Open access • Book Chapter • DOI:10.1007/978-3-540-85114-1_20

Efficient Stateful Dynamic Partial Order Reduction — Source link

Yu Yang, Xiaofang Chen, Ganesh Gopalakrishnan, Robert M. Kirby

Institutions: University of Utah

Published on: 10 Aug 2008 - International workshop on Model Checking Software

Topics: Partial order reduction, Stateful firewall, Stateless protocol, Model checking and State (computer science)

Efficient Stateful Dynamic Partial Order Reduction*

Yu Yang Xiaofang Chen Ganesh Gopalakrishnan Robert M. Kirby

School of Computing, University of Utah
Salt Lake City, UT 84112, U.S.A.

Abstract. In applying stateless model checking methods to realistic multithreaded programs, we find that stateless search methods are ineffective in practice, even with dynamic partial order reduction (DPOR) enabled. To solve the inefficiency of stateless runtime model checking, this paper makes two related contributions. The first contribution is a novel and conservative light-weight method for storing abstract states at runtime to help avoid redundant searches. The second contribution is a stateful dynamic partial order reduction algorithm (SDPOR) that avoids a potential unsoundness when DPOR is naively applied in the context of stateful search. Our stateful runtime model checking approach combines light-weight state recording with SDPOR, and strikes a good balance between state recording overheads, on one hand, and the elimination of redundant searches, on the other hand. Our experiments confirm the effectiveness of our approach on several multithreaded benchmarks in C, including some practical programs.

1 Introduction

Despite all the advances in developing new concurrency abstractions, explicit thread programming using thread libraries remains one of the most practical ways of realizing concurrent programs that take advantage of multiple cores. Many high level concurrency abstractions (*e.g.*, software transaction memories) also require the use of threads for their implementation. Unfortunately, it is not easy to write bug-free thread programs [1]. In this paper, we focus on the efficient checking of a given multithreaded program for safety violations over *all possible interleavings* on specific inputs.

Runtime model checking [2,3] is a promising method for bug detection. As model building, extraction, and model maintenance are expensive to carry out for thread programs written in practice, we believe in the importance of developing efficient runtime checking methods, as pioneered in [2]. However, even when running under specific inputs, the number of interleavings of a concurrent program can grow astronomically due to their internal concurrency.

Much of the interleaving explosion that occurs in practice during stateless runtime model checking can be attributed to redundant searches from already visited states. The example in Figure 1 illustrates this problem. This program has

* Supported in part by NSF award CNS00509379, Microsoft HPC Institute Program, and SRC Contract 2005-TJ-1318.

```
    const int N = 64;
    int d = 0;

        thread1:                          thread2:
    local int i = 0;                  local int j = 0;
L0: while (i < N){                M0:   while (j < N){
L1:   atomic{                     M1:     atomic{
        d = d + i;                M2:       d = d - j;
        assert(d % 5 != 4)                  assert(d % 5 != 4);
      }                                   }
L2:   i = i + 5;                  M4:     j = j + 2;
L3: }                             M5:   }
```

Fig. 1. A simple example for illustrating the idea

two threads that, in their own `atomic` blocks that are nested within loops, write to a shared variable `d`. Stateless search methods cannot handle this example even with the help of dynamic partial order reduction (DPOR) [4]. This is because: (i) the number of interleavings grows exponentially with respect to the number of loop iterations; (ii) working under stateless DPOR, at any reached state where both threads are enabled, there exists no non-trivial persistent set.

Consequently, with stateless search, many states of this program are re-visited multiple times via different interleavings. For example, given `thread1` at `L1` and `thread2` at `M1`, whether `thread1` executing `L1,L2` followed by `thread2` executing `M1,M2`, or vice versa, the program reaches the same state. Figure 2 illustrates this, where `T1` represents `thread1` and `T2` represents `thread2` (Note: The dotted states are the intermediate states attained after executing the visible operation of a transition;this detail illustrates a convention introduced in Section 2.) Failing to detect visited states makes the stateless search methods repeatedly explore visited state spaces, which results in very low efficiency.

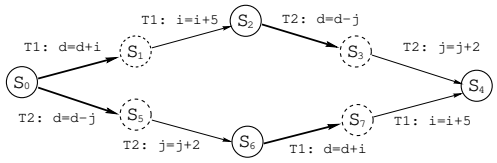

Fig. 2. Two different executions of the program in Figure 1 lead to the same state

While one straightforward solution that avoids redundant searches involves the use of visited states maintained in a hash table, this method, however, is complicated owing to the difficulty of capturing the states of realistic multithreaded program at runtime. This is especially true for programs written in program

languages such as C/C++. Although there have been model checkers such as CMC [5] and Java PathFinder [6] that have attempted such program state capture, these approaches are quite heavy-weight. For example, if we take CMC's approach, we need to capture the state of the kernel space plus the user space. Alternately, if we follow Java PathFinder's approach, we will have to build a virtual machine for C/C++ programs. Is there a light-weight approach to recording the states of concurrent programs at runtime? If we have such an approach, how do we combine it with partial order reduction techniques soundly? In this paper, we solve these problems in the context of terminating multithreaded programs. We make the following contributions:

- We propose a novel light-weight scheme for capturing the *local* states of threads. We observe that while capturing the entire state of a realistic program at runtime is difficult and expensive, capturing the *changes* between two successive local states of a thread can be easy and inexpensive. Based on this observation, we abstract local states of threads with IDs, and try to discover the same local state of a thread among different executions by tracking the changes (i.e., "deltas") between successive local states of threads. While an actual total system state of a thread program with N threads would be a tuple $(g, (l_1, \ldots, l_N), (p_1, \ldots, p_N))$ where g is the global state, l_k are the actual thread local states and p_k are the actual thread PCs, an *abstract* state would be $(g, (i_1, \ldots, i_N), (p_1, \ldots, p_N))$ where i_k are *IDs* we assign for thread local states. These IDs are computed in a conservative way based on the sequence of deltas that each thread undergoes, as explained in Section 3.
- We present a stateful dynamic partial order reduction (SDPOR) algorithm, which combines our light-weight runtime state capturing approach with dynamic partial order reduction. By introducing states in dynamic partial order reduction, an obvious soundness problem is not updating the backtrack set along a new path that revisits a state. To solve this problem efficiently, we dynamically construct a *visible operation dependency graph* while performing the search. When a visited state is encountered, we compute the summary of the visited sub-state-space using the visible operation dependency graph. With the summary, we conservatively update the backtrack sets of states and guarantee the soundness of our approach.
- We have implemented SDPOR within our runtime model checker `Inspect` [7], and evaluated our approach on a set of multithreaded C benchmarks. The experiments show that SDPOR is much more effective than the stateless DPOR.

