966	3205817	0	0
Shared Pointer (50)	EPOR-SH	125.372	101	11494347	3228215	0	-285.4
Shared Pointer (50)	EPOR	17.398	101	11459539	3213065	0	46.5
Shared Pointer (60)	SDPOR	52.435	121	27575051	5479577	0	0
Shared Pointer (60)	EPOR-SH	219.720	121	22323086	5511855	0	-319.0
Shared Pointer (60)	EPOR	43.751	121	22263258	5490075	0	16.6
Shared Pointer (70)	SDPOR	84.797	141	49302287	8633537	0	0
Shared Pointer (70)	EPOR-SH	370.194	141	39524860	8677495	0	-336.6
Shared Pointer (70)	EPOR	64.530	141	39430039	8647885	0	23.9
Shared Pointer (80)	SDPOR	84.948	161	83360055	12811697	0	0
Shared Pointer (80)	EPOR-SH	458.459	161	66218755	12869135	0	-439.7
Shared Pointer (80)	EPOR	95.521	161	66076608	12830495	0	-12.4
Shared Pointer (90)	SDPOR	143.694	181	128693768	18158057	0	0
Shared Pointer (90)	EPOR-SH	919.317	181	102871367	18230775	0	-539.8
Shared Pointer (90)	EPOR	132.781	181	102676446	18181905	0	7.6
Shared Pointer (100)	SDPOR	238.968	201	192707828	24816617	0	0
Shared Pointer (100)	EPOR-SH	1531.204	201	154847568	24906415	0	-540.8
Shared Pointer (100)	EPOR	170.762	201	154590222	24846115	0	28.5
Ring (2)	SDPOR	0.002	2	3	2	0	0
Ring (2)	EPOR-SH	0.001	2	2	1	0	50.0
Ring (2)	EPOR	0.001	2	2	1	0	50.0
Ring (3)	SDPOR	0.008	6	39	18	0	0
Ring (3)	EPOR-SH	0.005	6	11	3	2	37.5
Ring (3)	EPOR	0.005	6	11	3	2	37.5

Continued on next page

Continued from previous page

Benchmark	Algorithm	Time(s)	Traces	Dep. Checks	Race Checks	Unsat. TCS	Speedup(%)
Ring (4)	SDPOR	0.018	14	247	80	0	0
Ring (4)	EPOR-SH	0.022	14	43	6	2	-22.2
Ring (4)	EPOR	0.017	14	43	6	2	5.6
Ring (5)	SDPOR	0.064	30	1231	275	0	0
Ring (5)	EPOR-SH	0.045	30	139	10	2	29.7
Ring (5)	EPOR	0.047	30	139	10	2	26.6
Ring (6)	SDPOR	0.168	62	4932	813	0	0
Ring (6)	EPOR-SH	0.118	62	397	15	2	29.8
Ring (6)	EPOR	0.121	62	397	15	2	28.0
Ring (7)	SDPOR	0.459	126	17742	2283	0	0
Ring (7)	EPOR-SH	0.226	126	1038	21	2	50.8
Ring (7)	EPOR	0.298	126	1038	21	2	35.1
Ring (8)	SDPOR	1.297	254	59947	6275	0	0
Ring (8)	EPOR-SH	0.382	254	2540	28	2	70.5
Ring (8)	EPOR	0.710	254	2540	28	2	45.3
Ring (9)	SDPOR	3.530	510	191381	17288	0	0
Ring (9)	EPOR-SH	0.877	510	5577	36	2	75.2
Ring (9)	EPOR	1.635	510	5577	36	2	53.7
Ring (10)	SDPOR	8.967	1022	543438	44107	0	0
Ring (10)	EPOR-SH	3.418	1022	12281	45	2	61.9
Ring (10)	EPOR	2.919	1022	12281	45	2	67.4
Ring (11)	SDPOR	23.903	2046	1551020	116202	0	0
Ring (11)	EPOR-SH	8.452	2046	27769	55	2	64.6
Ring (11)	EPOR	6.020	2046	27769	55	2	74.8
Ring (12)	SDPOR	57.755	4094	4498596	299602	0	0
Ring (12)	EPOR-SH	18.373	4094	61507	66	2	68.2
Ring (12)	EPOR	17.331	4094	61507	66	2	70.0
Ring (13)	SDPOR	153.056	8190	12342751	752788	0	0
Ring (13)	EPOR-SH	34.668	8190	127345	78	2	77.3
Ring (13)	EPOR	40.175	8190	127345	78	2	73.8
Ring (14)	SDPOR	307.406	16382	36655573	2172569	0	0
Ring (14)	EPOR-SH	65.806	16382	261835	91	2	78.6
Ring (14)	EPOR	60.154	16382	261835	91	2	80.4
Ring (15)	SDPOR	731.446	32766	105588804	5623429	0	0
Ring (15)	EPOR-SH	143.513	32766	534423	105	2	80.4
Ring (15)	EPOR	145.635	32766	534423	105	2	80.1
Ring (16)	SDPOR	1782.207	65534	278381118	13318473	0	0
Ring (16)	EPOR-SH	327.465	65534	1084045	120	2	81.6
Ring (16)	EPOR	327.977	65534	1084045	120	2	81.6
Ring (17)	SDPOR	5984.174	131070	734642101	35656128	0	0
Ring (17)	EPOR-SH	708.740	131070	2096753	136	2	88.2
Ring (17)	EPOR	538.031	131070	2096753	136	2	91.0
Ring (18)	SDPOR						
Ring (18)	EPOR-SH	1542.738	262142	4167297	153	2	—
Ring (18)	EPOR	1062.553	262142	4167297	153	2	—
Ring (19)	SDPOR						
Ring (19)	EPOR-SH	3359.111	524286	8653144	171	2	—
Ring (19)	EPOR	2884.695	524286	8653144	171	2	—
Ring (20)	SDPOR						
Ring (20)	EPOR-SH	4454.283	1048574	9495364	190	2	—
Ring (20)	EPOR	4442.308	1048574	9495364	190	2	—
Ring (21)	SDPOR						
Ring (21)	EPOR-SH	13158.802	2097150	28329284	210	2	—
Ring (21)	EPOR	13084.234	2097150	28329284	210	2	—
Branching (2)	SDPOR	0.009	11	181	155	0	0
Branching (2)	EPOR-SH	0.009	11	174	147	0	0.0
Branching (2)	EPOR	0.008	11	142	124	1	11.1
Branching (3)	SDPOR	0.046	28	3169	1105	0	0
Branching (3)	EPOR-SH	0.055	28	2679	1124	0	-19.6
Branching (3)	EPOR	0.046	28	2206	943	1	0.0
Branching (4)	SDPOR	0.268	103	24945	6933	0	0
Branching (4)	EPOR-SH	0.308	103	21967	6960	0	-14.9
Branching (4)	EPOR	0.233	103	17296	5617	1	13.1

