Dynamic Model Checking with Property Driven Pruning to Detect Race Conditions *

Chao Wang[1], Yu Yang[2], Aarti Gupta[1], and Ganesh Gopalakrishnan[2]

[1] NEC Laboratories America, Princeton, New Jersey, USA
[2] School of Computing, University of Utah, Salt Lake City, Utah, USA

Abstract. We present a new property driven pruning algorithm in dynamic model checking to efficiently detect race conditions in multithreaded programs. The main idea is to use a lockset based analysis of observed executions to help prune the search space to be explored by the dynamic search. We assume that a stateless search algorithm is used to systematically execute the program in a depth-first search order. If our conservative lockset analysis shows that a search subspace is race-free, it can be pruned away by avoiding backtracks to certain states in the depth-first search. The new dynamic race detection algorithm is both sound and complete (as precise as the dynamic partial order reduction algorithm by Flanagan and Godefroid). The algorithm is also more efficient in practice, allowing it to scale much better to real-world multithreaded C programs.

1 Introduction

Concurrent programs are notoriously hard to debug because of their often large number of possible interleavings of thread executions. Concurrency bugs often arise in rare situations that are hard to anticipate and handle by standard testing techniques. One representative type of bugs in concurrent programs is a data race, which happens when multiple threads access a shared data variable simultaneously and at least one of the accesses is a write. Race conditions were among the flaws in the Therac-25 radiation therapy machine [12], which led to the death of three patients and injuries to several more. A race condition in the energy management system of some power facilities prevented alerts from being raised to the monitoring technicians, eventually leading to the 2003 North American Blackout.

To completely verify a multithreaded program for a given test input, one has to inspect all possible thread interleavings. For deterministic threads, the only source of nondeterminism in their execution comes from the thread scheduler of the operating system. In a typical testing environment, the user does not have full control over the scheduling of threads; running the same test multiple times does not necessarily translate into a better interleaving coverage. Static analysis has been used for detecting data races in multithreaded programs, both for a given test input [20, 16] and for all possible inputs [6, 4, 17, 11, 22]. However, a race condition reported by static analysis may be bogus (there can be many false alarms); even if it is real, there is often little information for the user to reproduce the race. Model checking [3, 18] has the advantage of exhaustive coverage which means all possible thread interleavings will be explored. However,

* Yu Yang and Ganesh Gopalakrishnan were supported in part by NSF award CNS-0509379 and the Microsoft HPC Institutes program.

model checkers require building finite-state or pushdown automata models of the software [10, 1]; they often do not perform well in the presence of lock pointers and other heap allocated data structures.

Dynamic model checking as in [9, 5, 14, 23, 24] can directly check programs written in full-fledged programming languages such as C and Java. For detecting data races, these methods are sound (no bogus race) due to their concrete execution of the program itself as opposed to a model. While a bounded analysis is used in [14], the other methods [9, 5, 23, 24] are complete for terminating programs (do not miss real races) by systematically exploring the state space without explicitly storing the intermediate states. Although such dynamic software model checking is both sound and complete, the search is often inefficient due to the astronomically large number of thread interleavings and the lack of property specific pruning. Dynamic partial order reduction (DPOR) techniques [5, 23, 7] have been used in this context to remove the redundant interleavings from each equivalence class, provided that the representative interleaving has been explored. However, the pruning techniques used by these DPOR tools have been generic, rather than *property-specific*.

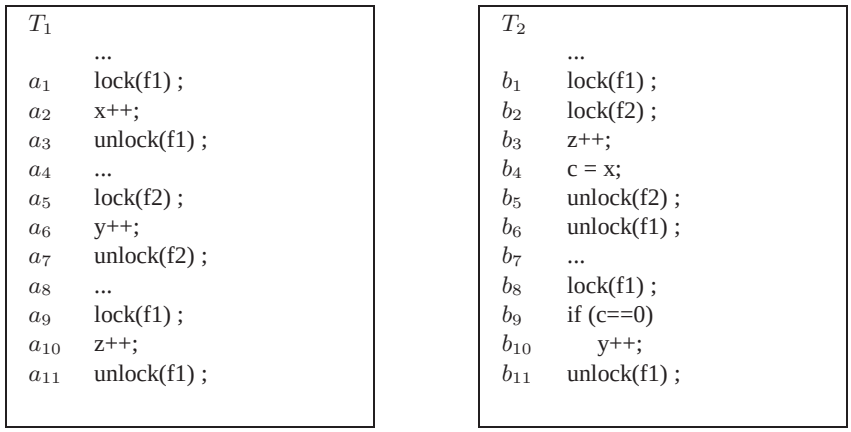

Fig. 1. Race condition on accessing variable y (assume that $x = y = 0$ initially)

Without a conservative or warranty type of analysis tailored toward the property to be checked, model checking has to enumerate all the equivalence classes of interleavings. Our observation is that, as far as race detection is concerned, many equivalence classes themselves may be redundant. Fig. 1 shows a motivating example, in which two threads use locks to protect accesses to shared variables x, y, and z. A race condition between a_6 and b_{10} may occur when b_4 is executed before a_2, by setting c to 0. Let the first execution sequence be $a_1 \ldots a_{11} b_1 \ldots b_9 b_{11}$. According to the DPOR algorithm by Flanagan and Godefroid [5], since a_{10} and b_3 have a read-write conflict, we need to backtrack to a_8 and continue the search from $a_1 \ldots a_8 b_1$. As a generic pruning technique, this is reasonable since the two executions are not Mazurkiewicz-trace equivalent [13]. For data race detection, however, it is futile to search any of these execution traces in which a_6 and b_{10} cannot be simultaneously reachable (which can be

revealed by a conservative lockset analysis). We provide a property-specific pruning algorithm to skip such redundant interleavings and backtrack directly to a_1.