The rest of the paper is organized as follows. We introduce the background definitions in Section 2. In Section 3, we describe how local states can be captured in a light-weight and conservative manner. Section 4 presents how the DPOR algorithm can be adapted, with the stateful search. Sections 5 and 6 then present the implementation details and the experimental results. An the end, we discuss related work and conclude the paper.

2 Background Definitions

In this section, we define the notations we employ in the rest of the paper, following the style of [4]. We consider a terminating multithreaded program with a fixed number of sequential threads as a state transition system. We use $Tid = \{1, \ldots, n\}$ to denote the set of thread identities. Threads communicate with each other via *global* objects which are visible to all threads. The operations on global objects are called *visible* operations, while thread local variable updates and PC updates are *invisible* operations. The total system state (S), the program counters of the threads (PCs), and the local states of threads ($Locals$) are now defined:

$$\begin{aligned} S &\subseteq Global \times Locals \times PCs \\ PCs &= Tid \rightarrow PC \\ Locals &= Tid \rightarrow Local \end{aligned}$$

Here, *Global* is the state of global objects, *Local* the local state of any thread, and PC the program counter of any thread. For $s \in S$, we use $g(s) \in Global$ to denote the state of global objects in s, $l(s) \in Locals$ to denote the local state component, and $l_\tau(s) \in Locals(\tau)$ to denote the local state of thread τ in s. For $ls \in Locals$, we write $ls[h := l]$ to denote the map that is identical to ls except that it maps the thread h to the local state l.

A transition $t : S \rightarrow S$ advances the program from one state to a subsequent state. More specifically, it starts with one visible operation, followed by a finite sequence of zero or more invisible operations of the same thread, and ends just before the next visible operation of the same thread. For instance, in Figure 2, `T1: d=d+i` is a visible operation, and `T1: i=i+5` is an invisible operation. We do not consider the read of `i` that occurs within `d=d+i` to be an invisible operation, as it does not change the state of `i`.

We can view a transition t as a composition of the *global* transition t_g and the *local* transition t_l. That is, $t = t_l \circ t_g$ where $t_l, t_g \in S \rightarrow S$. Here, t_g corresponds to the visible operation that the transition t starts with. It updates the state of global objects and the program counter of the thread. t_l corresponds to the finite sequence of invisible operations that follows t_g. It can only affect the local state and the program counter of the thread.

Let $\mathcal{T}$ denote the set of all transitions of a multithreaded program. A transition $t \in \mathcal{T}$ is enabled in a state s if $t(s)$ is defined. If t is enabled in s and $t(s) = s'$, we use $s \xrightarrow{t} s'$ to mean that s' is the successor of s by executing transition t. We use $tid(t)$ to denote the identity of the thread that executes t. Obviously we have $tid(t) \in Tid$.

The behavior of a multithreaded program P is given by a transition system $M = (S, s_0, \Gamma)$, where s_0 is the initial state, and $\Gamma \subseteq S \times S$ is the transition relation. $(s, s') \in \Gamma$ iff $\exists t \in \mathcal{T} : s \xrightarrow{t} s'$.

3 Capturing Local States of Threads

Although the local states of threads are not easy to capture precisely at runtime, we observe that in many cases, the changes δ between successive local states are easy to capture. For example, due to the correlations among global objects [8], it is commonly the case that there exist sequences of transitions in which each transition has only the visible operation component, with the invisible operation component being absent. In this case, the local states of threads do not change. It is also common that the changes of local states only involve several local objects and are easy to capture. As an example, in the program of Figure 1, the local state change of `thread1` between two successive executions of the atomic statement labeled `L1` only involves the local object `i`. Likewise, the local state change of `thread2` between two successive executions of the atomic block at `M1` only involves the local object `j`. This motivates us to capture the local states of threads by tracking the changes among local states.

We now detail our algorithm for capturing local states of threads at runtime, in the context of a depth first search of the state space of the threads. The key idea of the algorithm is to represent each local state of a thread with an abstract ID, and to *link* these IDs by tracking changes between successive local states of threads. This scheme helps conservatively determine whether local states of threads are repeating across different executions.

Let $LocalId$ denote the set of local state IDs (natural numbers). We define the *abstract state* of a multithreaded program formally as follows:

$$\begin{aligned} S_a &\subseteq Global \times Locals_a \times PCs \\ Locals_a &= Tid \rightarrow LocalId \\ LocalId &\subset \mathbb{N} \end{aligned}$$

With the local state IDs, a multithreaded program can be represented as a transition system $M_a = (S_s, s_{0_a}, \Gamma_a)$, where s_{0_a} is the initial state of the program, and $\Gamma_a \subseteq S_a \times S_a$ is the transition relation. Note that because of our conservative state maintenance scheme which we present later in this section, there could be more than one abstract state associated with a real state.

Let s_a be an abstract state in S_a. When the context is clear, we still use $g(s_a) \in Global$ to denote the global state of s_a, and use $ls(s_a) \in Locals_a$ to denote the local states identities. We use $lid_\tau(s_a) \in LocalId$ to denote the assigned local state identity of thread τ. For $ls_a \in Locals_a$, we write $ls_a[\tau := x]$ to denote that the map that is identical to ls_a except that it maps the thread τ to the local state identity x.

As the state of global objects are in general easy to capture, we do not abstract the states of global objects. Let $s_a \in S_a$ be an abstract state and s be its corresponding state in S. We have $g(s_a) = g(s)$. Similarly, we also have $s_a.PCs = s.PCs$.