Continued on next page

Continued from previous page

Benchmark	Algorithm	Time(s)	Traces	Dep. Checks	Race Checks	Unsat. TCS	Speedup(%)
Branching (5)	SDPOR	1.180	311	145186	32384	0	0
Branching (5)	EPOR-SH	1.458	311	143461	34068	0	-23.6
Branching (5)	EPOR	1.045	311	114640	26926	1	11.4
Branching (6)	SDPOR	5.600	1010	796033	155629	0	0
Branching (6)	EPOR-SH	6.679	1010	809098	156745	0	-19.3
Branching (6)	EPOR	4.512	1010	645243	120540	1	19.4
Branching (7)	SDPOR	23.737	3165	3963738	665731	0	0
Branching (7)	EPOR-SH	29.320	3165	4153755	677854	0	-23.5
Branching (7)	EPOR	18.819	3165	3332731	505448	1	20.7
Branching (8)	SDPOR	111.485	10063	19677616	3051999	0	0
Branching (8)	EPOR-SH	124.574	10063	19995225	2827886	0	-11.7
Branching (8)	EPOR	76.783	10063	16091273	2042519	1	31.1
Branching (9)	SDPOR	588.386	31780	102640823	15619776	0	0
Branching (9)	EPOR-SH	835.651	31775	106250930	17043326	0	-42.0
Branching (9)	EPOR	444.051	30921	68635810	11463305	1	24.5
Branching (10)	SDPOR	3107.106	100651	516099474	79852841	0	0
Branching (10)	EPOR-SH	3832.897	100327	530295199	73161559	0	-23.4
Branching (10)	EPOR	1964.219	99920	325828401	48463434	1	36.8
Branching (11)	SDPOR	19068.490	318363	2200202598	358100829	0	0
Branching (11)	EPOR-SH	21970.231	316881	2091377423	284175909	0	-15.2
Branching (11)	EPOR	8220.448	318978	1343673801	179170034	1	56.9
Ring Extended (2)	SDPOR	0.003	6	41	34	0	0
Ring Extended (2)	EPOR-SH	0.004	6	38	29	0	-33.3
Ring Extended (2)	EPOR	0.004	6	9	6	10	-33.3
Ring Extended (3)	SDPOR	0.050	90	2264	1029	0	0
Ring Extended (3)	EPOR-SH	0.047	72	1553	663	14	6.0
Ring Extended (3)	EPOR	0.365	90	126	15	4006	-630.0
Ring Extended (4)	SDPOR	0.692	786	44477	14734	0	0
Ring Extended (4)	EPOR-SH	0.737	786	39708	12722	30	-6.5
Ring Extended (4)	EPOR	7.826	786	1632	28	64750	-1030.9
Ring Extended (5)	SDPOR	7.497	5730	631224	156322	0	0
Ring Extended (5)	EPOR-SH	7.754	5730	565678	138590	62	-3.4
Ring Extended (5)	EPOR	164.094	5730	16734	45	1042846	-2088.8
Ring Extended (6)	SDPOR	70.729	38466	7537485	1427204	0	0
Ring Extended (6)	EPOR-SH	72.869	38466	6747840	1285045	126	-3.0
Ring Extended (6)	EPOR	3412.561	38466	144095	66	16738750	-4724.8
Ring Extended (7)	SDPOR	608.836	247170	81503018	11900225	0	0
Ring Extended (7)	EPOR-SH	622.568	247170	72416459	10706749	254	-2.3
Ring Extended (8)	SDPOR	6552.194	1548546	806537903	94539059	0	0
Ring Extended (8)	EPOR-SH	5061.882	1548546	720212287	83761394	510	22.7
Ring Extended (8)	EPOR						