A Pragmatic Approach to Stateful Partial Order Reduction

Berk Cirisci[1][0000−0003−4261−090X], Constantin Enea[2][0000−0003−2727−8865], Azadeh Farzan[3][0000−0001−9005−2653], and Suha Orhun Mutluergil[4][0000−0002−0734−7969]

[1] IRIF, Université Paris Cité
cirisci@irif.fr
[2] LIX, Ecole Polytechnique, CNRS and Institut Polytechnique de Paris
cenea@lix.polytechnique.fr
[3] University of Toronto
azadeh@cs.toronto.edu
[4] Sabanci University
suha.mutluergil@sabanciuniv.edu

Abstract. Partial order reduction (POR) is a classic technique for dealing with the state explosion problem in model checking of concurrent programs. Theoretical optimality, i.e., avoiding enumerating equivalent interleavings, does not necessarily guarantee optimal overall performance of the model checking algorithm. The computational overhead required to guarantee optimality may by far cancel out any benefits that an algorithm may have from exploring a smaller state space of interleavings. With a focus on overall performance, we propose new algorithms for stateful POR based on the recently proposed source sets, which are less precise but more efficient than the state of the art in practice. We evaluate efficiency using an implementation that extends Java Pathfinder in the context of verifying concurrent data structures.

1 Introduction

Concurrency results in insidious programming errors that are difficult to reproduce, locate, and fix. Therefore, verification techniques that can automatically detect and pinpoint errors in concurrent programs are invaluable. *Model checking* [7, 37] explores the state space of a given program in a systematic manner and verifies that each reachable state satisfies a given property. It provides high coverage of program behavior, but it faces the infamous state explosion problem, i.e., the number of possible thread interleavings grows exponentially in the size of the source code. In this paper, we consider shared-memory programs running on a sequentially consistent memory model, for which interleavings of atomic steps in different threads are a precise model of concrete executions.

Partial order reduction (POR) [8, 16, 34, 40] is an approach that limits the number of explored interleavings without sacrificing coverage. POR relies on an equivalence relation between interleavings, where two interleavings are equivalent

if one can be obtained from the other by swapping consecutive independent (non-conflicting) execution steps. It guarantees that at least one interleaving from each equivalence class (called a Mazurkiewicz trace [30]) is explored. Optimal POR techniques explore exactly one interleaving from each equivalence class. Beyond this classic notion of optimality, POR techniques may aim for optimality by avoiding visiting states from which no optimal execution may pass. There is a large body of work on POR techniques that address its soundness when checking a certain class of specifications for a certain class of programs, or its theoretical optimality (see Section 6). The set of interleavings explored by some POR technique is defined by restricting the set of threads that are explored from each state (scheduling point). Depending on the class specifications, assumptions about programs, or optimality targets, there are various definitions for this set of processes, including stubborn sets [40], persistent sets [16], ample sets [8], and source sets [3].

The design of a model checking algorithm based on POR has to consider several computational tradeoffs. First, such an algorithm can be stateful or stateless [17], which corresponds to a tradeoff between memory consumption versus execution time. Stateful model checking records visited states, thereby consuming more memory, but stateless model checking performs redundant exploration from already visited states. Second, the computation of the set of threads that are explored from some state can be more or less complex. Focusing on theoretical optimality, e.g., exploring *exactly* one interleaving from each Mazurkiewicz trace, may make this computation more complex. This complexity in turn may diminish the overall performance when the potential for reducing the state space is not large, i.e., most Mazurkiewicz traces contain a small number of interleavings. In such a case, exploring more interleavings can take less time than computing more precise constraints on the explored schedules. Third, POR algorithms may compute the information they use for the purpose of reduction *statically*, by some kind of conservative static analysis of the source code, or *dynamically*, during the exploration of interleavings. Static computation is usually cheaper and less precise than dynamic computation.

In this work, we investigate the use of POR from a practical point of view. In the context of verifying concurrent data structures, we investigate the following research question: what tradeoffs in POR families of algorithms may lead to practical net gains in verification or bug-finding times? We focus on the application domain of verification of Java concurrent data structures using a tool like Violat [9]. Concurrent data structures provide implementations of common abstract data types (ADTs) like queues, key-value stores, and sets. Their correctness amounts to observational refinement [22, 23, 35] which captures the substitutability of an ADT with an implementation [29]: any combination of values admitted by a given implementation is also admitted by the given ADT specification. Violat can be used to generate tests of observational refinement, i.e., bounded-size clients of the concurrent data structure that include assertions to check that any combination of return values observed in an execution belongs to a statically precomputed set of ADT-admitted return-value outcomes. Violat is integrated with the Java Pathfinder (JPF) model checker [41], which enables

complete systematic coverage of a given test program and outputting execution traces leading to consistency violations, thus facilitating diagnosis and repair. We investigate POR algorithms implementable in JPF.

We study several stateful model checking algorithms with POR in the context of Violat's test programs. This choice was inspired by experiments that demonstrated that it is much faster than the stateless variation (see Section 5). We introduce POR algorithms that combine *static and dynamic* computations of sets of threads to explore from a given state. In the context of stateful model checking, static techniques may seem like the better option. A dynamic computation usually requires re-traversing the state space starting in an already visited state which can be time consuming. Note however that re-traversing the state space that is already loaded in memory takes less time than generating that state space in the first place, which involves executing program statements.

Our starting point is a simple static POR algorithm, called S-POR, that makes use of *invisible* transitions. These transitions are independent of any transition of another concurrently-executing thread (they correspond to the safe actions introduced in [24]). Based on a syntactic analysis of the code, we identify shared and synchronization objects, and assume that every transition that does not access such an object is invisible. For clients of concurrent data structures, such objects correspond to class fields accessed in a method of the data structure. Invisible actions include starting and joining threads, and method calls and returns, for instance. The POR algorithm prioritizes the exploration of invisible transitions over visible ones, i.e., if an invisible transition is enabled in a given state then this is the only explored transition from that state, and otherwise, all enabled transitions are explored. We demonstrate that S-POR has a small overhead and the potential for substantial reductions, and therefore leads to significant speedups with respect to standard JPF which employs a very conservative heuristic for its POR (see Section 5).

S-POR is effective, but by the nature of being lightweight, does not always reduce the state space effectively. We introduce two new algorithms as extensions of S-POR, with the idea of performing a more aggressive reduction while keeping the overhead reasonably low. They *dynamically* compute *source sets*, which restrict the set of threads explored from a state with only *visible* enabled transitions. We focus on source sets since they are provably minimal, i.e., the set of explored threads from some state must be a source set in order to guarantee exploration of all Mazurkiewicz traces. Moreover, any superset of a source set is also a source set, which makes their computation less sensitive to the order in which transitions from a given state are enumerated. This property does not hold for other definitions such as stubborn sets, persistent sets, or ample sets.

The design principle behind our algorithms is to favor efficiency over theoretical optimality. Our algorithms are not theoretically optimal. However, we demonstrate that they are more efficient than the optimal algorithm [3] where the overhead of source set computation subsumes any gains from not exploring the redundant interleavings.

In general, a dynamic computation of source sets relies on tracking dependencies between actions in the already explored executions. Our two proposed

algorithms differ in the way in which the tracking is performed: one is *eager* and called DE-S-POR, and the other is *lazy* and called DL-S-POR. Intuitively, DE-S-POR advances the computation of source sets for predecessors in the current execution in a style similar previous dynamic POR algorithms, e.g. [13, 3], while DL-S-POR advances the computation of the source set in a given state only when the exploration backtracks to that state and one must decide if a new transition has to be explored.

The thesis of this paper is that when there is a big enough potential for reducing the state space of a concurrent program, i.e., many Mazurkiewicz traces are large enough, non-optimal but carefully customized algorithms, like DE-S-POR and DL-S-POR, can have the largest impact compared to the two extremes of the spectrum, that is, S-POR or theoretically optimal algorithms like [3]. If the potential for reduction is small, then a simple static algorithm like S-POR provides the best overhead-gain tradeoff.

To support this thesis, we implemented these algorithms in JPF and evaluated them on a number of clients of concurrent data structures from the Synchrobench repository [21]. Our evaluation shows that they outperform (1) their variations that are directly built on top of the standard setup of JPF, (2) their stateless variations, and (3) a best-effort implementation of a stateful variation of the optimal algorithm in [3] . The lazy algorithm DL-S-POR is more efficient than the eager DE-S-POR, and more efficient than S-POR on clients with a big enough potential for reducing the state space.

More details and experimental results can be found in a full version [6].

2 Preliminaries

We model a concurrent (multi-threaded) program with a bounded number of threads as a labeled transition system (LTS) $L = (\mathcal{S}, s_I, \Gamma)$. We assume that programs run under sequential consistency. A state in $\mathcal{S}$ represents a finite set of *shared* objects visible to all threads and a finite set of *local* objects visible to a single fixed thread, and a program counter for each thread. The state $s_I \in \mathcal{S}$ is the unique initial state. Γ is a set of labeled transitions (s, a, s') where $s, s' \in \mathcal{S}$ and a is an *action* (transition label) representing the execution of an *atomic* statement in the code. Action a records the executing thread, its program counter, and shared object accesses. There are two types of actions: (1) *invisible actions*: $a = (t, pc, \epsilon)$ where a thread t executes a statement at program counter pc that accesses no shared object, and (2) *visible actions*: $a = (t, pc, r/w, o)$ where t executes a statement at pc that reads (r) or writes (w) the shared object o. For an action a, $tid(a)$ is the thread id t, and $op(a)$ and $obj(a)$ refer to the third and fourth components when a is visible (otherwise they are undefined).

A transition labeled by a visible (resp. invisible) action is called visible (resp. invisible). In the context of a full-fledged programming language, invisible transitions are related to local computations, control-flow manipulations (e.g., starting/stopping threads and calling or returning from a method), or accesses to "low-level" shared objects that are irrelevant for the intended (functional) speci-

fication. Visible transitions correspond to the execution of a single atomic statement that accesses a shared object followed by a maximal sequence of *local* statements that only modify the local states of that thread.

We assume that LTSs are deterministic and acyclic. An action a is *enabled* in state s if there exists s' such that $(s, a, s') \in \Gamma$. We use $next(s,t)$ to denote the transition $(s, a, s') \in \Gamma$ for some a and s' with $tid(a) = t$, if it exists, and $succ(s,t)$ to denote the successor s' in this transition. Otherwise, we say that t is *blocked* in s. The set $enabled(s)$ is the set of threads that are not blocked in s. A state s is *final* if $enabled(s) = \emptyset$.

Two actions a and a' of different threads are *independent* if they are both enabled in a state s and either one of them is an invisible action, or they are both visible and access different shared objects ($obj(a) \neq obj(a')$), or they both perform a read access ($op(a) = op(a') = r$). The actions a and a' are called *dependent*, denoted by $a \nsim a'$, if they are not independent. We assume that if an action a enables or disables another action a', then $a \nsim a'$. Two transitions are (in)dependent iff they contain actions that are (in)dependent.

An *execution from a state* s is a sequence of alternating states and actions $E = s_0, a_0, s_1, a_1, \ldots, s_n$ with $s_0 = s$ and $(s_i, a_i, s_{i+1}) \in \Gamma$ for each $0 \leq i \leq n-1$. The set of execution starting from s in the LTS L is denoted by $E(L, s)$. An *initialized* execution is an execution from s_I. Initialized executions that end with a final state are called *full* executions. We assume absence of deadlocks, i.e., a full execution E contains every action enabled in a state of E.

The *happens-before* relation in an execution E, denoted by $\rightarrow_E$, captures the causal relation among actions in E (the program order between actions of the same thread and the order between actions accessing the same shared object where at least one of them is a write). Given two actions a and a' labeling transitions in E, $a \rightarrow_E a'$ holds iff $a \nsim a'$ and the transition labeled by a occurs before the transition labeled by a' in E. Two executions E and E' are called *equivalent* if $\rightarrow_E = \rightarrow_{E'}$. For a full execution E, we use $[E]$ to denote the set of full executions E' that are equivalent to E.