Let $s, s' \in S$ be two states, and t be a transition such that $s \xrightarrow{t} s'$. Let $\tau = tid(t)$. We define the changes of the local state of thread τ between s and s' as $\delta_\tau = l_\tau(s') \setminus l_\tau(s)$. We use δ_ε to represent that the local state does not change

for a thread. That is, for the thread τ in the above, $l_\tau(s') = l_\tau(s)$. Also, we write $\delta_\perp$ to denote that the local state changes are unknown. $\delta_\perp$ is used when it is hard to capture the local state changes, e.g. when the transition t_l involves calls to library routines, etc. We use Δ to denote the set of all possible local state changes (δs) for all threads in the program.

In order to detect that the same local state of thread is appearing in different executions of the multithreaded program, we maintain a *local state hash table* for each thread of the program. The local state hash table records the IDs of the local states that have been visited, as well as the changes between two successive local states. In more detail, for each thread τ, we have a local state hash table L_τ to store the IDs of the visited local states of τ. $L_\tau : LocalId \times \Delta \rightarrow LocalId$ is a mapping from a local state IDs plus the change to a local state, to a potentially new local state ID. However, if L_τ already contains the domain point, then the local state ID already in the hash table is returned. We use L to denote the set of local state hash tables for all threads.

Our basic search algorithm with abstract state recording is presented in Figures 3 and 4. The final SDPOR algorithm in Section 4 will build on this algorithm. Figure 3 shows DFS, a recursive procedure for depth-first search of the state space. DFS calls NEXTLOCAL of Figure 4 to compute the local state IDs of a thread. The main data structures used are:

- A hash table H to store all program states $s \in S_a$ that have already been visited during the search.
- For each thread τ, we have a local state hash table L_τ to store the identities of the visited local states of τ.

DFS of Figure 3 has four parameters: the abstract state hash table H, the local state hash tables L, the current state s, and finally s_a, which is the abstract state of s. Starting from the initial state, DFS recursively explores the successor states of all states encountered during the search, provided that the correspondent abstract state is not in the hash table. For each visited state, DFS stores the correspondent abstract state in the hash table H. Each time we reach a state s' by executing a transition t which is enabled in a state s, we will compute the abstract state of s' (line 7-9 of Figure 3), and recursively call DFS to explore the next level of the state space.

Figure 4 shows the algorithm for computing the local state identity of a thread. In the procedure NEXTLOCAL, we consider four possible cases:

- If the local state change is difficult to capture precisely, we simply return a new local state ID x.
- If the local state does not change (i.e., $\delta_\tau = \delta_\varepsilon$), the same ID is returned.
- If the hash table L_τ already has an entry for $(i, \delta_\tau) \rightarrow y$, then we return y as the ID.
- Otherwise, we return a new local state ID x, and at the same time add an entry $\langle (i, \delta_\tau) \rightarrow x \rangle$ to L_τ.

Now with DFS and NEXTLOCAL, we have the following theorem:

```
1: Initially: H is empty; ∀L_τ ∈ L : L_τ is empty; DFS(H, L, s_0, s_{0_a});

2: DFS(H, L, s, s_a) {
3:    if (s_a ∈ H) return;
4:    enter s_a in H;
5:    for each transition t that is enabled in s {
6:       let s' ∈ S such that s -t-> s';
7:       let τ = tid(t), δ_τ = l_τ(s') \ l_τ(s);
8:       let x = NEXTLOCAL(L_τ, lid_τ(s_a), δ_τ);
9:       let s'_a ∈ S_a s.t. g(s'_a) = g(s') ∧ ls(s'_a) = ls(s_a)[τ := x] ∧ s'_a.PCs = s'.PCs;
10:      DFS(H, L, s', s'_a);
11:   }
12: }
```

Fig. 3. Depth-first search with a light-weight state capturing scheme

```
1: NEXTLOCAL(L_τ, i, δ_τ) {
2:    let x ∈ LocalId be a unique new local state identity;
3:    if (δ_t = δ_⊥) return x;
4:    if (δ_τ = δ_ε) return  i;
5:    if (∃y : ⟨(i, δ_τ) → y⟩ ∈ L_τ) return y;
6:    add ⟨(i, δ_τ) → x⟩ to L_τ;
7:    return x;
8: }
```

Fig. 4. Computing the local state identity of a thread

Theorem 1. *Let $M = (S, s_0, \Gamma)$ be a multithreaded program. In a depth first search on S following the algorithm of Figure 3, let $s, s' \in S$ be states that can be reached from s_0, and let $s_a, s'_a \in S_a$ be the abstract states corresponding to s and s'. Then $\forall \tau \in Tid : lid_\tau(s_a) = lid_\tau(s'_a) \implies l_\tau(s) = l_\tau(s')$.* □

The detailed proof is in the appendix. Theorem 1 states that to detect the visited states at runtime, instead of capturing the local states of threads in detail, we can conservatively infer the equality of local states using the local state changes δ. In practice, capturing δ is usually much easier than capturing the whole local state. Therefore, the task of explicitly capturing states at runtime can be greatly simplified. In the next section, we show how to combine our approach of state capturing with dynamic partial order reduction.

4 Stateful Dynamic Partial Order Reduction

4.1 Background

Dynamic partial order reduction [4] has been shown as an effective reduction technique in stateless search. In DPOR, given a state s, the persistent set [9] of s is not computed immediately after reaching s. Instead, DPOR explores the

states that can be reached from s using depth-first search, and adds backtrack information into the backtrack set of s while exploring the sub-space that is reachable from s.

In more detail, let t_i be a transition that is enabled at state s. Suppose the model checker first selects t_i to execute at s. Let t_j be a transition which can be enabled with a depth first search (in one or more steps) from s by executing t_i. Then before t_j is executed, DPOR will check whether t_j and t_i are dependent and can be enabled concurrently, i.e. *co-enabled*. If so, $tid(t_j)$ or the id of the thread which t_j is dependent on will be added to the the backtrack set of s if a transition of $tid(t_j)$ is enabled at s. Later, in the process of backtracking, if the state s is found with non-empty backtrack set, DPOR will select one transition t which is enabled at s and $tid(t)$ is in the backtrack set of s, and explore a new branch of the state space by executing t from s; at the same time, $tid(t)$ will be removed from the backtrack set of s.