In this paper, we propose a trace-based dynamic lockset analysis to prune the search space in the context of dynamic model checking. Our main contributions are: (1) a new lockset analysis of the observed execution trace for checking whether the associated search subspace is race-free. (2) property driven pruning in a backtracking algorithm using depth-first search.

We analyze the various alternatives of the current execution trace to anticipate race conditions in the corresponding search space. Our trace-based lockset analysis relies on both information derived from the dynamic execution and information collected statically from the program; therefore, it is more precise than the purely static lockset analysis conducted *a priori* on the program [4, 6, 17, 11, 22]. Our method is also different from the *Eraser*-style dynamic lockset algorithms [20, 16], since our method decides whether the entire search subspace related to the concrete execution generated is race-free, not merely the execution itself. The crucial requirement for a method to be used in our framework for pruning of the search space is completeness—pruning must not remove real races. Therefore, neither the aforementioned dynamic lockset analysis nor the various predictive testing techniques [21, 2] based on happens-before causality (sound but incomplete) can be be used in this framework. CHESS [14] can detect races that may show up within a preemption bound; it exploits the preemption bounding for pruning, but does not exploit the lock semantics to effect reduction.

In our approach, if the search subspace is found to be race-free, we prune it away during the search by avoiding backtracks to the corresponding states. Recall that essentially the search is conducted in a DFS order. If there is a potential race, we analyze the cause in order to compute a proper backtracking point. Our backtracking algorithm shares the same insights as the DPOR algorithm [5], with the additional pruning capability provided by the trace-based lockset analysis. Note that DPOR relies solely on the *independence relation* to prune redundant interleavings (if t_1, t_2 are independent, there is no need to flip their execution order). In our algorithm, *even if* t_1, t_2 *are dependent*, we may skip the corresponding search space if flipping the order of t_1, t_2 does not affect the reachability of any race condition. If there is no data race at all in the program, our algorithm can obtain the desired race-freedom assurance much faster.

2 Preliminaries

2.1 Concurrent Programs

We consider a concurrent program with a finite number of threads as a state transition system. Let $Tid = \{1, \ldots, n\}$ be a set of thread indices. Threads may access local variables in their own stacks, as well as global variables in a shared heap. The operations on global variables are called *visible* operations, while those on thread local variables are called *invisible* operations. We use $Global$ to denote the set of states of all global variables, $Local$ to denote the set of local states of a thread. PC is the set of values of the program counter of a thread. The entire system state ($\mathcal{S}$), the program counters of the threads (PCs), and the local states of threads ($Locals$) are defined as follows:

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

A transition $t : \mathcal{S} \rightarrow \mathcal{S}$ advances the program from one state to a subsequent state. Following the notation of [5, 23], each transition t consists of one visible operation, followed by a finite sequence of invisible operations of the same thread up to (but excluding) the next visible operation. We use $tid(t) \in Tid$ to denote the thread index of the transition t. Let $\mathcal{T}$ be the set of all transitions of a program. A transition $t \in \mathcal{T}$ is *enabled* in a state s if the next state $t(s)$ is defined. We use $s \xrightarrow{t} s'$ to denote that t is enabled in s and $s' = t(s)$. Two transitions t_1, t_2 *may be co-enabled* if there exists a state in which both t_1 and t_2 are enabled. The state transition graph is denoted $\langle \mathcal{S}, s_0, \Gamma \rangle$, where $s_0 \in \mathcal{S}$ is the unique initial state and $\Gamma \subseteq \mathcal{S} \times \mathcal{S}$ is the transition relation: $(s, s') \in \Gamma$ iff $\exists t \in \mathcal{T} : s \xrightarrow{t} s'$. An execution sequence is a sequence of states $s_0, \ldots, s_n$ such that $\exists t_i \,.\, s_{i-1} \xrightarrow{t_i} s_i$ for all $1 \leq i \leq n$.

Two transitions are *independent* if and only if they can neither disable nor enable each other, and swapping their order of execution does not change the combined effect. Two execution trace are equivalent iff they can be transformed into each other by repeatedly swapping adjacent independent transitions. In model checking, partial order reduction (POR [8]) has been used to exploit the redundancy of executions from the same equivalence class to prune the search space; in particular, model checking has to consider only one representative from each equivalence class.