Given an LTS $L = (\mathcal{S}, s_I, \Gamma)$ that models a concurrent program, an LTS $L_r = (\mathcal{S}_r, s_I, \Gamma_r)$ with $\mathcal{S}_r \subseteq \mathcal{S}$ and $\Gamma_r \subseteq \Gamma$ is called *sound for* L if for each full execution E of L, there exists a full execution E' of L_r that is equivalent to E.

2.1 Partial Order Reduction

The set of executions explored by POR techniques is defined by restricting the set of threads that are explored from each state. The algorithms discussed in this paper fall into two categories in this respect: persistent sets and source sets. Both guarantee soundness, i.e., at least one execution from each equivalence class is explored.

Intuitively, a set of threads T is *persistent* for a state s if in any execution starting from s, the first transition that is dependent on some transition starting from s of some thread $t \in T$ is taken by some thread $t' \in T$ (t and t' may be equal). A set of threads T is a *source* set for s if for any execution starting

from s, there is some thread in T that can take the first step, modulo reorderings of independent transitions. We define persistent and source sets as sets of threads, which correspond to sets of transitions in the classical sense, under the assumption of determinacy of individual threads.

Definition 1 (Persistent Set [16]) *A set of threads T is called a* persistent set for a state *s if for every execution E from s that contains only transitions from thread $t' \notin T$, every transition in E is independent of every transition $next(s, t)$ with $t \in T$.*

For an execution E from a state s that ends in a final state, a thread t is called a *weak initial* of E if there exists an execution E' that is equivalent to E and starts with a transition of t.

Definition 2 (Source Set [3]) *A set of threads T is called a* source set for a state *s if every execution from s that ends with a final state has a weak initial thread in T.*

An exploration where each state is expanded w.r.t. the threads in a persistent or source set is sound (when finished it produces an LTS which is sound for the "full" LTS of the program). However, source sets guarantee a stronger notion of optimality [3]. There exist programs where any persistent set (for the initial state) is strictly larger than a source set [2], but every persistent set is also a source set. Note that source sets are monotonic in the sense that any superset of a source set is also a source set, but this is not true for other definitions such as stubborn sets, persistent sets, or ample sets.

3 Eager Source Set POR (DE-S-POR)

We present a first stateful POR algorithm that selects a sufficient set of threads to expand a state based on two criteria: (1) a static criterion based on (in)visible actions, and (2) a dynamic criterion based on *source sets* computed on-the-fly during the exploration. Source sets are maintained *eagerly* for each new transition that is explored, in a style similar to previous algorithms, e.g. [13, 3]. For presentation reasons, we start with a simplified version that includes only the static criterion and continue with the full version afterwards.

3.1 Safe Set POR (S-POR)

Algorithm 1 presents a stateful DFS traversal of a concurrent program, represented by an LTS, which restricts the traversal to so called *safe sets*. Figure 1 illustrates the core idea of this algorithm. The safe sets prioritize the exploration of *invisible* transitions over visible ones.

For a state s, if there is an enabled thread $t \in enabled(s)$ whose enabled transition is invisible, then $safeSet(s) = \{t\}$. Otherwise, $safeSet(s)$ contains all the threads enabled in s, and s is called an *irreducible state*. In Figure 1, only state s' is irreducible since any other state has at least one enabled invisible transition, and all other states are reducible.

Algorithm 1: Safe Set POR (S-POR)

Initialize: $Stack \leftarrow \emptyset$; $Stack.\textbf{push}(s_I)$; $L_r \leftarrow \emptyset$;
1 **Explore()**
2 $s \leftarrow Stack.\textbf{top}$;
3 **if** $notVisited(s)$ **then**
4 **forall** $t \in safeSet(s)$ **do**
5 $(s, a, s') \leftarrow next(s, t)$;
6 $Stack.\textbf{push}(s')$; `// transition` (s, a, s') `is added to` L_r
7 **Explore**();
8 $Stack.\textbf{pop}()$;

$notVisited(s)$ holds if s is final in L_r but $enabled(s) \neq \emptyset$

$$safeSet(s) = \begin{cases} \{t\}, & \exists t \in enabled(s) : next(s, t) = (s, a, s') \text{ and } a \text{ is invisible} \\ enabled(s), & \text{otherwise} \end{cases}$$

In Algorithm 1, *Stack* represents the stack of the DFS traversal and it is considered to be a global variable, and L_r records transitions explored during the traversal. Note that the DFS traversal stops the exploration whenever it visits a state s that has been visited in the past (see the condition at line 3). The choice of safe sets then provides additional savings on top of the standard DFS traversal strategy. When the traversal ends, L_r is sound (for the "full" LTS of the program).

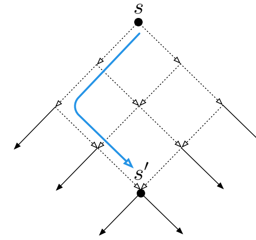

Fig. 1. Full traversal vs. partial S-POR (in blue).

Observe that Algorithm 1 can reduce the number of visited states in a significant way. The diagram in Figure 1 corresponds to a fully explored program LTS while the path marked by the blue arrow is the result of Algorithm 1. It is easy to observe that one can obtain an exponential reduction (with the base of the number of consecutive invisible transitions and the exponent of the number of threads) with this algorithm.

3.2 Full Algorithm

Algorithm 2 builds on top of Algorithm 1 by computing on-the-fly source sets to limit exploration of transitions from the *irreducible* states. More precisely, reducible states are traversed according to the strategy of Algorithm 1 (i.e., only one enabled invisible transition is followed) and for irreducible states, source sets determine what transitions are followed. Since safe sets are also source sets, the overall algorithm remains sound if the new source sets are computed correctly.

Figure 2 provides a declarative description of the key components of Algorithm 2. For a state s in the current execution (stored on the stack), the $s.$**current** set may be updated every time a new visible transition is explored, and the $s.$**backtrack** set may be updated every time the exploration backtracks

Algorithm 2: Eager Source Set POR (DE-S-POR)

```
   Initialize: Stack ← ∅; Stack.push(s_I); L_r ← ∅;
1  Explore()
2    s ← Stack.top;
3    if notVisited(s) then
4      if ∃t ∈ safeSet(s) then
5        s.backtrack ← {t}; s.current ← ∅; s.done ← ∅;
6        while ∃t' ∈ s.backtrack \ s.done do
7          (s, a, s') = next(s, t');
8          Stack.push(s');
9          s.done = s.done ∪ {t'};
10         s.current[t'] ← {t'};
11         if a is visible then UpdateCurr(a) ;
12         Explore();
13         s.backtrack ← UpdateBack(s, a);
14         Stack.pop();
15   else
16     A_s ← {a' : a' occurs in an execution from E(L_r, s)};
17     foreach a' ∈ A_s do UpdateCurr(a');
18 UpdateCurr(a)
19   E is the initialized execution of L_r following states in Stack;
20   (s, a', s') is the last transition of E with a≁a' ∧ tid(a)≠tid(a')
21   if (s, a', s') ≠ null then
22     s.current[tid(a')] = s.current[tid(a')] ∪ {tid(a)};
```

$$UpdateBack(s,a) = \begin{cases} safeSet(s), & \exists t \in s.\mathbf{current}[tid(a)] \setminus safeSet(s) \\ s.\mathbf{done}, & \exists T \subset s.\mathbf{done} : T = \bigcup_{t \in T} s.\mathbf{current}[t] \\ \bigcup_{t \in s.\mathbf{done}} s.\mathbf{current}[t], & \mathbf{otherwise} \end{cases}$$

s.**current**$[t]$: set of threads that execute a transition dependent on $next(s,t)$ which appears after it in an execution.
s.**done**: set of threads whose transitions have been fully explored from s.
s.**backtrack**: when equal to s.**done**, a source set for s.

Fig. 2. Description of important components in Algorithm 2.

to s. The update of s.**backtrack** relies on the sets s.**current** computed while traversing successors of s.

When a new transition (s, a, s') from a state s is traversed, the active thread $tid(a)$ is added to the current set s_l.**current**$[t]$, where s_l is the last state from which the current execution performs a transition that is dependent on a such that $t \neq tid(a)$ is the thread of that transition. See line 11 and the **UpdateCurr** function. When a transition is followed to a visited state s, the same update is done for *every* transition that is reachable from s, as if these transitions are traversed again. See lines 16-17 and note that the declarative definition of

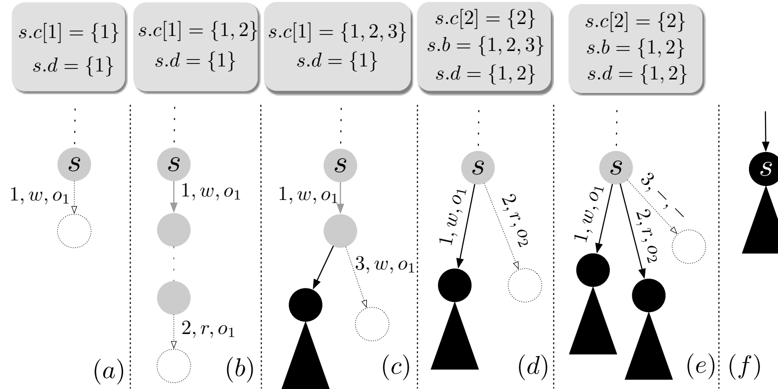

Fig. 3. An example for Algorithm 2. Solid grey circles represent states stored on the stack, hollow dotted circles represent states on the top of the stack, and solid black circles represent states from which the exploration has been completed, i.e. their **backtrack** sets are equal to their **done** sets. Transitions follow the same pattern: dotted transitions are the latest to have been explored, solid grey ones are between states on the stack, and solid black ones are the ones taken in the past. Solid black triangles represent completed explorations starting from some state. We omit program counters from actions. **backtrack**, **current**, and **done** are abbreviated by the first letter.

A_s at line 16 corresponds to a traversal of all the executions starting from s. This may be time-consuming, and yet, such updates are unavoidable in stateful POR algorithms because the current execution reaching s (stored on the stack) may belong to a different Mazurkiewicz trace compared to a previous execution reaching s (whose sequence of transitions leading to s was different).

When backtracking to a state s, the set s.**backtrack** is updated to take into account the transitions which are dependent on and occur after the last explored transition starting from s, called τ_s. If τ_s is a transition of thread t, the threads performing those dependent transitions are stored in s.**current**$[t]$. If there is a dependent transition τ performed by a thread t' that is not in the safe set of s, then s.**backtrack** is updated conservatively to contain the safe set of s. This situation occurs when τ becomes enabled after executing some other thread t'' enabled in s, and observing an execution where τ_s occurs after τ requires first executing the transition of t''. Otherwise, the algorithm checks to see if a subset T of threads enabled in s which have already been explored are sufficient to cover s.**current**$[t]$, and that T's transitions in s are independent from future transitions of threads not in T. In that case, s.**backtrack** is assigned with s.**done** and the exploration from s is halted. The subset of threads T defines a persistent set **and** a source set for s. Since source sets are monotonic, s.**done** is a source set for s. If none of the previous conditions hold, then s.**current**$[t]$ is simply added to s.**backtrack**. This computation is defined by the macro *UpdateBack* in Figure 2.