For convenience, we use the following notations to represent the notions used in DPOR:

- $s.enabled$ denotes the set of transitions that are enabled at s. We say a thread τ is enabled at s if $\exists t \in s.enabled : tid(t) = \tau$.
- $s.backtrack$ refers to the backtrack set of state s, i.e. the set of threads whose transitions are enabled at s but have not been executed, $s.backtrack \subseteq Tid$.
- $s.done$ denotes the set of threads whose transitions are enabled at s and have been executed from s, $s.done \subseteq Tid$.

As DPOR is a *stateless* depth first search, it also suffers from the redundant exploration of the state space as described in Section 1. In the rest of this section, we show how to adapt dynamic partial order reduction in the context of stateful search, and how to combine the state capturing scheme of Section 3 with the stateful dynamic partial order reduction.

4.2 Stateful DPOR

The problem: In a depth first search with DPOR, it seems that if visited states can be recognized, DPOR can simply stop the search at the visited states and start backtracking. However, it is not that simple because the transitions to be executed after the visited states may update the backtrack sets of the states in the search stack. Simple backtracking may result in unsoundness. For example, suppose we have two different executions

$$S_1 = s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \dots \xrightarrow{t_{u-1}} s_u \dots$$

$$S_2 = s_0 \xrightarrow{t'_0} s'_1 \xrightarrow{t'_1} \dots \xrightarrow{t'_{v-1}} s'_v \dots$$

of a program starting from the same state s_0, and s'_v is a visited state, $s'_v = s_u$, $u, v \geq 0$ (a fact also noted in [10]). Also, suppose that S_2 is explored after S_1 in the depth first search with DPOR. Now, for every transition t which is executed after s'_v, the backtrack sets of states $s_0, s'_1, \dots, s'_v$ of S_2 may have to be updated. As a result, if we simply stop exploring the state space after s'_v, we may miss exploring a subset of the state space, i.e., this naïve approach is not sound.

Initial solution: A straightforward way to fix this problem is that when a visited state is encountered, for each state s in the search stack, we update the backtrack set as follows – for all $t \in s.enabled$ where $tid(t) \notin s.done$, add $tid(t)$ into $s.backtrack$. This solves the problem of missing potential backtrack sets. However, it may also have the side effect of introducing too many unnecessary backtrack points which would not be introduced in the stateless DPOR. [1] This side effect may put significant overhead on the stateful approach and make the stateful DPOR run *slower* than the stateless one. Our initial experiments confirmed this conjeture.

Visible operation dependency graph: To avoid unsoundness we employ an efficient mechanism called *visible operation dependency graph*. Let s_v be a visited state encountered in the stateful DPOR. As only the visible operations of transitions determine whether two transitions are dependent or not, our approach is to compute a summary for the state space the element of which can be reachable from s_v. This summary captures all the visible operations that might be executed from s_v in one or more steps. We can use this summary to update the backtrack sets of the states that are in the search stack.

Obviously, computing such a summary for every state is very heavy-weight. However, we observe that with multithreading, the programs are usually designed in such a way that each thread is assigned some specific tasks to get the most benefit out of parallelism. The number of resources that require mutual exclusive accesses, and the number of conditions that threads need to be synchronized are limited, and usually small in number. This implies that *while the number of states of a multithreaded program can be large, the number of visible operations that each thread may execute is limited.*

For instance, for the program of Figure 1, although the number of states can be large, the only visible operation that `thread1` and `thread2` may execute is updating the global object `d`.

Based on this observation, instead of trying to maintain a summary for each state and keep the summaries updated, *we compute the summary dynamically only when a visited state is encountered by looking up the visible operation dependency graph which is constructed dynamically during the search.*

In more detail, let $M = (S, s_0, \Gamma)$ be a multithreaded program. Let $\mathcal{T}$ be the transition set of M. A visible operation dependency graph $G = \langle V, E \rangle$ for M is a directed graph which captures the happen-before relation of visible operations for the traversed state space. Every node $v \in V$ of G is a visible operation. That is, $\forall v \in V : \exists t \in \mathcal{T} : t_g = v$. For each transition sequence $s_1 \xrightarrow{t} s_2 \xrightarrow{t'} s_3$ we encounter during the search, we add a directed edge (t_g, t'_g) into the graph.

In a depth-first search, when a visited state s is encountered, all the states that are reachable from s must have been visited because of the depth first search. Hence, all the visible operations that may be executed in states reachable from

[1] The solution in [10] was this, but the method was experimented only in the context of a custom-built model checker on very small MPI program examples.

s must have been executed. Therefore, we can traverse the visible operation dependency graph to find out all the visible operations that may be executed from some transition in $s.enabled$, and use this as a summary to update the backtrack sets of the states which are in the search stack. As the size of the graph is proportional to the number of visible operations that a multithreaded program may execute, this is a light-weight method for computing summaries of the visited states.

SDPOR: Figure 5 presents our stateful dynamic partial order reduction algorithm (SDPOR). The procedure SDPOR takes three parameters: the search stack S, the state hash table H, and the visible operation dependency graph G. Similar to DPOR, given a multithreaded program P, SDPOR first explores an arbitrary interleaving of the program, and thereafter, continues explore alternative interleavings until all relevant interleavings are explored, i.e., when no backtrack points are in the search stack. The differences between SDPOR and DPOR are:

- SDPOR uses a hash table H to record the visited states (line 9 of Figure 5). When a visited state is encountered, SDPOR conservatively updates the backtrack sets for states in the search stack S, and start backtracking (line 5-7 of Figure 5).
- SDPOR uses a visible operation dependency graph G to dynamically learn the happen-before relation of visible operations during the depth-first search (line 19 of Figure 5). G is used to compute the state summary $\mathcal{U}$ when a visited state is encountered (line 5 of Figure 5).