2.2 Dynamic Partial Order Reduction

Model checking of a multithreaded program can be conducted in a stateless fashion by systematically executing the program in a depth-first search order. This can be implemented by using a special *scheduler* to control the execution of visible operations of all threads; the scheduler needs to give permission to, and observe the result of every visible operation of the program. Instead of enumerating the reachable states, as in classic model checkers, it exhaustively explores all the feasible thread interleavings. Fig. 2 shows a typical stateless search algorithm. The scheduler maintains a *search stack* S of states. Each state $s \in S$ is associated with a set $s.enabled$ of enabled transitions, a set $s.done$ of executed transitions, and a *backtracking set*, consisting of the thread indices of some enabled transitions in s that need to be explored from s. In this context, backtracking is implemented by re-starting the program afresh under a different thread schedule [23], while ensuring that the replay is deterministic—i.e. all external behaviors (e.g., mallocs and IO) are also assumed to be replayable [1].

The procedure DPORUPDATEBACKTRACKSETS(S, t) implements the dynamic partial order reduction algorithm of [5]. It updates the backtrack set only for the last transition t_d in T such that t_d is dependent and may be co-enabled with t (line 19). The set $s_d.backtrack$ is also a subset of the enabled transitions, and the set E consists of

[1] While malloc replayability is ensured by allocating objects in the same fashion, IO replayability is ensured by creating suitable closed environments.

```
1:  Initially: S is empty; DporSearch(S, s_0)

2:  DporSearch(S, s) {
3:      if (DetectRace(s)) exit (S);
4:      S.push(s);
5:      for each t ∈ s.enabled, DporUpdateBacktrackSets(S, t);
6:      let τ ∈ Tid such that ∃t ∈ s.enabled : tid(t) = τ;
7:      s.backtrack ← {τ};
8:      s.done ← ∅;
9:      while (∃t: tid(t) ∈ s.backtrack and t ∉ s.done) {
10:         s.done ← s.done ∪ {t};
11:         s.backtrack ← s.backtrack \ {tid(t)};
12:         let s' ∈ S such that s -t-> s';
13:         DporSearch(S, s');
14:         S.pop(s);
15:     }
16: }
17: DporUpdateBacktrackSets(S, t) {
18:     let T = {t_1, ..., t_n} be the sequence of transitions associated with S;
19:     let t_d be the latest transition in T that is dependent and may be co-enabled with t;
20:     if (t_d ≠ null){
21:         let s_d be the state in S from which t_d is executed;
22:         let E be {q ∈ s_d.enabled | either tid(q) = tid(t), or q was executed after t_d in T and
            a happens-before relation exists for (q, t)}
23:         if (E ≠ ∅)
24:             choose any q in E, add tid(q) to s_d.backtrack;
25:         else
26:             s_d.backtrack ← s_d.backtrack ∪ {tid(q) | q ∈ s_d.enabled};
27:     }
28: }
```

Fig. 2. Stateless search with dynamic partial order reduction (c.f. [5])

transitions q in T such that (q, t) has a *happens-before* relation (line 22). Intuitively, q happens-before t means that flipping the execution order of q and t may lead to interleavings in a different equivalence class. For a better understanding, a plain depth-first search, with no partial order reduction at all, would correspond to an alternative implementation of line 19 in which t_d is defined as the last transition in T such that $tid(t_d) \neq tid(t)$, regardless of whether t_d and t are dependent, and an alternative implementation of line 22 in which $E = \emptyset$.

Data race detection is essentially checking the simultaneous reachability of two conflicting transitions. The procedure DetectRace(s) used in line 3 of Fig. 2 checks in each state s whether there exist two transitions t_1, t_2 such that (1) they access the same shared variable; (2) at least one of them is a write; and (3) both transitions are enabled in s. If all three conditions hold, it reports a data race; in this case, the sequence of states $s_0, s_1, \ldots, s$ currently in the stack S serve as a counterexample. The advantage of this race detection procedure is that it does not report bogus races (of course, the race itself may be benign; detecting whether races are malicious is outside the scope of our approach, as well as most other approaches in this area). If the top-level

DPORSEARCH(S, s_0) completes without finding any race, then the program is proved to be race-free under the given input. As pointed out in [5], DPOR is sound and complete for detecting data races (as well as deadlocks and assertion violations), although there is no property driven pruning employed in [5].

3 Race-Free Search Subspace

Given an execution sequence $s_0, \ldots, s_i, \ldots, s_n$ stored in the stack S and a state s_i ($0 \leq i \leq n$), we check (conservatively) whether the search space starting from s_i is race-free. This search subspace consists of all the execution traces sharing the same prefix $s_0, \ldots, s_i$. During dynamic model checking, instead of backtracking for each conflicting transition pair as in DPOR, we backtrack to state s_i only if the corresponding search subspace has potential races.

3.1 Set of Locksets

Let $T = \{t_1, \ldots, t_n\}$ be a transition sequence such that $s_0 \xrightarrow{t_1} s_1 \ldots \xrightarrow{t_n} s_n$. First, we project T to each thread as a sequence $T_\tau = \{t_{\tau_1}, \ldots, t_{\tau_k}\}$ of thread-local transitions; that is, $\forall t \in T_\tau : tid(t) = \tau$. For the example in Fig. 1, T is projected to $T_1 = \{a_1, \ldots, a_{11}\}$ and $T_2 = \{b_1, \ldots, b_9, b_{11}\}$. Next, we partition each thread-local sequence T_τ into smaller segments. In the extreme case, each segment would consist of a single transition. For each segment $seg_i \subseteq T_\tau$, we identify the global variables that may be accessed within seg_i; for each access, we also identify the corresponding *lockset*—the set of locks held by thread τ when the access happens.