We illustrate the algorithm using Figure 3. In (a), s is reached for the first time and the transition labeled by $(1, w, o_1)$ is selected first to be executed. This

is a visible transition of thread 1 that writes to the shared object o_1. After this transition is taken, s.**current**[1] and s.**done** become $\{1\}$. In (b), from some state which is reached later, the transition labeled by action $(2, r, o_1)$ is selected to be followed next. Since this action is dependent on $(1, w, o_1)$, thread 2 is added to s.**current**[1]. Then in (c), a transition of thread 3 with an action dependent on $(1, w, o_1)$ is taken, and 3 is added to s.**current**[1]. After backtracking to s in (d), s.**backtrack** is updated by simply copying s.**current**[1]. The next transition to be taken from s belongs to thread 2 which is in s.**backtrack**$\setminus s$.**done**. This entails the initialization of s.**current**$[2] = \{2\}$ and the addition of 2 to s.**done**. In (e), we backtrack to s again and without having changed s.**current**[2]. This means that $(2, r, o_2)$ is independent of any later action of another thread. Therefore, $\{2\}$ is a persistent set of s and s.**done** $= \{1, 2\}$ a source set of s, and s.**backtrack** is assigned with s.**done** to stop the exploration from s, as pictured in (f).

This example shows that Algorithm 2 explores sets of transitions from a given state s that may correspond to a source set which is *not* a persistent set. The exploration in Figure 3 stops when s.**backtrack** $=$ s.**done** $= \{1, 2\}$, but the only persistent set that includes thread 1 is $\{1, 2, 3\}$.

Theorem 1. *Given a program represented by an LTS L, Algorithm 2 terminates with an LTS L_r that is sound for L.*

Proof. Based on the soundness of source sets (see Section 2.1), it is enough to show that for every state s in L_r,

s.**backtrack** is a source set for s in L when it becomes equal to s.**done** (1)

Due to the condition of **while** in line 6 of Algorithm 2, equality of s.**backtrack** and s.**done** is the only condition for stopping an exploration from a state s. Therefore, if some successor state s' of s is already explored and the search is backtracked to s, then s'.**backtrack** $= s'$.**done**, because otherwise, **while** loop in line 6 wouldn't be terminated for s'. Since s.**done** keep tracks of threads whose enabled transitions from s is already executed, the proof is reduced to showing that the following proposition holds:

For any state s, s.**backtrack** is a source set when the exploration from s is finished (2)

If s is a reducible state, then only one transition is explored from s which is an invisible transition. The fact that the thread performing this invisible transition is a persistent set and hence a source set follows directly from definitions as every persistent set is also a source set. When s.**backtrack** $=$ s.**done** is different from $\mathit{safeSet}(s) = \mathit{enabled}(s)$ (which is trivially a source set), it must be the case that there exists $T \subset s$.**done** such that $T = \bigcup_{t \in T} s.\mathbf{current}[t]$ due to the definition of *UpdateBack* method. Now we show that T is a persistent set for s in L. Assume by contradiction that this is not the case, then due to the definition of persistent set, L admits an execution E starting from s that contains only transitions of threads different from those in T and at least one of these transitions τ of a

thread $t' \notin T$ is dependent on some transition $next(s, t)$ with $t \in T$. For every $t \in T$, the successor state s' of s reached by $next(s, t)$ must be in L_r. Due to deadlock freedom assumption, some transition that has the same transition label with τ must be enabled eventually in some successor state of s'. Let $E' \in L_r$ be that execution from s which starts with $next(s, t)$ and contains such transition that shares the same label with τ. Now we will move forward by showing that the following proposition is correct, which will be used in the rest of the proof:

If L_r admits an execution E'' from s whose last transition that depends on and occurs before τ' is $next(s, t)$ where $act(\tau') = act(\tau)$ and $tid(\tau') = tid(\tau) = t'$, then $t' \in T$ (3)

When τ' is executed from some successor state of s, t' is added to s.**current**$[t]$ (and eventually will be added to T due to *UpdateBack* method) by invoking the **UpdateCurr** function as $next(s, t)$ will be the transition in line 20 of Algorithm 2. This contradicts the assumption of $t' \notin T$ and therefore, it is enough to show that proposition below is correct for concluding the proof:

If L_r admits such an execution E', then L_r admits such an execution E'' (4)

To show that Proposition 4 holds, we proceed by induction on the order of $next(s, t)$ when we go backwards in E'. The base step is trivial since E'' can be E' when $next(s, t)$ is the first transition. Assuming by induction that $next(s, t)$ is the n-th transition that is dependent on and occurs before τ' in E' and L_r admits such execution E'', we show that this also holds when $next(s, t)$ is $(n+1)$-th transition with the same properties. Let s'' be the state that is reached from s by executing E'_p which is the prefix of E' until (not included) the last transition τ'' that is dependent on and occurs before τ' and hence, τ'' is enabled in s''. Using Proposition 3, as $tid(\tau')$ must be in T of s'', there must be another execution E''' from s'' such that τ''' occurs before τ'' where $act(\tau') = act(\tau''')$ and $tid(\tau') = tid(\tau''') = t'$. Since in the execution starting from s as E'_p and continues as E''', $next(s, t)$ is the n-th transition that is dependent on and occurs before τ''' (when we go backwards), proposition 4 is correct by using induction assumption. As mentioned, this contradicts the assumption of $t' \notin T$ in proof of proposition 2 and thus, T is a persistent set and a source set. By monotonicity of source sets, s.**backtrack** is also a source set. □

4 Lazy Source Set POR (DL-S-POR)

Algorithm 2 tracks dependencies between transitions in an eager manner, i.e., every new transition leads to updates of **current** sets. In this section, we present a lazy variation that computes such dependencies only when the exploration backtracks to a state. The incentive is to compute such dependencies only when needed to decide if the exploration from a given state should continue or not. Also, this enables several optimizations when traversing the state space to compute such dependencies that are not possible in the eager version.

Algorithm 3: Lazy Source Set POR (DL-S-POR)

```
   Initialize: Stack ← ∅; Stack.push(s_I); L_r ← ∅;
1  Explore()
2    s ← Stack.top;
3    s.backtrack ← ∅; s.done ← ∅; s.current ← ∅;
4    while true do
5      if ∃t_1 ∈ s.backtrack \ s.done then
6        t ← t_1
7      else
8        choose t ∈ safeSet(s) \ s.done
9      (s, a, s') = next(s, t);
10     Stack.push(s');
11     if notVisited(s') then
12       Explore();
13     if IsComplete(s) then
14       Stack.pop();
15       return
16     Stack.pop();
17 IsComplete(s)
18   forall (s, a, s') ∈ {s' ∈ L_r : t = tid(a) ∉ s.done} do
19     s.done = s.done ∪ {t};
20     T ← safeSet(s);
21     if s.done = T ∨ (∀t' ∈ T : isVisited(succ(s, t')) ∨ t' ∈ s.done) then
22       add transitions (s, a, s') to L_r;
23       s.done ← T;
24       s.backtrack ← s.done;
25       return true;
26     s.current[t] ← {t};
27     A_{s'} ← {a' : a' occurs in an execution from E(L_r, s')};
28     s.current[t] = s.current[t] ∪ {tid(a') : a' ∈ A_{s'} and a ≁ a'};
29     s.backtrack ← UpdateBack(s, a);
30     if s.backtrack = s.done then
31       return true;
32   return false;
```

Algorithm 3 presents our POR algorithm based on a lazy computation of source sets. Rather than updating the **current** sets on-the-fly for states on the stack, this algorithm re-traverses part of the state space each time it backtracks to a state s in order to update just **current** sets of s. This is done in the function **IsComplete**. As a result, s.**done** is populated with a new thread t just before computing dependencies with t's transition in s and not after executing that transition in the style of Algorithm 2 (see line 19). For every transition τ of a thread t from s that has been followed since the last time the algorithm backtracks to s (i.e., t is not already in s.**done** – see line 18), the algorithm updates s.**current**$[t]$ to include all threads t' that later execute a transition that is dependent on τ (see lines 26-28). Subsequently, the s.**backtrack** set is updated

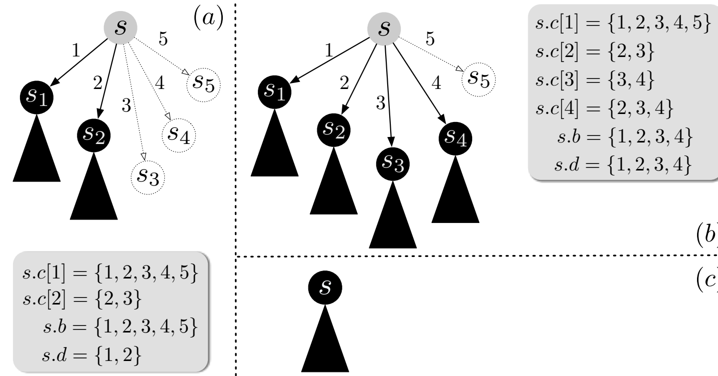

Fig. 4. An example exploration of Lazy Source Set POR. We use the same conventions as in Figure 3.

exactly in the style of Algorithm 2 (see line 29). If s.**backtrack** becomes equal to s.**done** then **IsComplete** returns *true* and the exploration from s stops.

We explain how Algorithm 3 works by an example. Figure 4(a) illustrates a scenario in which the exploration backtracks to a state s for the second time. After the first backtrack to s, the state space starting from the successor s_1 (resulted from following a transition of thread 1) was re-traversed in order to compute s.**current**[1]. We assume that s.**current**[1] is changed to $\{1, 2, 3, 4, 5\}$ due to the dependent transitions encountered during this traversal. The set s.**backtrack** is set to s.**current**[1] as the latter contains all the enabled transitions. The exploration continues with a transition from s of thread 2 which is possible because thread 2 is in s.**backtrack** $\setminus$ s.**done**. After backtracking to s for the second time, the re-traversal of the state space starting in s_2 leads to s.**current**[2] $= \{2, 3\}$. The set s.**backtrack** remains the same after this computation. Then, in Figure 4(b), when backtracking to s for the fourth time, we assume that s.**current**[3] $= \{3, 4\}$ and s.**current**[4] $= \{2, 3, 4\}$. Since transitions of threads 2, 3, and 4 starting in s are independent of transitions of other threads that occur later, we can conclude that $\{2, 3, 4\}$ is a persistent set and $\{1, 2, 3, 4\}$ is a source set, and update s.**backtrack** to s.**done**. Therefore, the exploration from s stops, as pictured in Figure 4(c). The set of transitions explored from s corresponds to a source set which is not a persistent set. The only persistent set that includes thread 1 is $\{1, 2, 3, 4, 5\}$.

In both DE-S-POR and DL-S-POR, we optimize re-traversals after some state s (computing A_s at line 16 and line 27, respectively) by not traversing all the executions after s but just traversing each transition after s only once. DL-S-POR is also amenable to other optimizations that are not possible or difficult to implement for DE-S-POR. These optimizations for DL-S-POR either prevent some re-traversals inside the **IsComplete** method (see the if block at line 21) or provide early exit conditions for them. All these optimizations are explained in detail in the full version [6].

The soundness of Algorithm 3, stated in the following theorem, is also based on proving that every state is expanded according to a source set. As in Theo-

rem 1, it can be shown that s.**backtrack** is a source set for s when it becomes equal to s.**done**. When backtracking to a state, the **current** sets satisfy the same specification as in the eager version.

Theorem 2. *Given a program represented by an LTS L, Algorithm 3 terminates with an LTS L_r that is sound for L.*

Proof. Similar to the proof of Theorem 1, it is enough to show that Proposition 1 holds. To end an exploration from a state s, **while** loop in line 4 of Algorithm 3 must be terminated. For this, **return** statement in line 15 must be reached and therefore, **IsComplete** in line 13 should return true. First, we show that **IsComplete** method eventually returns true. Due to to method *UpdateBack*, we know that s.**backtrack** can not contain a thread $t \notin \mathit{safeSet}(s)$. Hence, all the transitions of s that can be executed are only from threads in $\mathit{safeSet}(s)$ because of lines 5–8. Each time a transition of s is executed and then the search backtracks to s, **IsComplete**(s) is initiated. By the **for** loop in line 18, every transition from s that is executed is considered and by line 19, we know that all these transitions will be added to s.**done**. Thus, s.**done** eventually becomes equal to $\mathit{safeSet}(s)$ which satisfies the condition in line 21 and as a result, **IsComplete** method returns true.