SDPOR uses UPDATEBACKTRACKSETS of Figure 6 to update the backtrack sets for states in the search stack. UPDATEBACKTRACKSETS is the same as that in the stateless DPOR. We present it here for completeness.

Theorem 2. *Let $M = (S, s_0, \Gamma)$ be a multithreaded program. For every execution of a transition $s \xrightarrow{t} s'$ of M, if it is explored by the stateless DPOR, it must be explored by* SDPOR. □

The soundness of SDPOR is guaranteed by Theorem 2. The detailed proof is given in the appendix. This theorem shows that given a multithreaded program, the *set* of states visited by SDPOR is a superset of the states visited by DPOR. This means that SDPOR is a conservative approach.

Note that DPOR may re-explore the *same* state space many times, while SDPOR will, whenever abstract states are found in the hash-table, avoid all those re-visits. Therefore, the *bag* of DPOR visited states usually has size far higher than the *bag* of states that SDPOR visits. This is the reason that SDPOR can be more efficient than DPOR in checking multithreaded programs. The experiments to be shown in Section 6 confirm that comparing with DPOR, SDPOR is more efficient in checking realistic multithreaded programs.

```
1: Initially: S.push(s_0); H is empty; G is empty;

2: SDPOR(S, H, G) {
3:     s ← S.top;
4:     if (s ∈ H) {
5:         let U = {v | ∃t ∈ s.enabled, v is reachable in G from the node t_g};
6:         for each t ∈ U, UPDATEBACKTRACKSETS(S, t);
7:         return;
8:     }
9:     add s into H;
10:    for each t ∈ s.enabled, UPDATEBACKTRACKSETS(S, t);
11:    if (∃ thread τ, ∃t ∈ s.enabled, tid(t) = τ){
12:        s.backtrack ← {τ};
13:        s.done ← ∅;
14:        while (∃h ∈ s.backtrack \ s.done)  {
15:            s.backtrack ← s.backtrack \ {h};
16:            s.done ← s.done ∪ {h};
17:            let t ∈ s.enabled, tid(t) = h, and let s' = next(s, t);
18:            S.push(s');
19:            if ∃s_x ∈ S s.t. s_x --t_x--> s --t--> s', add a directed edge (t_{x_g}, t_g) to G
20:            SDPOR(S, H, G);
21:            S.pop();
22:        }
23:    }
24: }
```

Fig. 5. Stateful dynamic partial order reduction (SDPOR)

```
1: UPDATEBACKTRACKSETS(S, t) {
2:     let T be the sequence of transitions that are executed from the initial state of
       the program, following the sequence of states in S;
3:     let t_d be the latest transition in T that is dependent and may be co-enabled
       with t;
4:     if (t_d ≠ null){
5:     let s_d be the state in S from which t_d is executed;
6:     let E be {q ∈ s_d.enabled | tid(q) = tid(t), or q in T, q happened after t_d
       and is dependent with some transition in T which was executed by tid(t) and
       happened after q }
7:     if (E ≠ ∅)
8:         choose any q in E, add tid(q) to s_d.backtrack;
9:     else
10:        s_d.backtrack ← s_d.backtrack ∪ {tid(q) | q ∈ s_d.enabled};
11:    }
12: }
```

Fig. 6. Updating the backtrack sets for states in the search stack

4.3 Efficient SDPOR

The algorithm of Figure 5 assumes that the model checker is capable of capturing the program states precisely. Here we present the practical algorithm which combines SDPOR of Figure 5 with the light-weight state capturing scheme that is presented in Section 3. Figure 7 shows the algorithm. Here the procedure SDPOR takes four parameters – although the parameter S is still the search stack, the element of the stack is a pair (s, s_a) such that $s \in S$, $s_a \in S_a$, and s_a is the abstract state of s. The parameter L is the set of local state hash tables. The parameter H, G are the same as in Figure 5.

```
1: Initially: S.push(s_0); H is empty; ∀L_τ ∈ L : L_τ is empty; G is empty;

2: SDPOR(S, H, L, G) {
3:    ⟨s, s_a⟩ ← S.top;
4:    if (s_a ∈ H) {
5:       let U = {v | ∃t ∈ s.enabled, v is reachable in G from the node t_g};
6:       for each t ∈ U, UPDATEBACKTRACKSETS(S, t);
7:       return;
8:    }
9:    add s_a into H;
10:   for each t ∈ s.enabled, UPDATEBACKTRACKSETS(S, t);
11:   if (∃ thread τ, ∃t ∈ s.enabled, tid(t) = τ){
12:      s.backtrack ← {τ};
13:      s.done ← ∅;
14:      while (∃h ∈ s.backtrack \ s.done)  {
15:         s.backtrack ← s.backtrack \ {h};
16:         s.done ← s.done ∪ {h};
17:         let t ∈ s.enabled, tid(t) = h, and let s' = next(s, t);
18:         let δ_h = l_h(s') \ l_h(s), and x = NEXTLOCAL(L_h, lid_h(s_a), δ_h);
19:         let s'_a ∈ S_a s.t. g(s'_a) = g(s') ∧ ls(s'_a) = ls(s_a)[τ := x] ∧ s'_a.PCs = s'.PCs;
20:         S.push(⟨s', s'_a⟩);
21:         if ∃s_x ∈ S s.t. s_x --t_x--> s --t--> s', add a directed edge (t_{x_g}, t_g) to G
22:         SDPOR(S, H, L, G);
23:         S.pop();
24:      }
25:   }
26: }
```

Fig. 7. The combination of SDPOR shown in Figure 5 with the light-weight state capturing scheme which is presented in Section 3.

Comparing with SDPOR in Figure 5, in this combined algorithm, line 18-19 are the new statements for computing the abstract states, line 3 and line 20 are modified to adapt the changes of the search stack, and line 22 is changed to adapt the local state hash tables. The rest of the algorithm is the same as in Figure 5.

5 Implementation

We implemented the algorithm of Figure 7 in the infrastructure of the runtime model checker `Inspect` [7,11] . `Inspect` can instrument a multithreaded C program with code to intercept the visible operations, compile the the instrumented program along with a stub library into an executable, and uses a centralized monitor to systematically explore interleavings of the program by concretely executing the program.