Definition 1. *For each segment seg_i and global variable x, the set $lsSet_x(seg_i)$ consists of all the possible locksets that may be held when x is accessed in seg_i.*

By conservatively assuming that transitions of different threads can be interleaved arbitrarily, we check whether it is possible to encounter a race condition. Specifically, for each global variable x, and for each pair (seg_i, seg_j) of transition segments from different threads, we check whether $\exists set_1 \in lsSet_x(seg_i), set_2 \in lsSet_x(seg_j)$ such that $set_1 \cap set_2 = \emptyset$. An empty set represents a potential race condition—x is not protected by a common lock. The result of this analysis can be refined by further partitioning seg_i, seg_j into smaller fragments. To check whether the search space starting from s_i is race-free, we will conservatively assume that $t_{i+1}, \ldots, t_n$ (transitions executed after s_i in T) may interleave arbitrarily, subject only to the program orders.

Note first, that the lockset analysis is thread-local, i.e., the analysis is performed on a single thread at a time. Second, a precise computation of $lsSet_x(seg_i)$ as in Definition 1 requires the inspection of all feasible execution traces (exponentially many) in which x is accessed in seg_i; we do not perform this precise computation. For conservatively checking the race-free subspace property, it suffices to consider a set of locksets S such that any constituent lockset of S is a subset of the actually held locks. For instance, the coarsest approximation is $lsSet_x(seg_i) = \{\emptyset\}$; that is, x is not protected at all. Under this coarsest approximation for seg_i, if another thread also accesses x in seg_j, our algorithm will report a potential race condition between seg_i and seg_j.

Consider again the example in Fig. 1, let the first execution trace be partitioned into

$$seg_1 = a_1, \ldots, a_8 \qquad seg_3 = b_1, \ldots, b_7$$
$$seg_2 = a_9, \ldots, a_{11} \qquad seg_4 = b_8, \ldots, b_{11}.$$

Since seg_2 shares only z with seg_3, and z is protected by lock $f1$, any execution trace starting from seg_1 is race-free. Therefore, we do not need to backtrack to a_8.

However, the concrete execution itself may not be able to provide enough information to carry out the above analysis. Note that, by definition, $lsSet_x(seg_i)$ must include *all the possible locksets* that may be formed in an interleaving execution of seg_i. In Fig. 1, for instance, although y is accessed in both threads (a_6 and b_{10}), the transition b_{10} does not appear in seg_4 since the `else`-branch was taken. However, $lsSet_y(seg_4)$ is $\{\{f1\}\}$. In general, we need a *may-set* of shared variables that are accessed in seg_i and the corresponding *must-set* of locks protecting each access. We need the information of all the alternative branches in order to compute these sets at runtime.

3.2 Handling the Other Branch

Our solution is to augment all branching statements in the form of `if(c)-else`, through source code instrumentation, so that the information of not-yet-executed branches (computed *a priori*) is readily available to our analysis during runtime. To this end, for both branches of every `if-else` statement, we instrument the program by inserting calls to the following routines ('rec' stands for record):

- `rec-var-access-in-other-branch(`x, L_{acq}, L_{rel}`)` for each access to x; with the set L_{acq} of locks acquired and the set of L_{rel} of locks released before the access.
- `rec-lock-update-in-other-branch(`L_{acq}, L_{rel}`)`; with the set L_{acq} of locks acquired and the set L_{rel} of locks released in the other branch.

The instrumentation is illustrated by a simple example in Fig. 3. In addition to the above routines, we also add recording routines to notify the scheduler about the branch start and end. When the `if`-branch is executed, the scheduler knows that, in the `else`-branch, x is accessed and lock C is acquired before the access (line 4); it also knows that C is the only lock acquired and no lock is released throughout that branch (line 5). Similarly, when the `else`-branch is executed, the scheduler knows that in the `if`-branch, x, y are accessed and lock A is protecting x but not y. According to lines 5 and 16, lock C will be held at the branch merge point because $(L_{acq} \setminus L_{rel}) = \{C\}$. Therefore, our algorithm knows that z is protected by both B and C.