For **IsComplete** method to return true, either condition in line 21 or line 30 should be satisfied, where in both conditions s.**backtrack** must be equal to s.**done** before the return statement. That's why, equality of s.**done** and s.**backtrack** is the only condition for stopping an exploration from a state s. Similar to Algorithm 2, since s.**done** keep tracks of threads whose enabled transitions from s is already executed, the proof is deduced to showing that Proposition 2 holds for Algorithm 3 as well. The part between Proposition 2 and Proposition 3 in the proof of Theorem 1 applies totally the same and we show that T is a persistent set for s in L using the fact that L_r admits such an execution E' as it is concluded in the same proof. Assume by contradiction that T is not a persistent set for s. But as a result of backtracking to s after executing E' and invoking **IsComplete** method, t' will be added to s.**current**[t] (and eventually to T due to *UpdateBack* method) in line 28 of Algorithm 3 since the transition label of τ' is an element of $A_{s'}$ ($t' \notin T$ and $\mathit{act}(\tau) \nsim \mathit{act}(\tau')$) in line 27. Since it contradicts the assumption, T (and also s.**backtrack** by monotonicity of source sets) is a persistent set and a source set. □

5 Experimental Evaluation

We evaluate an implementation of the three algorithms S-POR, DE-S-POR, and DL-S-POR, presented in Section 3 and Section 4, in the context of the Java Pathfinder (JPF) model checker. As benchmark, we use bounded-size clients of Java concurrent data structures.

Implementation We implement our algorithms as an extension of the `DFSHeuristic` class in JPF. To identify (in)visible actions (for computing safe sets), the only manual input is a list of class names that constitute the implementation of

the concurrent data structure. The (in)visible transitions are automatically inferred from these class names and Java synchronization-related native methods used to implement compare-and-swap (CAS) for instance, which are all known. Every action reading or writing a field of an object in one of these classes, or which corresponds to a native method call are marked as visible (JPF makes it possible to parse the Bytecode instructions executed in a transition and determine the read/written object fields). Calls to the `lock` and `unlock` methods of a lock object are both considered as writes to the lock object, and therefore, visible. Any other action is considered as invisible. The dependency relation between visible actions is defined as usual, i.e., two actions that access the same object field, one of them being a write, are considered dependent. The way we define (in)visible actions is sound because the clients we consider do not contain additional computation. They simply call methods of the data structure (from different threads), the verification goal being related to combinations of return values observed in their executions.

Benchmarks Our benchmark consists of bounded-size clients of 7 concurrent data structures from JDK8 or Synchrobench [21]: two set implementations based on *coarse-grain* and *fine-grain* locking, respectively (RWLockCoarseGrainedListIntSet and OptimisticListSortedSetWaitFreeContains), a set implementation based on a binary search tree and CAS, a wrapper on top of java.util.concurrent.ConcurrentLinkedQueue, java.util.concurrent.ConcurrentHashMap and a wrapper on top of it, and a hash map implementation based on coarse-grain locking. Since these implementations update shared memory using compare-and-swap or guarded by locks, they are data-race free and the restriction to sequential consistency is sound.

To evaluate our algorithms, we sampled 75 clients of these data structures where each client calls add and remove methods from 3 threads. Each thread contains up to 5 calls. We varied the contention on shared objects using less or more distinct inputs for add and remove methods.

We also use a number of buggy variations of the lock-based sets, RWLockCoarseGrainedListIntSet and OptimisticListSortedSetWaitFreeContains. We used Violat to generate client programs of these variations that admit consistency violations. Violat generates these client programs in three steps. First, Violat enumerates arbitrary test programs of a given data structure based on other inputs such as number of threads, maximum number of programs and so on. Next, it computes expected (ADT-admitted return-value) outcomes for each test program by computing and then recording the outcomes of all possible sequential executions. Finally, it runs the threads of each test program in parallel (using a stress testing tool or JPF), checks if the results are as expected, and reports the test programs that violates linearizability which is witnessed by observing an unexpected outcome.

To introduce bugs in the selected data structures before inputting them to Violat, we modify the placement of locks dynamically under certain conditions in certain methods (e.g., when the set contains a specific element). These conditions make it possible to control the difficulty of a bug. We consider four different

classes of clients based on the number of invocations to methods that lead to bugs: (1) all of the invocations, (2) half of the invocations, (3) just a single invocation and (4) none of the invocations. We sampled 310 clients of these buggy variations with 3 threads and up to 4 calls per thread using Violat.

Results We use S-POR, DL-S-POR, DE-S-POR to denote the three algorithms presented in this paper. For the same algorithms, we use JPF, DL-JPF, DE-JPF to represent the standard setup of JPF, and variations of the DL-S-POR and DE-S-POR when the safe set of a state s contains all the enabled threads in s ($\mathit{safeSet}(s) = \mathit{enabled}(s)$). The latter are used to evaluate the performance of the eager and lazy approaches while disabling the benefit of the static S-POR method. We compare implementations of S-POR, DL-S-POR and DE-S-POR between them, with JPF, DL-JPF and DE-JPF, with their stateless variations, and with a stateful variation of the optimal source set algorithm in [3] (called O-DPOR). For a fair comparison, we implement O-DPOR on top of S-POR without wakeup trees as their operations are quite expensive. The experiments were run on a 2,3 GHz Dual-Core Intel Core i5 processor with 8GB of RAM. We consider a timeout of 30 minutes.

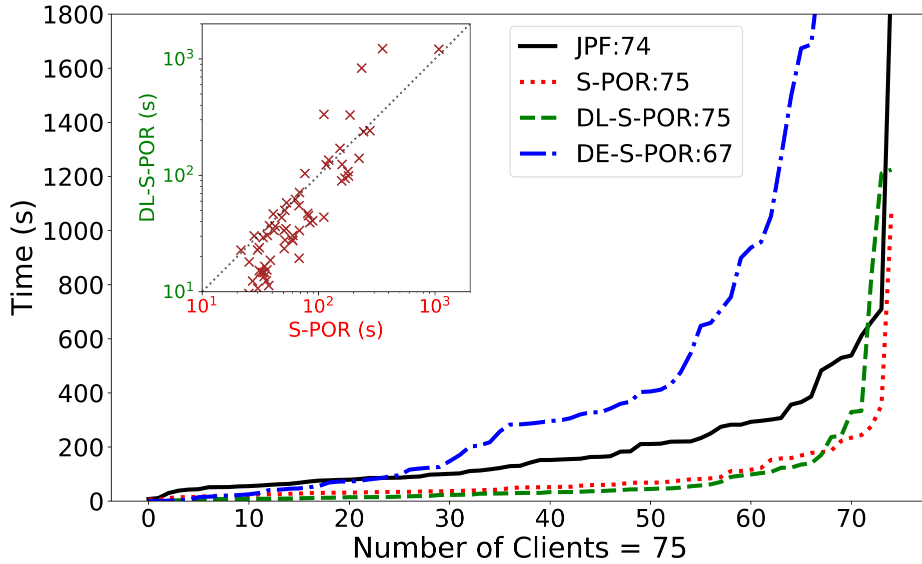

Fig. 5. Quantile plot of running times for S-POR, DL-S-POR, DE-S-POR and JPF (for each algorithm, clients are ordered w.r.t. time in ascending order). The top left part shows a scatter plot for comparing S-POR and DL-S-POR.

Execution time comparison. Figure 5 and Figure 6 present a comparison in terms of execution time between different sets of algorithms. In Figure 5, we compare JPF, S-POR, DL-S-POR and DE-S-POR to observe the advantages of using our algorithms against the standard setup of JPF. In Figure 6, we compare DL-S-POR, DE-S-POR, DL-JPF and DE-JPF for investigating the gain by applying static filtering using S-POR as a baseline in dynamic algorithms. To ease

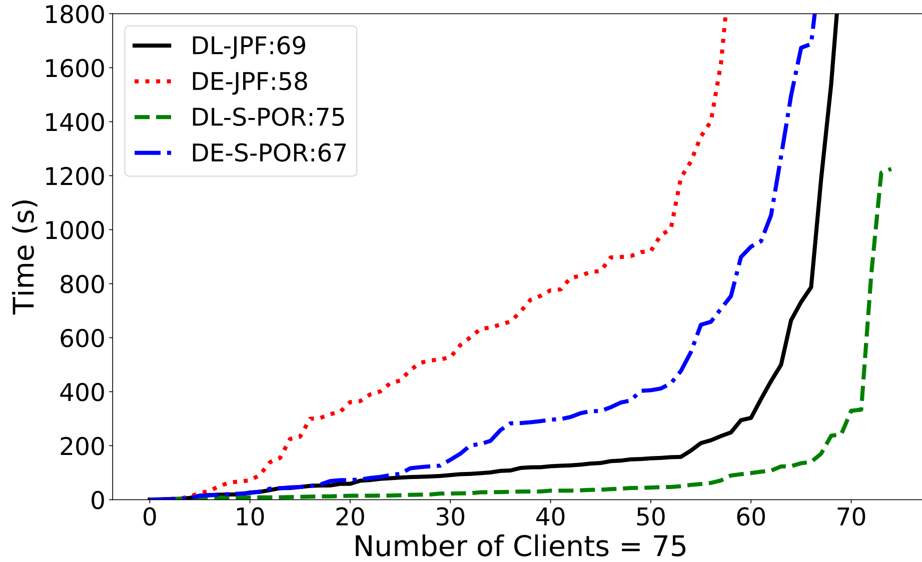

Fig. 6. Quantile plot of running times for DL-JPF, DE-JPF, DL-S-POR and DE-S-POR (for each algorithm, clients are ordered w.r.t. time in ascending order as in Figure 5).

the interpretation of the results, for each algorithm, we order clients according to execution time in ascending order. The numbers in the legend represent the number of clients on which a given algorithm terminates before the timeout. We omit O-DPOR because it times out for a large part of the benchmark, i.e., 39 out of the 46 clients on which it was run (our implementation of the algorithm in [3] does not support programs using locks which makes it inapplicable to the rest of the clients). This optimal algorithm manipulates happens-before constraints between steps in an execution, which results in a large overhead compared to our simpler tracking of pairwise dependencies. We also omit stateless variations of our algorithms since none of them finished before the timeout for any client. Note that stateless versions are obtained by disabling the state matching[5] in JPF, which also disables storing the full reachability graph.

Results based on Figure 5 show that the lazy source set computation in DL-S-POR gives a significant speedup w.r.t. DE-S-POR (and intuitively O-DPOR) while outperforming JPF. While S-POR processes few more clients faster w.r.t. DL-S-POR, the scatter plot on the top-left of Figure 5 shows that it is mostly in favor to DL-S-POR when clients are observed individually (this plot is given in logarithmic scale). DL-S-POR performs better than S-POR if there is a high potential for reduction, i.e., the ratio between the number of states explored by DL-S-POR over S-POR is smaller, and otherwise, S-POR is the best. This supports the hypothesis that if the potential for reduction is high enough then a

[5] JPF uses hashing for state matching which is theoretically imperfect and can lead to incomplete results on rare occasions

carefully customized dynamic computation of source sets has a significant impact on performance. DL-S-POR gives an average speedup (average of speedups for each client) of 2.6 compared to S-POR. Overall picture suggests using a portfolio model checker where S-POR and DL-S-POR are run in parallel.

Similar to Figure 5, Figure 6 illustrates a comparison in terms of time between DL-S-POR, DE-S-POR, DL-JPF and DE-JPF. It shows that our algorithms outperforms their variations that are directly built on top of JPF (DL-S-POR against DL-JPF and DE-S-POR against DE-JPF). It also highlights the fact that the lazy approach is still better than the eager one even when the lazy approach is not based on S-POR.