`Inspect` uses escape analysis [12] to reveal potential visible operations in a multithreaded program. Building upon this approach, we implemented an intra-procedural forward data-flow analysis to determine the local state changes between successive visible operations. For any transition t, we treat δ_t as $\delta_\perp$ when: (i) t_g of t is the first visible operation in the procedure, or (ii) there are function calls or updates of pointers between the previous visible operation and t_g. Otherwise, we compute δ_t by capturing the changes of the local variables.

In [11], we described how automated instrumentation is done for stateless runtime checking. To capture the local state changes of threads, we instrument extra code into the program under test to inform the scheduler the local state changes.

6 Experimental Results

We performed experiments on a set of multithreaded benchmarks: `example1` is the program shown in Figure 1, `sharedArray` is a benchmark from [13]. It has two threads that iteratively write to different elements of a shared array. `bbuf` is an implementation of a bounded buffer with concurrent producers and consumers. `bzip2smp` [14] and `pfscan` [15] are two real multithreaded applications. `bzip2smp` is a multithreaded compression program that uses multiple threads to speed up the compression of a file. `pfscan` is a multithreaded file scanner that uses multiple threads to search in parallel through directories. `bzip2smp` contains 6.4k lines of C code, and `pfscan` has 1k lines of C code.

Table 1 shows the experimental results using stateless DPOR and our stateful approach. All the experiments were performed on a PC with an Intel quad-core CPU of 2.4GHz and 2GB of memory. We use "-" to denote that the program cannot be completely checked within 24 hours (86400 seconds).

We compared SDPOR with DPOR on the number of executions (or runs) they require to check a program, the number of transitions explored, and the checking time. Note that for SDPOR, the number of transitions being explored minus the number of "re-visited" states is the number of states encountered in the search. From the experimental results, it is clear that our stateful DPOR approach is more effective than the stateless DPOR, in reducing both the number of transitions to be explored and the checking time.

7 Related Work

There has been substantial work on stateful model checking. Model checkers such as SPIN [16] and Bogor [17] have been very successful in revealing bugs

Table 1. Experimental results on the comparison between DPOR and SDPOR

Benchmarks	Threads	DPOR			SDPOR			
		runs	transitions	time(s)	runs	transitions	re-visited	time(s)
`example1`	2	-	-	-	35	2,084	12	1.41
`sharedArray`	2	-	-	-	98	18,557	33	5.70
`bbuf`	4	47,096	1,058,962	938.27	16,246	349,717	669	344.88
`bzip2smp`	4	-	-	-	4,598	26,442	4460	1311.15
`bzip2smp`	5	-	-	-	18,709	92,276	18,278	9456.34
`bzip2smp`	6	-	-	-	51,400	236,863	50,401	25659.38
`pfscan`	3	84	1,157	0.527	71	967	2	0.485
`pfscan`	4	13,617	189,218	240.74	3,168	40,395	334	57.43
`pfscan`	5	-	-	-	272,873	3,402,486	39,008	5328.84

and proving the correctness of systems. However, it is difficult for classic model checkers to check realistic multithreaded programs, which often heavily use library routines and have sophisticated memory manipulation operations. The advantage of our approach is able to directly examine the programs and avoid the modeling overhead (and potential consistency issues).

Musuvathi et al. [5] developed CMC, which is a runtime model checker that can precisely capture the states of a concurrent program by snapshoting the kernel space plus the user space of the program. In our work, we do not capture the whole state of a multithreaded program. Instead, we abstract the local states of threads as identities, and try to recognize the same states in different executions by tracking the local state changes. Compared with CMC, our approach is more light-weight in capturing states at runtime.

Gueta et al. [13] proposed Cartesian partial order reduction, which reduces the search space by delaying unnecessary context switches using Cartesian vectors. Cartesian partial order reduction performs stateful search, and can deal with cyclic state space. However, their approach assumed that the model checker is capable of capturing the states precisely, and did not address the problem of practical state capturing at runtime. We present a light-weight method for capturing the states of concurrent programs at runtime, and show how to adapt the stateful search into dynamic partial order reduction.

Yi et al. [18] proposed another stateful dynamic partial order reduction method based on the summary of interleavings. [18] also assumed that the model checker is able to precisely capture the states, and did not address the problem of state capturing at runtime. Their definition of *summary* for interleavings is a set of happen-before transition mappings. In their method, each state is associated with a summary of interleaving information, which could be very expensive to store and to keep updated. When a visited state is encountered, our SDPOR computes a summary for the states that can be reached from the visited state in one or more steps. Different from their work, we use a visible operation dependency graph to dynamically compute the summary when a visited state is encountered. As a result, in our approach, the state summary computation is more light-weight.

8 Conclusion

We present an efficient stateful runtime model checking approach to testing multithreaded C programs. To overcome the problem of capturing local states of multithreaded C programs at runtime, we propose a novel light-weight state abstraction scheme to conservatively capture local states. We also propose a stateful dynamic partial order reduction algorithm, and show how to combine it with our light-weight state capturing scheme. Compared with the traditional stateless DPOR approach, our approach is able to detect commutativity of transitions in different executions of multithreaded programs at runtime, and avoid exploring redundant interleavings. The experiments show that our approach is more efficient than stateless DPOR in checking realistic programs.