The information passed to these recording routines need to be collected *a priori* by a static analysis of the individual threads (in Section 5). Note that neither the set of shared variables nor any of the corresponding locksets L_{acq}, L_{rel} has to be precise. For a conservative analysis, it suffices to use an over-approximated set of shared variables, a subset $\check{L}_{acq} \subseteq L_{acq}$ of acquired locks, and superset $\hat{L}_{rel} \supseteq L_{rel}$ of released locks. By using $\check{L}_{acq}$ and $\hat{L}_{rel}$, we can compute a must-set $(\check{L}_{acq} \setminus \hat{L}_{rel})$, which is a subset of the actually held locks.

```
1:   lock(B)
2:   if (c) {
3:     rec-branch-begin();                          //added
4:     rec-var-access-in-other-branch(x,{C},{});    //added
5:     rec-lock-update-in-other-branch({C},{});     //added
6:     lock(A);
7:     x++;
8:     unlock(A);
9:     y=5;
10:    lock(C);
11:    rec-branch-end();                            //added
12:  }else {
13:    rec-branch-begin();                          //added
14:    rec-var-access-in-other-branch(x,{A},{ });   //added
15:    rec-var-access-in-other-branch(y,{A},{A});   //added
16:    rec-lock-update-in-other-branch({A,C},{A}); //added
17:    lock(C);
18:    x++;
19:    rec-branch-end();                            //added
20:  }
21:  z++;
22:  unlock(C);
23:  unlock(B)
```

Fig. 3. Instrumenting the branching statements of each thread

3.3 Checking Race-Free Subspace

The algorithm for checking whether a search subspace is race-free is given in Fig. 4. For each transition $t \in T$ and global variable x, we maintain:

- $lsSet(t)$, the *set of locksets* held on one of the paths by t;
- $mayUse(t, x)$ if t is a branch begin, the set of locksets of x in the other branch.

In state s_i, the set ls_τ of locks held by thread τ is known. First, we use COMPUTELOCKSETS to update $lsSet(t)$ and $mayUse(t, x)$ for all variables x accessed and transitions t executed after s_i. Potential race conditions are checked by intersecting pairwise locksets of the same variable in different threads. If any of the intersection in line 11 is empty, SUBSPACERACEFREE returns FALSE.

In Fig. 5, COMPUTELOCKSETS starts with ls_τ, which comes from the concrete execution and hence is precise. T_τ consists of the following types of transitions: (1) instrumented recording routines; (2) lock/unlock; (3) other program statements. The stack $update$ is used for temporary storage. Both $lsSet(t)$ and $mayUse(t, x)$ are *sets of locksets*, of which each constituent lockset corresponds to a distinct unobserved path (a path skipped due to a false branch condition) or variable access. Note that we do not merge locksets from different branches into a single must-lockset, but maintain them as separate entities in $lsSet(t)$ and then propagate to the subsequent transitions in T_τ.

Multiple branches may be embedded in the observed sequence T_τ, as shown in Fig. 6. In the left-hand-side figure, the unobserved branch itself has two branches, each of which needs a recording routine in T_τ to record the lock updates. Inside COMPUTELOCKSETS, lock updates from t_{l_2} are stored temporarily in the stack $update$ and finally used to compute $lsSet(t_i)$ at the merge point. In the right-hand-side figure, the observed branch (from t'_j to t'_i) contains another observed branch (from t_j to t_i). This is why a stack $update$, rather than a set, is needed. Note that t'_{l_2} is executed before t_{l_2}, but $lsSet(t'_i)$ is computed after $lsSet(t_i)$.

```
1:  SUBSPACERACEFREE(s_i) {
2:      let T = {t_1, t_2, ..., t_m} such that s_i -t_1-> s_{i+1} ... -t_m-> s_{m+1} and s_{m+1}.enabled = ∅;
3:      for each (τ ∈ Tid) {
4:          let T_τ = {t_{τ_1}, ..., t_{τ_k}} be a subsequence T_τ ⊆ T such that ∀t ∈ T_τ : tid(t) = τ;
5:          let ls_τ be the set of locks held by thread τ at s_i;
6:          COMPUTELOCKSETS(ls_τ, T_τ);
7:      }
8:      for each (global variable x) {
9:          let t_1, t_2 ∈ T, tid(t_1) ≠ tid(t_2), both may access x, and at least one is a write;
10:         let ls_1 ∈ (lsSet(t_1) ∪ mayUse(t_1, x)), let ls_2 ∈ (lsSet(t_2) ∪ mayUse(t_2, x));
11:         if (∃ls_1, ls_2 such that ls_1 ∩ ls_2 = ∅) return FALSE;
12:     }
13:     return TRUE;
14: }
```

Fig. 4. Checking whether the search subspace from s_i is race-free at run time

```
1:  COMPUTELOCKSETS(ls_τ, T_τ) {
2:      let lsSet(t_0) ={ ls_τ };
3:      let T_τ = {t_1, ..., t_k}; ∀t_i ∈ T_τ, ∀x : lsSet(t_i) ← ∅ and mayUse(t_i, x) ← ∅;
4:      i ← 1;
5:      while (i ≤ k) {
6:          if (t_i is rec-branch-begin)
7:              update.push(∅);
8:          if (t_i is lock(f1))
9:              lsSet(t_i) ← {ls ∪ {f1} | ls ∈ lsSet(t_{i-1});
10:         else if (t_i is unlock(f1))
11:             lsSet(t_i) ← {ls \ {f1} | ls ∈ lsSet(t_{i-1});
12:         else if (t_i is rec-var-access-in-other-branch(x, L_acq, L_rel))
13:             let t_j be the last branch begin that precedes t_i;
14:             mayUse(t_j, x) ← mayUse(t_j, x) ∪ {ls ∪ L_acq \ L_rel | ls ∈ lsSet(t_j)};
15:         else if (t_i is rec-lock-update-in-other-branch(L_acq, L_rel))
16:             let t_j be the last branch begin that precedes t_i;
17:             update.top() ← update.top() ∪ {ls ∪ L_acq \ L_rel | ls ∈ lsSet(t_j)};
18:         else if (t_i is rec-branch-end)
19:             lsSet(t_i) ← update.pop() ∪ lsSet(t_{i-1});
20:         else
21:             lsSet(t_i) ← lsSet(t_{i-1});
22:         i ← i + 1;
23:     }
24: }
```