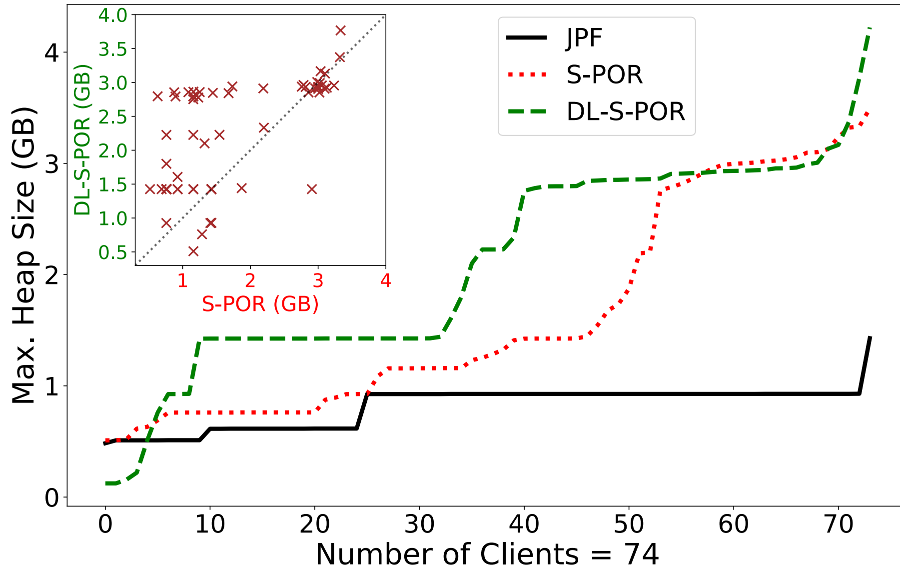

Fig. 7. Quantile plot of memory consumption for S-POR, DL-S-POR, and JPF (for each algorithm, clients are ordered w.r.t. memory in ascending order). The top left part shows a scatter plot for comparing S-POR and DL-S-POR.

Memory consumption comparison. Figure 7 presents a comparison in terms of memory consumption between S-POR and DL-S-POR, the most efficient algorithms according to Figure 5 and Figure 6, against the standard setup of JPF. We compared the maximum heap sizes using 74 clients that terminate before timeout for all algorithms. In all the experiments, the highest allocated heap size is 4.2GB. S-POR and DL-S-POR consume more memory than JPF because they have to store the transition labels which are used to reduce the explored state space. This overhead is unavoidable for any form of dynamic partial order reduction. However, this memory consumption overhead is counterbalanced by significant speedups in terms of time. There is some memory overhead also due to storing the sets of transition labels manipulated by the algorithms, e.g., s.**current**. But since these sets are maintained only for irreducible states and

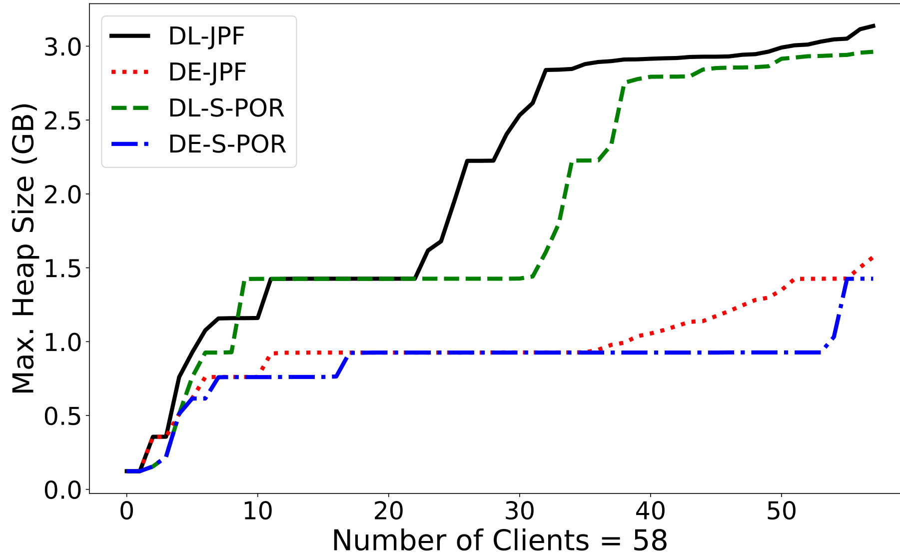

Fig. 8. Quantile plot of memory consumption for DL-JPF, DE-JPF, DL-S-POR and DE-S-POR (for each algorithm, clients are ordered w.r.t. memory in ascending order as in Figure 7).

they are deleted for a state s when s.**done** equals s.**backtrack**, their effects are not significant as storing transition labels.

For 32% of the clients, S-POR and DL-S-POR consume at most twice the memory consumed by JPF. For these clients, the average memory overhead is 1.00 for S-POR and 1.34 for DL-S-POR while the average speedup against JPF is 2.54 and 6.67, respectively. For 50% of the clients, S-POR and DL-S-POR consume in between 2 and 4 times the memory used by JPF. The average memory overhead for these clients is 2.20 for S-POR and 2.63 for DL-S-POR while the average speedup is 2.86 and 7.81, respectively. For the rest of the clients, the memory overhead is at most 7.79 and in average 4.11 for S-POR and 5.39 for DL-S-POR while the average speedup 3.28 and 5.31, respectively.

The top-left part of Figure 7 shows a pair-wise comparison of allocated maximum heap sizes in S-POR and DL-S-POR. These algorithms are incomparable in general. After investigating the clients individually, the results confirm that DL-S-POR consumes less memory than S-POR when there is a high potential for reduction. The memory consumed for computing source sets is compensated by the reduction in the state space.

When we take a look at to Figure 8, it illustrates a comparison between DL-S-POR, DE-S-POR, DL-JPF and DE-JPF as in Figure 5, but in terms of memory consumption. As in Figure 7, we compared the maximum heap sizes of clients that finished before it timed out for all algorithms, which are 58 of them. It demonstrates that our algorithms are slightly better than their variations that are directly built on top of JPF. This memory overhead is mainly because

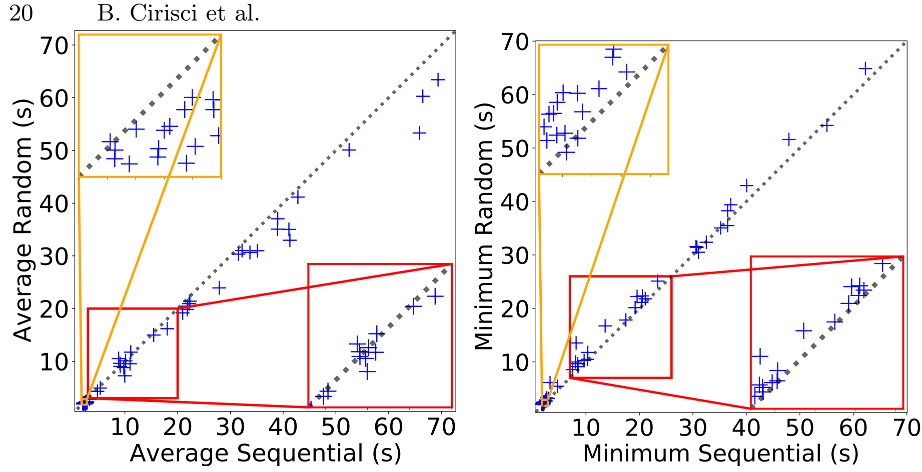

Fig. 9. Time comparison between the sequential and random strategies when DL-S-POR enumerates all states, using clients with a single buggy invocation.

of the additional transition labels that are not removed by the static filter. The overhead is also due to storing the sets of transition labels manipulated by the algorithms for all of the states rather than just for the irreducible ones but as mentioned previously, this overhead is negligible. This figure also shows that tracking dependencies with an eager approach does not increase the heap size as much as the lazy approach, although they explore the same state space. This difference in the memory overhead can not be explained by the memory that is used for storing the sets of transition labels or LTSs as they are all the same for both algorithms and they are kept in the same data structures. We suspect that this overhead can be due to low-level, internal details of JPF or due to the garbage collection process which might not keep up.

Transition enumeration. The performance of dynamic POR algorithms is generally affected by the order in which transitions starting in a certain state are enumerated. This order influences the size of the computed persistent/source sets. This order is also important when enumerating states only until the first error is detected. We evaluate two strategies for defining this order, called *sequential* and *random*. For both of these strategies, the algorithm first selects a transition that leads to an already visited state, if one exists. We made this choice because it leads to better performance (this is adopted by the standard setup of JPF as well). In the sequential strategy, if there is no such transition, then the algorithm picks a transition by respecting a pre-defined order between the thread ids. In random, the next transition is selected uniformly at random.

We ran S-POR and DL-S-POR, the best algorithms as shown above, with all 6 permutations of the 3 threads for sequential and 3 different seeds for random. For each strategy, we report the average and minimum time over different instances.

We report on the impact of these enumeration strategies for DL-S-POR when computing *all* reachable states (up to POR) in Figure 9 and *only until the first error* in Figure 10. For S-POR, the enumeration strategy is not important

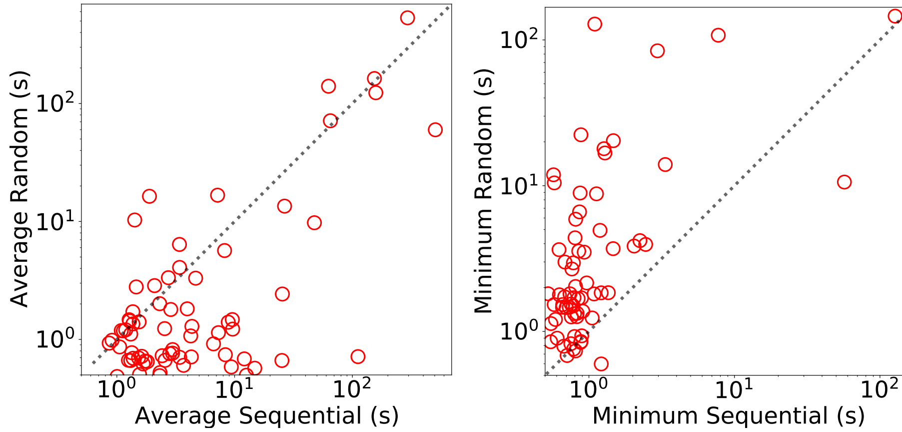

Fig. 10. Time comparison (log scale) between the sequential and random strategies when DL-S-POR enumerates states only until the first error, using clients in which all the invocations are buggy.

for the first case (since there is no dynamic computation of persistent/source sets) and it has a similar effect as for DL-S-POR in the second case. These figures consider clients of buggy libraries where a single or all method invocations are buggy. The rest of the cases are presented in the full version [6] and are similar.

The results show that the random strategy performs better in average, shown on the left of Fig. 9–10, but worse w.r.t. minima, shown on the right of Fig. 9–10.

The differences are more significant when enumerating states only until the first error. Fig. 10 reporting on this case is given in logarithmic scale. Thus, the sequential strategy should be preferred when using a portfolio model checker, i.e., parallel runs for each permutation of thread ids, and otherwise, the random strategy is better. This follows also from the average standard deviation being 28 seconds for random and 60 seconds for sequential, where the means are 17 and 20 seconds, resp. Note that, there is no significant impact observed from changing the algorithms or the number of buggy method invocations in the clients.

6 Related Work

Over the years various different techniques have been introduced to deal with the state explosion problem in model checking. For concurrent programs specifically, depth bounding [17], delay bounding [10], context bounding (bounding the number of context switches) [36], preemption bounding [32] and phase bounding [5] bring tractability to the model checking problem and have been shown to be effective for bug finding. These techniques are all *incomplete*, in the sense that lack of bugs does not guarantee the correctness of the system.