References

1. Lee, E.A.: The problem with threads. Volume 39., Los Alamitos, CA, USA, IEEE Computer Society Press (2006) 33–42
2. Godefroid, P.: Model Checking for Programming Languages using Verisoft. In: POPL. (1997) 174–186
3. Musuvathi, M., Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs. In Ferrante, J., McKinley, K.S., eds.: PLDI, ACM (2007) 446–455
4. 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, ACM (2005) 110–121
5. Musuvathi, M., Park, D.Y.W., Chou, A., Engler, D.R., Dill, D.L.: CMC: A Pragmatic Approach to Model Checking Real Code. In: OSDI. (2002)
6. Visser, W., Havelund, K., Brat, G.P., Park, S.: Model checking programs. In: ASE. (2000) 3–12
7. http://www.cs.utah.edu/∼yuyang/inspect
8. Pratikakis, P., Foster, J.S., Hicks, M.: Locksmith: context-sensitive correlation analysis for race detection. In: PLDI, ACM Press (2006) 320–331
9. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem. Springer-Verlag (1996)
10. Palmer, R.L.: Formal Analysis for MPI-based High Performance Computing Software, Ph.D. Dissertation, University of Utah. (2007)
11. http://www.cs.utah.edu/∼yuyang/inspect/inspect-intro.pdf
12. Salcianu, A., Rinard, M.: Pointer and escape analysis for multithreaded programs. In: PPoPP, New York, NY, USA, ACM Press (2001) 12–23
13. Gueta, G., Flanagan, C., Yahav, E., Sagiv, M.: Cartesian partial-order reduction. In Bosnacki, D., Edelkamp, S., eds.: SPIN. Volume 4595 of Lecture Notes in Computer Science., Springer (2007) 95–112
14. http://bzip2smp.sourceforge.net/
15. http://freshmeat.net/projects/pfscan
16. Holzmann, G.J.: The Spin Model Checker: Primer and Reference Manual. Addison-Wesley (2004)
17. Robby, Dwyer, M.B., Hatcliff, J.: Bogor: an extensible and highly-modular software model checking framework. In: ESEC / SIGSOFT FSE. (2003) 267–276

18. Yi, X., Wang, J., Yang, X.: Stateful Dynamic Partial-Order Reduction. In Liu, Z., He, J., eds.: ICFEM. Volume 4260 of Lecture Notes in Computer Science., Springer (2006) 149–167

Appendix

Theorem 1. *Let $M = (S, s_0, \Gamma)$ be a multithreaded program. In a depth first search on S following the algorithm of Figure 3, let $s, s' \in S$ be states that can be reached from s_0, and let $s_a, s'_a \in S_a$ be the abstract states of s and s'. Then $\forall \tau \in Tid : lid_\tau(s_a) = lid_\tau(s'_a) \implies l_\tau(s) = l_\tau(s')$.*

Proof. In a depth first search on S following the algorithm of Figure 3, let n be the number of times that NextLocal returns to Dfs from line 4 or line 5 of Figure 4. That is, n is the number of times that NextLocal is invoked with either $\delta_\tau = \delta_\varepsilon$ or $\exists y.\langle (i, \delta_\tau) \to y\rangle \in L_\tau$. We now prove the theorem by induction on n.

- (Base case) $n = 0$: Following NextLocal, if $n = 0$, all calls to NextLocal must return from either line 3 or line 7. That is, NextLocal has never been invoked with $\delta_\tau = \delta_\varepsilon$ or $\exists y.\langle (i, \delta_\tau) \to y\rangle \in L_\tau$. Hence, following the algorithm of Figure 4, every local state that has been visited must be assigned a unique id. Therefore, in this situation, $lid_\tau(s_a) = lid_\tau(s'_a)$ holds if and only if $s_a = s'_a$. Obviously, $lid_\tau(s_a) = lid_\tau(s'_a) \implies l_\tau(s) = l_\tau(s')$ holds in this situation.
- (Induction hypothesis) Let $k \geq 0$. For all n, $n \leq k$, Theorem 1 holds.
- (Induction step) Let $n = k + 1$. Let $s' \in S$ be the state and $\tau \in Tid$ be the thread such that by invoking NextLocal$(L_\tau, lid_\tau(s'), \delta_\tau)$ at line 8, n changes from k to $k + 1$. Consider the situation that Dfs has finished executing line 8 of Figure 3, but has not started executing line 9. Let $s \in S$ be a state and t be a transition such that $s \xrightarrow{t} s'$ and $\tau = tid(t)$. There are two cases with respect to s':
  - If $\delta_\tau = \delta_\varepsilon$, according to line 7 of Dfs, we have $l_\tau(s) = l_\tau(s')$, and $lid_\tau(s_a) = lid_\tau(s'_a)$. Obviously $lid_\tau(s_a) = lid_\tau(s'_a) \implies l_\tau(s) = l_\tau(s')$ holds. Hence the theorem holds.
  - If $\exists y.\langle (lid_\tau(s_a), \delta_\tau) \to y\rangle \in L_\tau$: Let $s_1, s_2 \in S$ be the two states that have been visited and t_1 be the transition such that $\langle (lid_\tau(s_a), \delta_\tau) \to y\rangle$ was added to L_τ when Dfs explored $s_1 \xrightarrow{t_1} s_2$. Let s_{1_a} and s_{2_a} respectively be the abstract state of s_1 and s_2 Obviously we have $lid_\tau(s_{1_a}) = lid_\tau(s_a)$, $lid_\tau(s_{2_a}) = y$, and $l_\tau(s_2) \setminus l_\tau(s_1) = l_\tau(s') \setminus l_\tau(s)$. According to the induction hypothesis, $l_\tau(s_1) = l_\tau(s)$ must hold. As $l_\tau(s_2) \setminus l_\tau(s_1) = l_\tau(s') \setminus l_\tau(s)$, we have $l_\tau(s_2) = l_\tau(s')$. Hence, the theorem holds. Otherwise, it contradicts the induction hypothesis.

□

Let $M = (S, s_0, \Gamma)$ be a multithreaded program. Let s be a state in S. We use R_s to denote the set of states that are reachable from s by executing one or more transitions. Obviously we have $R_s \subseteq S$.