Fig. 5. Computing locksets that may be held by each transition in T_τ

Let $s_j \xrightarrow{t_{j+1}} s_{j+1}$ be a branch begin and $s_{i-1} \xrightarrow{t_i} s_i$ be the matching branch end. From the pseudo code in Fig. 5, it is clear that the following two theorems hold.

Theorem 1. *$lsSet(t_i)$ contains, for each unobserved path from s_j and to s_i, a must-set of locks held at s_i (if that path were to be executed).*

Theorem 2. *$mayUse(t_i, x)$ contains, for each access of x in an unobserved path from s_j to s_i, a must-set of locks held when accessing x in that path.*

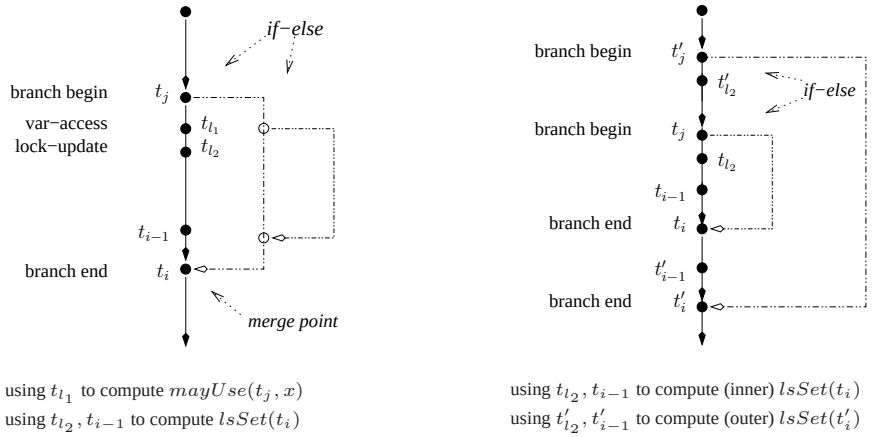

Fig. 6. Multiple branches in an execution trace (observed and unobserved branches)

Although the standard notion of locksets is used in our analysis, the combination of dynamically computed information of the observed execution and statically computed information of not-yet-executed branches differentiates us from the existing dynamic [20, 16] and static [6, 4, 17, 11, 22] lockset algorithms. It differs from the Eraser-style lockset algorithms [20, 16] in that it has to consider not only the current execution but also the not-yet-activated branches. It differs from the purely static lockset analysis [6, 4, 17, 11, 22] in that it utilizes not only the statically computed program information, but also the more precise information derived dynamically from the execution. In particular, our lockset computation starts with a precise lockset ls_τ of the concrete execution (line 5 of Fig. 4). In the presence of pointers to data and locks, a purely static analysis may be imprecise; the actual set of shared variables accessed or locks held during a concrete execution may be significantly smaller than the (conservatively computed) points-to sets of the pointers.

4 The Overall Algorithm

We rely on the conservative lockset analysis to prune the search space, and the concrete program execution to ensure that no bogus race is reported. The overall algorithm is given in Fig. 7. The procedure PDPSEARCH, where PDP stands for Property-Driven Pruning, takes the stack S and a state s as input. Each time PDPSEARCH is called on

a new state s, lines 10-24 will be executed. DETECTRACE(s) is used to detect race conditions in s during runtime (explained in Section 2). If a race condition is found, it terminates with a counterexample in S. When an execution terminates ($s.enabled = \varnothing$ of line 3), we update the backtracking points for the entire trace. This is significantly different from the DPOR algorithm, which updates the backtracking points for each state s when it is pushed into the stack S. Rather than updating the backtracking points in the pre-order of DFS as in DPOR, our algorithm waits until the information pertaining to an entire execution trace is available. In line 27, for each state t_d that is dependent and may be co-enabled with t, we check (in addition to that of DPOR) whether the search subspace from s_d is race-free. If the answer is yes, we can safely skip the backtracking points at s_d. Otherwise, we proceed in the same fashion as DPOR.

The Running Example We show how the overall algorithm works on the example in Fig. 1. Assume that the first execution trace is

$$s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \ldots \xrightarrow{a_6} s_6 \ldots s_9 \xrightarrow{a_{10}} s_{10} \ldots s_{13} \xrightarrow{b_3} s_{14} \xrightarrow{b_4} s_{15} \ldots \xrightarrow{b_9} s_{20} \xrightarrow{b_{11}} s_{21} \ ,$$

produced by lines 11-20 of Fig. 7. Since $s_{21}.enabled = \emptyset$, the call PDPSEARCH($S, s_{21}$) executes lines 3-9. For every $s_b \in S$, we update the backtrack sets; we go through the stack in the following order: $s_0, s_1, \ldots, s_{21}$.