POR techniques reduce the search space by not exploring multiple executions from the same equivalence class, and are *complete*. Early techniques like *ample sets* [7, 24] and *stubborn sets* [20, 15] were based on static analysis. *Sleep sets* [15] were the first to guarantee optimality (one execution from each equivalence

class) [18] by keeping track of information from the history of the exploration. However, they only prune transitions and cannot eliminate any state when used alone. Persistent sets [19, 25] generalized stubborn and ample sets and enabled development of dynamic POR (DPOR) methods.

In [13], an efficient stateful algorithm is proposed for computing persistent sets dynamically by considering currently explored parts of the state space. This algorithm needed large memory for keeping discovered states and the happens-before relation. The algorithm is improved in [42] with a more efficient state representation, and in [43] with a summary-based representation of the happens before.

In [28, 39], stateless dynamic POR techniques were introduced. *Source sets* [3] were introduced in the context of dynamic POR techniques such that the state space can be reduced up to the limit that is theoretically possible. They are generalizations of persistent sets and their relation with persistent sets are investigated in [2]. Our DE-S-POR and DL-S-POR algorithms are relying on source sets but operate in the context of stateful model checking. The technique from [33] is similar to our S-POR algorithm for the GPU setting, but their choice of invisible actions is different than ours.

While we focus on shared-memory programs running on top of a sequential consistency memory model, POR techniques have been also investigated in the context of weak memory models such as TSO or C11, e.g., [27, 26, 1, 4].

7 Conclusions

We proposed two algorithms for stateful model checking based on POR which build on the recently proposed source sets. Our algorithms focus on overall performance instead of theoretical optimality. Their evaluation in the context of JPF shows that they outperform a theoretically optimal algorithm [3], and a simple static POR algorithm when there is a big enough potential for reducing the state space. This suggests that an effective model checker would have to run S-POR and DL-S-POR in parallel, and depending on the amount of parallelism resources available, with different instances of the sequential or random strategies for enumerating transitions starting in a certain state.

Reductions based on Mazurkiewicz trace equivalence [30] has also been used in proof simplification for concurrent verification [12] hypersafety verification [11]. A relevant problem of interest is the automatic inference of inductive invariants [31, 14, 38] that prove the correctness of concurrent libraries. The sound LTS's that are computed in this paper for the verification of individual instances, each provide a data point in what the inductive invariant for a most general client may look like. The key observation is that the inductive invariant for a most general client under some reduction scheme may be substantially simpler than an inductive invariant for all executions of the most general client. An interesting direction of future work is to investigate if the results of a sequence of individual client tests can be generalized to the discovery of a complete invariant (and hence a complete proof) under an appropriate reduction scheme. DL-S-POR provides an efficient way to produce data for a data-driven inference algorithm.

References

1. Abdulla, P.A., Aronis, S., Atig, M.F., Jonsson, B., Leonardsson, C., Sagonas, K.: Stateless model checking for TSO and PSO. Acta Informatica **54**(8), 789–818 (2017). https://doi.org/10.1007/s00236-016-0275-0, https://doi.org/10.1007/s00236-016-0275-0
2. Abdulla, P.A., Aronis, S., Jonsson, B., Sagonas, K.: Comparing source sets and persistent sets for partial order reduction. In: Aceto, L., Bacci, G., Bacci, G., Ingólfsdóttir, A., Legay, A., Mardare, R. (eds.) Models, Algorithms, Logics and Tools - Essays Dedicated to Kim Guldstrand Larsen on the Occasion of His 60th Birthday. Lecture Notes in Computer Science, vol. 10460, pp. 516–536. Springer (2017). https://doi.org/10.1007/978-3-319-63121-9_26, https://doi.org/10.1007/978-3-319-63121-9_26
3. Abdulla, P.A., Aronis, S., Jonsson, B., Sagonas, K.: Source sets: A foundation for optimal dynamic partial order reduction. J. ACM **64**(4), 25:1–25:49 (2017). https://doi.org/10.1145/3073408, https://doi.org/10.1145/3073408
4. Abdulla, P.A., Atig, M.F., Jonsson, B., Ngo, T.P.: Optimal stateless model checking under the release-acquire semantics. Proc. ACM Program. Lang. **2**(OOPSLA), 135:1–135:29 (2018). https://doi.org/10.1145/3276505, https://doi.org/10.1145/3276505
5. Bouajjani, A., Emmi, M.: Bounded phase analysis of message-passing programs. In: Flanagan, C., König, B. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7214, pp. 451–465. Springer (2012). https://doi.org/10.1007/978-3-642-28756-5_31, https://doi.org/10.1007/978-3-642-28756-5_31
6. Cirisci, B., Enea, C., Farzan, A., Mutluergil, S.O.: A pragmatic approach to stateful partial order reduction https://arxiv.org/abs/2211.11942
7. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite state concurrent systems using temporal logic specifications: A practical approach. In: Wright, J.R., Landweber, L., Demers, A.J., Teitelbaum, T. (eds.) Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 1983. pp. 117–126. ACM Press (1983). https://doi.org/10.1145/567067.567080, https://doi.org/10.1145/567067.567080
8. Clarke, E.M., Grumberg, O., Minea, M., Peled, D.A.: State space reduction using partial order techniques. Int. J. Softw. Tools Technol. Transf. **2**(3), 279–287 (1999). https://doi.org/10.1007/s100090050035, https://doi.org/10.1007/s100090050035
9. Emmi, M., Enea, C.: Violat: Generating tests of observational refinement for concurrent objects. In: Dillig, I., Tasiran, S. (eds.) Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II. Lecture Notes in Computer Science, vol. 11562, pp. 534–546. Springer (2019). https://doi.org/10.1007/978-3-030-25543-5_30, https://doi.org/10.1007/978-3-030-25543-5_30
10. Emmi, M., Qadeer, S., Rakamaric, Z.: Delay-bounded scheduling. In: Ball, T., Sagiv, M. (eds.) Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011. pp. 411–422. ACM (2011). https://doi.org/10.1145/1926385.1926432, https://doi.org/10.1145/1926385.1926432

11. Farzan, A., Vandikas, A.: Automated hypersafety verification. In: Dillig, I., Tasiran, S. (eds.) Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I. Lecture Notes in Computer Science, vol. 11561, pp. 200–218. Springer (2019). https://doi.org/10.1007/978-3-030-25540-4_11, https://doi.org/10.1007/978-3-030-25540-4_11
12. Farzan, A., Vandikas, A.: Reductions for safety proofs. Proc. ACM Program. Lang. **4**(POPL), 13:1–13:28 (2020). https://doi.org/10.1145/3371081, https://doi.org/10.1145/3371081
13. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: Palsberg, J., Abadi, M. (eds.) Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005. pp. 110–121. ACM (2005). https://doi.org/10.1145/1040305.1040315, https://doi.org/10.1145/1040305.1040315
14. Garg, P., Neider, D., Madhusudan, P., Roth, D.: Learning invariants using decision trees and implication counterexamples. In: Bodík, R., Majumdar, R. (eds.) Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016. pp. 499–512. ACM (2016). https://doi.org/10.1145/2837614.2837664, https://doi.org/10.1145/2837614.2837664
15. Godefroid, P.: Using partial orders to improve automatic verification methods. In: Clarke, E.M., Kurshan, R.P. (eds.) Computer-Aided Verification, Proceedings of a DIMACS Workshop 1990, New Brunswick, New Jersey, USA, June 18-21, 1990. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 3, pp. 321–340. DIMACS/AMS (1990). https://doi.org/10.1090/dimacs/003/21, https://doi.org/10.1090/dimacs/003/21
16. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem, Lecture Notes in Computer Science, vol. 1032. Springer (1996). https://doi.org/10.1007/3-540-60761-7, https://doi.org/10.1007/3-540-60761-7
17. Godefroid, P.: Model checking for programming languages using verisoft. In: Lee, P., Henglein, F., Jones, N.D. (eds.) Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15-17 January 1997. pp. 174–186. ACM Press (1997). https://doi.org/10.1145/263699.263717, https://doi.org/10.1145/263699.263717
18. Godefroid, P., Holzmann, G.J., Pirottin, D.: State-space caching revisited. Formal Methods Syst. Des. **7**(3), 227–241 (1995). https://doi.org/10.1007/BF01384077, https://doi.org/10.1007/BF01384077
19. Godefroid, P., Pirottin, D.: Refining dependencies improves partial-order verification methods (extended abstract). In: Courcoubetis, C. (ed.) Computer Aided Verification, 5th International Conference, CAV '93, Elounda, Greece, June 28 - July 1, 1993, Proceedings. Lecture Notes in Computer Science, vol. 697, pp. 438–449. Springer (1993). https://doi.org/10.1007/3-540-56922-7_36, https://doi.org/10.1007/3-540-56922-7_36
20. Godefroid, P., Wolper, P.: Using partial orders for the efficient verification of deadlock freedom and safety properties. Formal Methods Syst. Des. **2**(2), 149–164 (1993). https://doi.org/10.1007/BF01383879, https://doi.org/10.1007/BF01383879

21. Gramoli, V.: More than you ever wanted to know about synchronization: synchrobench, measuring the impact of the synchronization on concurrent algorithms. In: Cohen, A., Grove, D. (eds.) Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, San Francisco, CA, USA, February 7-11, 2015. pp. 1–10. ACM (2015). https://doi.org/10.1145/2688500.2688501, https://doi.org/10.1145/2688500.2688501
22. He, J., Hoare, C.A.R., Sanders, J.W.: Data refinement refined. In: Robinet, B., Wilhelm, R. (eds.) ESOP 86, European Symposium on Programming, Saarbrücken, Federal Republic of Germany, March 17-19, 1986, Proceedings. Lecture Notes in Computer Science, vol. 213, pp. 187–196. Springer (1986). https://doi.org/10.1007/3-540-16442-1_14, https://doi.org/10.1007/3-540-16442-1_14
23. Hoare, C.A.R., He, J., Sanders, J.W.: Prespecification in data refinement. Inf. Process. Lett. **25**(2), 71–76 (1987). https://doi.org/10.1016/0020-0190(87)90224-9, https://doi.org/10.1016/0020-0190(87)90224-9
24. Holzmann, G.J., Peled, D.A.: An improvement in formal verification. In: Hogrefe, D., Leue, S. (eds.) Formal Description Techniques VII, Proceedings of the 7th IFIP WG6.1 International Conference on Formal Description Techniques, Berne, Switzerland, 1994. IFIP Conference Proceedings, vol. 6, pp. 197–211. Chapman & Hall (1994)
25. Katz, S., Peled, D.A.: Verification of distributed programs using representative interleaving sequences. Distributed Comput. **6**(2), 107–120 (1992). https://doi.org/10.1007/BF02252682, https://doi.org/10.1007/BF02252682
26. Kokologiannakis, M., Vafeiadis, V.: HMC: model checking for hardware memory models. In: Larus, J.R., Ceze, L., Strauss, K. (eds.) ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020. pp. 1157–1171. ACM (2020). https://doi.org/10.1145/3373376.3378480, https://doi.org/10.1145/3373376.3378480
27. Kokologiannakis, M., Vafeiadis, V.: Genmc: A model checker for weak memory models. In: Silva, A., Leino, K.R.M. (eds.) Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I. Lecture Notes in Computer Science, vol. 12759, pp. 427–440. Springer (2021). https://doi.org/10.1007/978-3-030-81685-8_20, https://doi.org/10.1007/978-3-030-81685-8_20
28. Lauterburg, S., Karmani, R.K., Marinov, D., Agha, G.: Evaluating ordering heuristics for dynamic partial-order reduction techniques. In: Rosenblum, D.S., Taentzer, G. (eds.) Fundamental Approaches to Software Engineering, 13th International Conference, FASE 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6013, pp. 308–322. Springer (2010). https://doi.org/10.1007/978-3-642-12029-9_22, https://doi.org/10.1007/978-3-642-12029-9_22
29. Liskov, B., Wing, J.M.: A behavioral notion of subtyping. ACM Trans. Program. Lang. Syst. **16**(6), 1811–1841 (1994). https://doi.org/10.1145/197320.197383, https://doi.org/10.1145/197320.197383
30. Mazurkiewicz, A.W.: Trace theory. In: Brauer, W., Reisig, W., Rozenberg, G. (eds.) Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part II, Proceedings of an Advanced Course, Bad Honnef, Germany, 8-19 Septem-