Let SdporK$_k$ be the algorithm of Figure 8. Comparing with Sdpor, the only difference between SdporK$_k$ and Sdpor is that SdporK$_k$ takes one more parameter k, which bounds SdporK$_k$ to return only at the first k visited states. In more detail, SdporK$_k$ uses a global counter c to record the number of visited states that it has encountered during the depth-first search (line 5 of Figure 8). When a visited state s_v is encountered, if

```
1: Initially: c = 0; S.push(s_0); H is empty;

2: SDPOR_k(S, H) {
3:     s ← S.top;
4:     if (s ∈ H) {
5:       c ← c + 1;
6:       if (c ≤ k) {
7:           let T_p = {t_g | t can be executed from states which are reachable from s };
8:           for each t ∈ T_p, UPDATEBACKTRACKSETS(S, t);
9:           return;
10:      }
11:    }
12:    add s into H;
13:    for each t ∈ s.enabled, UPDATEBACKTRACKSETS(S, t);
14:    if (∃ thread τ, ∃t ∈ s.enabled, tid(t) = τ){
15:      s.backtrack ← {τ};
16:      s.done ← ∅;
17:      while (∃h ∈ s.backtrack \ s.done)  {
18:        s.backtrack ← s.backtrack \ {h}, s.done ← s.done ∪ {h};
19:        let t ∈ s.enabled, tid(t) = h, and let s' = next(s, t);
20:        S.push(s');
21:        SDPOR_k(S, H);
22:        S.pop();
23:      }
24:    }
25: }
```

Fig. 8. SDPOR$_k$ only stops depth-first search and backtrack immediately at the first k revisited states.

$c \leq k$, then SDPOR$_k$ updates the backtrack sets of states in the search stack (line 7-8 of Figure 8) and returns immediately; otherwise, SDPOR$_k$ continues exploring R_{s_v}.

We have a class of algorithms {SDPOR$_0$, SDPOR$_1$, ... } by assigning k specific values. Let $\mathcal{A}$ denote an algorithm that explores the state space of M. Let $S_{\mathcal{A}} \subseteq S$ refer to the set of states that is explored using $\mathcal{A}$ by starting from s_0. Obviously, we have $S_{\text{DPOR}} = S_{\text{SDPOR}_0}$ and $S_{\text{SDPOR}} = S_{\text{SDPOR}_\infty}$.

To prove the correctness of Theorem 2, we first prove Lemma 1, which characterize the relationship between SDPOR$_k$ and SDPOR$_{k+1}$.

Lemma 1. *Let $M = (S, s_0, \Gamma)$ be a multithreaded program. Let $s, s' \in S$ and t be a transition of M such that $s \xrightarrow{t} s'$. Let $k \geq 0$. If $s \xrightarrow{t} s'$ is explored by* SDPOR$_k$*, it must be explored by* SDPOR$_{k+1}$.

Proof. Let r be the value of the global variable c when we finish checking the state space of M using SDPOR$_k$. There are two cases with respect to r:

- If $r \leq k$, obviously that the state spaces traversed by SDPOR$_k$ and SDPOR$_{k+1}$ are identical. The lemma holds.
- If $r > k$, let v_i be the i-th visited state SDPOR$_k$ and SDPOR$_{k+1}$ encounter while exploring the state space of M. Let $\Gamma_i^x \subseteq \Gamma$ be the transition relation that has

been explored by SDPOR_x before reaching the i-th visited states. It is obvious that $\forall i, i \leq k+1 : \Gamma_i^k = \Gamma_i^{k+1}$ holds.
Now we consider the exploration of the search space by SDPOR_k and SDPOR_{k+1} after they encounter v_{k+1}. Following the algorithm of Figure 8, while encountering v_{k+1}, SDPOR_k is equivalent to a stateless search that does not record the search history, and explores $R_{v_{k+1}}$. However, SDPOR_{k+1} does not explore $R_{v_{k+1}}$. Before encountering v_{k+1}, the state spaces explored by SDPOR_k and SDPOR_{k+1} are identical. Hence, the search stacks of SDPOR_k and SDPOR_{k+1} are identical at the point of encountering v_{k+1}. Let s^k and s^{k+1} denote the correspondent states that are respectively in the search stacks of SDPOR_k and SDPOR_{k+1}. To prove the lemma, all that we need to prove is that, when SDPOR_k and SDPOR_{k+1} backtrack from $s_{v_{k+1}}$, for all pairs of states $\langle s^k, s^{k+1} \rangle$, we have $s_{sdpor_k}.backtrack \subseteq s_{sdpor_{k+1}}.backtrack$. This can be proved by contradiction. Suppose while backtracking from v_{k+1}, there exists $\langle s^k, s^{k+1} \rangle$ such that $s^k.backtrack \supset s^{k+1}.backtrack$. This implies that $\exists h \in Tid : h \in s^k.backtrack \wedge h \notin s^{k+1}.backtrack$. As the only difference between SDPOR_k and SDPOR_{k+1} is that SDPOR_k explores $R_{v_{k+1}}$ while SDPOR_{k+1} does not, this can happen if and only if $\exists s_1, s_2 \in R_{v_{k+1}}, \exists t \in s^{k+1} : s_1 \xrightarrow{t_1} s_2$ and t_1 is dependent with t. However, following the algorithm of Figure 8, if the execution of t_1 happens before backtracking v_{k+1} in SDPOR_k, $t_1 \in T_p$. Hence, if $h \in s^k.backtrack$, h must be in $s^{k+1}.backtrack$. This contradicts $h \notin s^{k+1}.backtrack$.

□

Let $M = (S, s_0, \Gamma)$ be a multithreaded program. Let G be the transition dependency graph of M. G is dynamically constructed following the algorithm of Figure 5. Let $\mathcal{U}$ be the set of visible operations that is computed at line 5 of Figure 5. Let T_p be the set of visible operations that is computed as line 7 of Figure 8. We have the following lemma:

Lemma 2. $T_p \subseteq \mathcal{U}$.

Proof: Following the algorithm of Figure 5, it is clear that while backtracking from a state s, all transition dependency edges that can appear in R_s must have been added to G. Therefor, we have $\forall t \in T_p : t \in \mathcal{U}$. □

Theorem 2. *Let $M = (S, s_0, \Gamma)$ be a multithreaded program. For every execution of a transition $s \xrightarrow{t} s'$ of M, if it is explored by the stateless DPOR, it must be explored by* SDPOR.

Proof. With Lemma 1, Lemma 2, $S_{\text{DPOR}} = S_{\text{SDPOR}_0}$ and $S_{\text{SDPOR}} = S_{\text{SDPOR}_\infty}$, it is clear that the theorem holds. □