- For $s_0, \ldots, s_{10}$, there is no need to add a backtracking point, because (per line 27) there is no t_d from a thread different from $tid(t)$.
- For s_{13}, the enabled transition b_3:`z++` is dependent and may be co-enabled with $t_d = a_{10}$:`z++`. (We assume *lock-atomicity* by grouping variable accesses with protecting lock/unlock and regarding each block as atomic.) However, since the search subspace from s_8 is race-free, we do not add backtracking points at s_8.
- For s_{14}, the enabled transition b_4:`c=x` is dependent and may be co-enabled with $t_d = a_2$:`x++`. Since the search subspace from s_0 has a potential race condition between a_6 and b_{10}, we set $s_0.backtrack = \{2\}$ to make sure that in a future execution, thread T_2 is scheduled at state s_0.

After this, PDPSEARCH(S, s_i) keeps returning for all $i > 0$ as indicated by lines 20-21. Since $s_0.backtrack = \{2\}$, PDPSEARCH(S, s_0) executes lines 16-20. The next execution starts from $s_0 \xrightarrow{b_1} s'$.

Proof of Correctness The correctness of the overall algorithm is summarized as follows: First, any race condition reported by PDPSEARCH is guaranteed to be real.

Second, if PDPSEARCH returns without finding any race condition, the program is guaranteed to be race-free under the given input. Finally, PDPSEARCH always returns a conclusive result (either race-free or a concrete race) for terminating programs. If a program is nonterminating, PDPSEARCH can be used for bounded analysis as in CHESS [14]—to detect bugs up to a bounded number of steps. The soundness is ensured by the fact that it is concretely executing the actual program within its target environment. The completeness (for terminating programs) can be established by the following arguments: (1) the baseline DPOR algorithm as in [5] is known to be sound and complete for detecting race conditions; and (2) our trace-based lockset analysis is conservative in checking race-free subspaces. The procedure returns 'yes' only if no race condition can be reached by any execution in the search subspace.

```
1:  Initially: S is empty; PDPSEARCH(S, s_0)

2:  PDPSEARCH(S, s) {
3:    if (s.enabled = ∅) {
4:      for (i = 0; i < S.size(); i++) {
5:        let s_b be the i-th element in S;
6:        for each (t ∈ s_b.enabled)
7:          PDPUPDATEBACKTRACKSETS(S, t);
8:      }
9:    }
10:   else {
11:     if (DETECTRACE(s)) exit (S);
12:     S.push(s);
13:     let τ ∈ Tid such that ∃t ∈ s.enabled : tid(t) = τ;
14:     s.backtrack ← {τ};
15:     s.done ← ∅;
16:     while (∃t: tid(t) ∈ s.backtrack and t ∉ s.done) {
17:       s.done ← s.done ∪ {t};
18:       s.backtrack ← s.backtrack \ {tid(t)};
19:       let s' ∈ S such that s -t-> s';
20:       PDPSEARCH(S, s');
21:       S.pop(s);
22:     }
23:   }
24: }

25: PDPUPDATEBACKTRACKSETS(S, t) {
26:   let T = {t_1, ..., t_n} be the sequence of transitions associated with S;
27:   let t_d be the latest transition in T that (1) is dependent and may be co-enabled with t, and
      (2) let s_d ∈ S be the state from which t_d is executed, SubspaceRaceFree(s_d) is FALSE;
28:   if (t_d ≠ null){
29:     let E be {q ∈ s_d.enabled | either tid(q) = tid(t), or q was executed after t_d in T and
        a happens-before relation exists for (q, t)}
30:     if (E ≠ ∅)
31:       choose any q in E, add tid(q) to s_d.backtrack;
32:     else
33:       s_d.backtrack ← s_d.backtrack ∪ {tid(q) | q ∈ s_d.enabled};
34:   }
35: }
```

Fig. 7. Property driven pruning based dynamic race detection algorithm

5 Experiments

We have implemented the proposed method on top of our implementation of the DPOR algorithm, inside Inspect [23]. We use CIL [15] for parsing, whole-program static analysis, and source code instrumentation. Our tool is capable of handling multithreaded C programs written using the Linux POSIX thread library. The source code instrumentation consists of the following steps: (1) for each shared variable access, insert a request to the scheduler asking for permission to execute; (2) for each thread library routine, add a wrapper function which sends a request to the scheduler before executing the actual library routine; (3) for each branch, add recording routines to notify about the branch begin and end, the shared variables and the lock updates in the other branch.

In order to control every visible operation, we need to identify the set of shared variables during the source code instrumentation. Shared variable identification requires a conservative static analysis of the concurrent program, e.g., pointer and may-escape analysis [19, 11]. Since this analysis [19] is an over-approximated analysis, our instrumentation is safe for intercepting all visible operations of the program. This ensures that we do not miss any bug due to missing identification of a shared variable. Similarly, when a whole program static analysis is either ineffective or not possible (due to missing source code) to identify the precise locksets, during instrumentation, we resort to subsets of acquired locks and supersets of released locks.