ber 1986. Lecture Notes in Computer Science, vol. 255, pp. 279–324. Springer (1986). https://doi.org/10.1007/3-540-17906-2_30, https://doi.org/10.1007/3-540-17906-2_30
31. Miltner, A., Padhi, S., Millstein, T.D., Walker, D.: Data-driven inference of representation invariants. In: Donaldson, A.F., Torlak, E. (eds.) Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020. pp. 1–15. ACM (2020). https://doi.org/10.1145/3385412.3385967, https://doi.org/10.1145/3385412.3385967
32. Musuvathi, M., Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs. In: Ferrante, J., McKinley, K.S. (eds.) Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007. pp. 446–455. ACM (2007). https://doi.org/10.1145/1250734.1250785, https://doi.org/10.1145/1250734.1250785
33. Neele, T., Wijs, A., Bosnacki, D., van de Pol, J.: Partial-order reduction for GPU model checking. In: Artho, C., Legay, A., Peled, D. (eds.) Automated Technology for Verification and Analysis - 14th International Symposium, ATVA 2016, Chiba, Japan, October 17-20, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9938, pp. 357–374 (2016). https://doi.org/10.1007/978-3-319-46520-3_23, https://doi.org/10.1007/978-3-319-46520-3_23
34. Peled, D.A.: All from one, one for all: on model checking using representatives. In: Courcoubetis, C. (ed.) Computer Aided Verification, 5th International Conference, CAV '93, Elounda, Greece, June 28 - July 1, 1993, Proceedings. Lecture Notes in Computer Science, vol. 697, pp. 409–423. Springer (1993). https://doi.org/10.1007/3-540-56922-7_34, https://doi.org/10.1007/3-540-56922-7_34
35. Plotkin, G.D.: LCF considered as a programming language. Theor. Comput. Sci. **5**(3), 223–255 (1977). https://doi.org/10.1016/0304-3975(77)90044-5, https://doi.org/10.1016/0304-3975(77)90044-5
36. Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: Halbwachs, N., Zuck, L.D. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3440, pp. 93–107. Springer (2005). https://doi.org/10.1007/978-3-540-31980-1_7, https://doi.org/10.1007/978-3-540-31980-1_7
37. Queille, J., Sifakis, J.: Specification and verification of concurrent systems in CESAR. In: Dezani-Ciancaglini, M., Montanari, U. (eds.) International Symposium on Programming, 5th Colloquium, Torino, Italy, April 6-8, 1982, Proceedings. Lecture Notes in Computer Science, vol. 137, pp. 337–351. Springer (1982). https://doi.org/10.1007/3-540-11494-7_22, https://doi.org/10.1007/3-540-11494-7_22
38. Sharma, R., Aiken, A.: From invariant checking to invariant inference using randomized search. Formal Methods Syst. Des. **48**(3), 235–256 (2016). https://doi.org/10.1007/s10703-016-0248-5, https://doi.org/10.1007/s10703-016-0248-5
39. Tasharofi, S., Karmani, R.K., Lauterburg, S., Legay, A., Marinov, D., Agha, G.: Transdpor: A novel dynamic partial-order reduction technique for testing actor programs. In: Giese, H., Rosu, G. (eds.) Formal Techniques for Distributed

Systems - Joint 14th IFIP WG 6.1 International Conference, FMOODS 2012 and 32nd IFIP WG 6.1 International Conference, FORTE 2012, Stockholm, Sweden, June 13-16, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7273, pp. 219–234. Springer (2012). https://doi.org/10.1007/978-3-642-30793-5_14, https://doi.org/10.1007/978-3-642-30793-5_14
40. Valmari, A.: Stubborn sets for reduced state space generation. In: Rozenberg, G. (ed.) Advances in Petri Nets 1990 [10th International Conference on Applications and Theory of Petri Nets, Bonn, Germany, June 1989, Proceedings]. Lecture Notes in Computer Science, vol. 483, pp. 491–515. Springer (1989). https://doi.org/10.1007/3-540-53863-1_36, https://doi.org/10.1007/3-540-53863-1_36
41. Visser, W., Pasareanu, C.S., Khurshid, S.: Test input generation with java pathfinder. In: Avrunin, G.S., Rothermel, G. (eds.) Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2004, Boston, Massachusetts, USA, July 11-14, 2004. pp. 97–107. ACM (2004). https://doi.org/10.1145/1007512.1007526, https://doi.org/10.1145/1007512.1007526
42. Yang, Y., Chen, X., Gopalakrishnan, G., Kirby, R.M.: Efficient stateful dynamic partial order reduction. In: Havelund, K., Majumdar, R., Palsberg, J. (eds.) Model Checking Software, 15th International SPIN Workshop, Los Angeles, CA, USA, August 10-12, 2008, Proceedings. Lecture Notes in Computer Science, vol. 5156, pp. 288–305. Springer (2008). https://doi.org/10.1007/978-3-540-85114-1_20, https://doi.org/10.1007/978-3-540-85114-1_20
43. Yi, X., Wang, J., Yang, X.: Stateful dynamic partial-order reduction. In: Liu, Z., He, J. (eds.) Formal Methods and Software Engineering, 8th International Conference on Formal Engineering Methods, ICFEM 2006, Macao, China, November 1-3, 2006, Proceedings. Lecture Notes in Computer Science, vol. 4260, pp. 149–167. Springer (2006). https://doi.org/10.1007/11901433_9, https://doi.org/10.1007/11901433_9

A Optimizations

The re-traversals used to track dependencies in **current** sets are time consuming, but several optimizations can be applied to DE-S-POR and DL-S-POR. First optimization is the only one that is applied to both of these algorithms, which is traversing each transition only once rather than traversing all the executions after some state s for calculating A_s at line 16 and line 27, respectively. This approach is much cheaper in general due to the high ratio of visited states and can be achieved easily by disabling (re-)visiting an already visited state during the same re-traversal. For guaranteeing soundness, we need to update **UpdateCurr** function for re-traversals in DE-S-POR to find not just the last transition (as it is stated at line 20) but all the transitions that their actions are dependent on a for conservatively updating **current** sets for all such transitions (at line 22). Such an update is needed since some of the different orderings between transitions can be missed during these optimized re-traversals. On the other hand, DL-S-POR does not need such an update for guaranteeing soundness after this optimization as it already updates **current** sets conservatively (see line 28). Rest of the optimizations in this section are only applied to DL-S-POR.

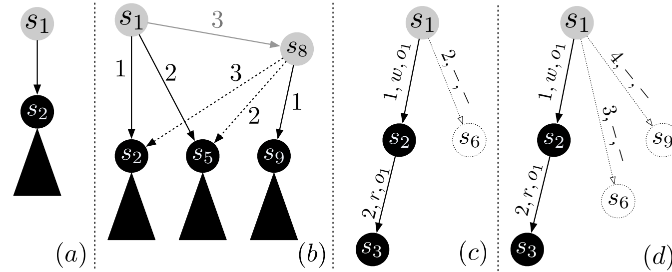

Fig. 11. Optimizations to avoid re-traversing the state space. We use the same conventions as in Figure 3.

The simplest optimization for DL-S-POR is not performing a traversal from the last successor of s, called s', when backtracking to s. Then, s.**done** becomes equal to $safeSet(s)$ after adding the thread t leading to s' (see the first disjunct at line 21). This implies that no re-traversal is initiated for a state s that is reducible or it is irreducible but has only one enabled transition; for an example see Figure 11(a).

Second, a traversal from a successor of s does not have to be performed if after adding t to s.**done**, all the threads enabled in s and not already in s.**done** lead to already visited states (see the second disjunct at line 21). The successors of those transitions may have not been pushed to the stack so they are added to the current LTS L_r (see line 22). This optimization also relies on initiating re-traversals of the state space only when backtracking from a new state (see the condition at line 11). Stopping the traversal is sound because there are no

new states to explore from s and the exploration from s is already complete. A scenario where this optimization is enabled is shown in Figure 11(b). When backtracking to s_8 for the first time (from s_9), all the other threads enabled in s_8 lead to an already visited state. The re-traversal of the state space starting in s_9 is stopped since the exploration from s_8 is already complete (before returning, the transitions of threads 2 and 3 from s_8 are added to the current LTS).

While these optimizations for DL-S-POR stop re-traversals altogether, several optimizations concern stopping a traversal early before reaching every state. In a concrete implementation, the declarative definitions at lines 27-28 translate to a (DFS) traversal of all the transitions starting in s' and populating $s.\mathbf{current}[t]$ as new dependent transitions are found. This traversal can be stopped as soon as $s.\mathbf{current}[t]$ becomes "complete", i.e., it stores all threads in $\mathit{safeSet}(s)$. An example is shown in Figure 11(c) where the traversal starting in s_2 can stop immediately after the first transition since only threads 1 and 2 are enabled in s_1. Similarly, the traversal can be stopped immediately as $s.\mathbf{current}[t]$ contains a thread which is not enabled in s. In this case, $s.\mathbf{backtrack}$ will anyway be updated conservatively to include all the threads in $\mathit{safeSet}(s)$. An example is shown in Figure 11(d).

Only the first optimization is adapted in the eager version and it requires an update for soundness. Rest of these optimizations are not even applicable in the eager version, or if they are, they are much harder to apply. Here, we take advantage of having to compute dependencies using forward traversals that explore transitions starting in a certain state as opposed to a backward traversal of states in the stack.

B More Experimental Results

We present more results obtained from the experiments for pair-wise comparison between the random and sequential enumeration strategies. We report the average and minimum time over the different instances as follows:

- The first set of experiments are for computing all reachable states, only with DL-S-POR. For S-POR, the enumeration strategy is not important since there is no dynamic computation of backtrack sets. We present separate figures for each different class of clients based on the number of invocations to methods that lead to bugs: (1) none of the invocations, (Figure 12), (2) just a single invocation (Figure 13) (3) half of the invocations (Figure 14) and (4) all of the invocations (Figure 15).
- The second set of experiments are for computing reachable states only until the first error, both with S-POR (Figures from 16 to 18) and DL-S-POR (Figures from 19 to 21). As the purpose is to find the first bug, clients in which none of the invocations are buggy, are not included in this set of experiments. Figures for the rest of the classes of clients are represented with the same order and visualization as in the first set. Differently from the first set of figures, the figures in second set are in logarithmic scale.

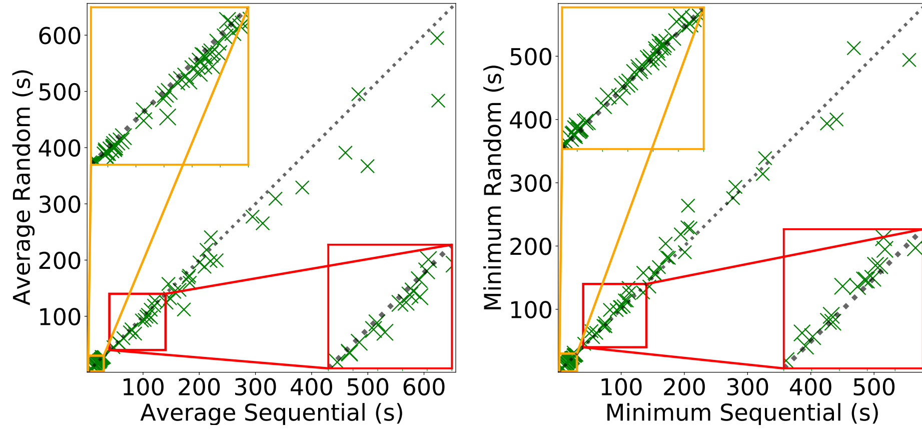

Fig. 12. ALL STATES \ DL-S-POR \ BUG: NO INVOCATION
Time comparison between sequential and random strategies when DL-S-POR computes all states, using clients in which none of the invocations are buggy.

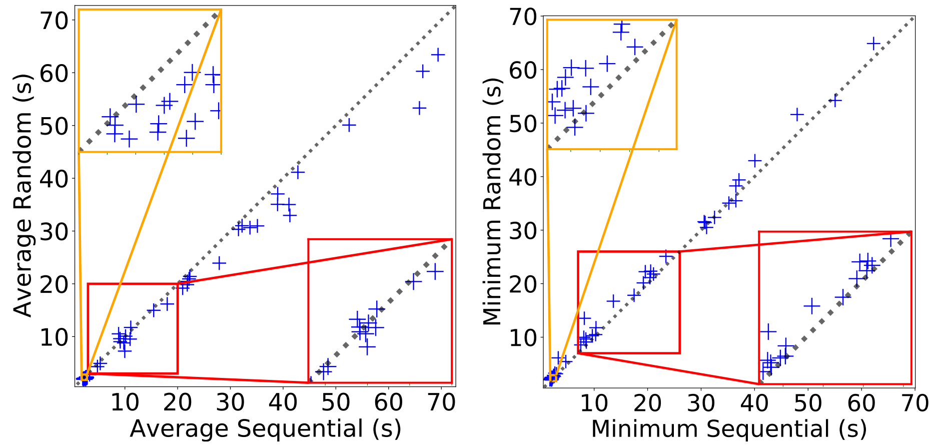

Fig. 13. ALL STATES \ DL-S-POR \ BUG: SINGLE INVOCATION
Time comparison between sequential and random strategies when DL-S-POR computes all states, using clients in which only a single invocation is buggy.

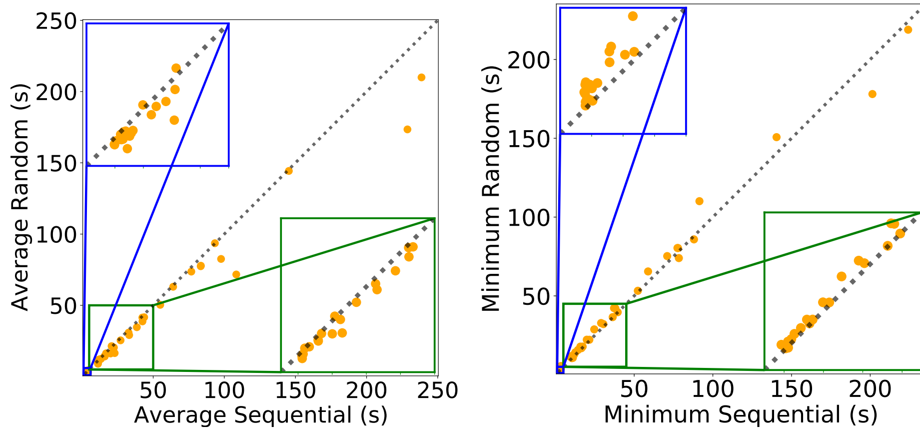

Fig. 14. ALL STATES \ DL-S-POR \ BUG: $\frac{1}{2}$ OF THE INVOCATIONS
Time comparison between sequential and random strategies when DL-S-POR computes all states, using clients in which half of the invocations are buggy.

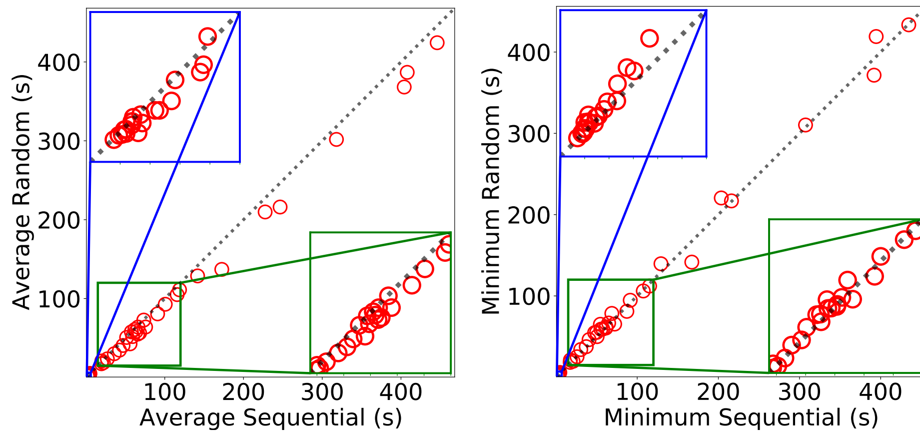

Fig. 15. ALL STATES \ DL-S-POR \ BUG: ALL INVOCATIONS
Time comparison between sequential and random strategies when DL-S-POR computes all states, using clients in which all of the invocations are buggy.

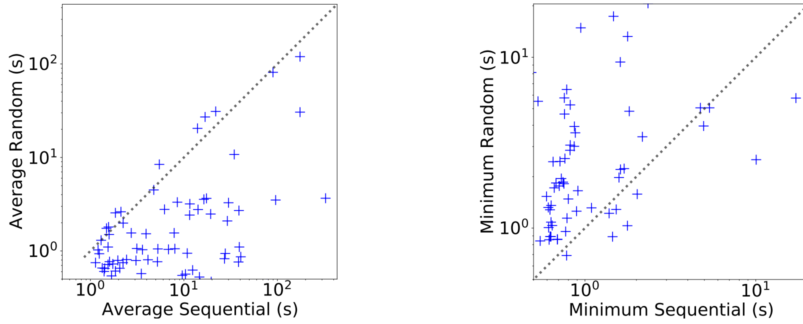

Fig. 16. FIRST ERROR \ S-POR \ BUG: SINGLE INVOCATION
Time comparison (log scale) between sequential and random strategies when S-POR enumerates states only until the first error, using clients in which only a single invocation is buggy.

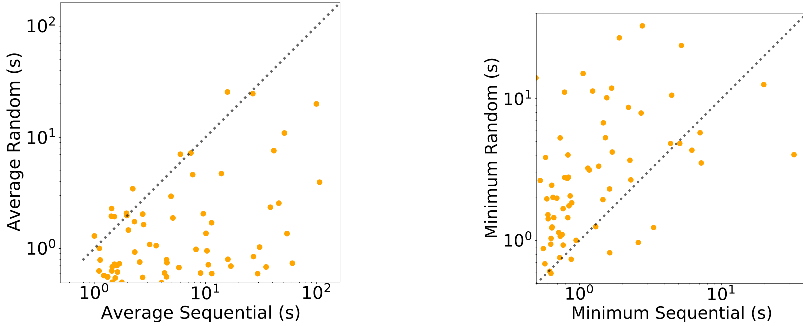

Fig. 17. FIRST ERROR \ S-POR \ BUG: ½ OF THE INVOCATIONS
Time comparison (log scale) between sequential and random strategies when S-POR enumerates states only until the first error, using clients in which half of the invocations are buggy.

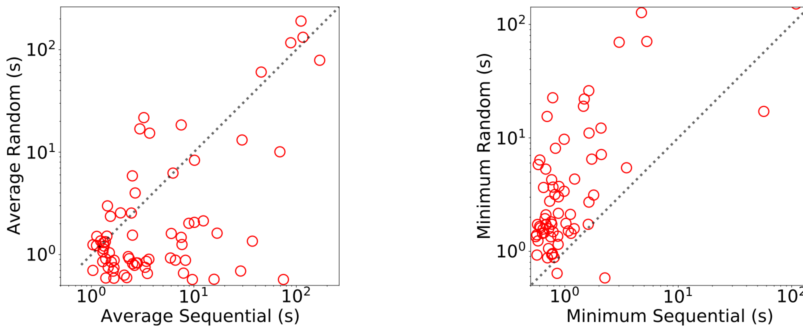

Fig. 18. FIRST ERROR \ S-POR \ BUG: ALL INVOCATIONS
Time comparison (log scale) between sequential and random strategies when S-POR enumerates states only until the first error, using clients in which all of the invocations are buggy.

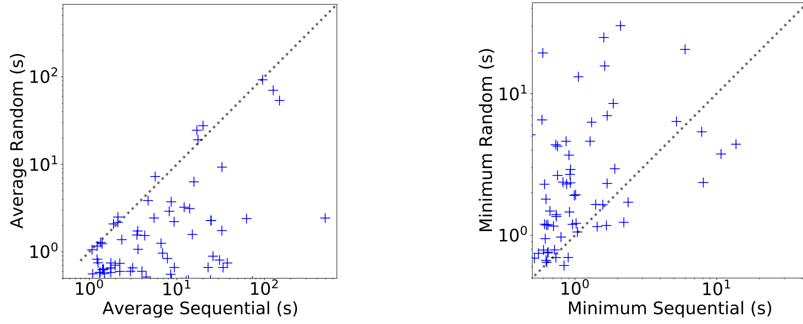

Fig. 19. FIRST ERROR \ DL-S-POR \ BUG: SINGLE INVOCATION
Time comparison (log scale) between sequential and random strategies when DL-S-POR enumerates states only until the first error, using clients in which only a single invocation is buggy.

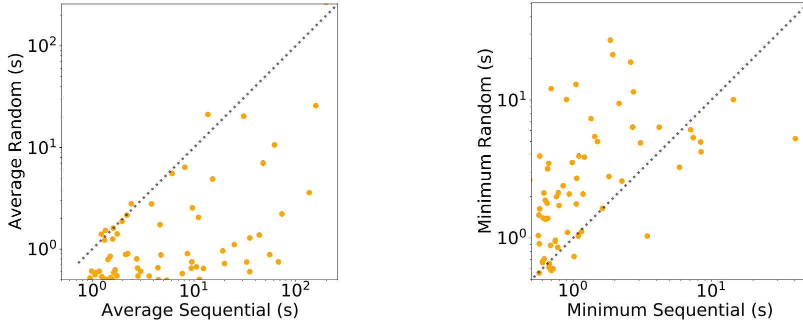

Fig. 20. FIRST ERROR \ DL-S-POR \ BUG: ½ OF THE INVOCATIONS
Time comparison (log scale) between sequential and random strategies when DL-S-POR enumerates states only until the first error, using clients in which half of the invocations are buggy.

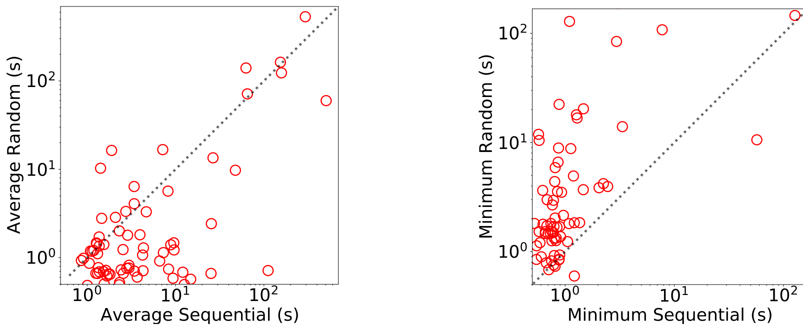

Fig. 21. FIRST ERROR \ DL-S-POR \ BUG: ALL INVOCATIONS
Time comparison (log scale) between sequential and random strategies when DL-S-POR enumerates states only until the first error, using clients in which all of the invocations are buggy.