We have conducted experimental comparison of our new method with the baseline DPOR algorithm. The benchmarks are Linux applications written in C using the POSIX thread library; many are obtained from public domain including `sourceforge.net` and `freshmeat.net`. Among the benchmarks, `fdrd2` and `fdrd4` are variants of our running example. `qsort` is a multithreaded quick sort algorithm. `pfscan` is a file scanner implemented using multiple threads to search directories and files in parallel; the different threads share a dynamic queue protected by a set of mutex locks. `aget` implements a ftp client with the capability of concurrently downloading different segments of a large file. `bzip2smp` is a multithreaded version of the Linux application `bzip`. All benchmarks are accompanied by test cases to facilitate the concrete execution. Our experiments were conducted on a workstation with 2.8 GHz Pentium D processor and 2GB memory running Fedora 5.

Table 1 shows the experimental results. The first seven columns show the statistics of the test cases, including the name, the lines of C code, the number of threads, the number of shared variables, the number of shared variable accesses, the number of locks, and the number of data races. Columns 8-13 compare the two methods in terms of the runtime, and the number of executed transitions, and the number of completed execution traces. For DPOR, every completed trace (reported in Column 12) belongs to a distinct equivalence class of interleavings; however, many of them are pruned away by PDP since they are redundant as far as race detection is concerned. Columns 14-16 provide the following statistics of PDP: the number of race-free checks, the number of race-free check successes, and the number of skipped backtrack points.

The results show that our PDP method is significantly more efficient than DPOR in pruning the search space. For all examples, PDP took significantly less time in either finding the same data race or proving the race freedom; the number of transitions/traces that PDP has to check during the process was also significantly smaller. Although the

Table 1. Comparing the performance of two race detection algorithms (with 1 hour time out)

Test Program							Runtime (s)		# of Trans (k)		# of Traces		Race-free Chk		
name	loc	thrd	gvar	accs	lock	race	dpor	PDP	dpor	PDP	dpor	PDP	chks	yes	skip
fdrd2	66	2	3	3	2	1	3	1	2	0.6	89	14	88	75	75
fdrd4	66	2	3	3	2	1	11	3	10	4	233	68	232	165	165
qsort	743	2	2	2000	5	0	17	8	12	8	4	1	2	2	2
pfscan-good	918	2	21	118	4	0	179	15	71	10	2519	182	398	217	217
pfscan-bug	918	2	21	39	4	1	3	1	1	1	31	10	5	5	6
aget-0.4	1098	3	5	72	1	0	183	1	103	0.1	3432	1	6	6	9
aget-0.4	1098	4	5	78	1	0	>1h	1	-	0.1	-	1	9	9	18
aget-0.4	1098	5	5	84	1	0	>1h	1	-	0.1	-	1	12	12	30
bzip2smp	6358	4	9	18	3	0	128	3	63	2	1465	45	48	5	5
bzip2smp	6358	5	9	18	3	0	203	4	99	2	2316	45	48	5	7
bzip2smp	6358	6	9	18	3	0	287	4	135	2	3167	45	48	5	9
bzip2smp2	6358	4	9	269	3	0	291	136	63	21	1573	45	48	5	5
bzip2smp2	6358	5	9	269	3	0	487	155	85	21	2532	45	48	5	7
bzip2smp2	6358	6	9	269	3	0	672	164	116	21	3491	45	48	5	9
bzip2smp2	6358	10	9	269	3	0	1435	183	223	21	7327	45	48	5	17

average time for PDP to complete one execution is longer than DPOR, e.g., 4066 ms vs. 195 ms as indicated by data from the last row of Table 1 (due to the overhead of tracking branch begin/end and other auxiliary transitions), the overhead in PDP is well compensated by the skipped executions due to property driven pruning.

6 Conclusions

We have proposed a new data race detection algorithm that combines the power of dynamic model checking with property driven pruning based on a lockset analysis. Our method systematically explores concrete thread interleavings of a program, and at the same time prunes the search space with a trace-based conservative analysis. It is both sound and complete (as precise as the DPOR algorithm); at the same time, it is significantly more efficient in practice, allowing the technique to scale much better to real-world applications. For future work, we would like to extend the proposed framework to check other types of properties. Since race detection is a problem of simultaneous reachability of two transitions, the techniques developed here should be readily applicable to checking deadlocks and many other simple safety properties.

References

[1] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *International Conference on Concurrency Theory*, pages 135–150. Springer, 1997. LNCS 1243.

[2] F. Chen and G. Rosu. Parametric and sliced causality. In *Computer Aided Verification*, pages 240–253. Springer, 2007. LNCS 4590.

[3] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proceedings Workshop on Logics of Programs*, pages 52–71, Berlin, 1981. Springer-Verlag. LNCS 131.

[4] D. Engler and K. Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *ACM Symposium on Operating Systems Principles*, pages 237–252. ACM, 2003.

[5] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *Principles of programming languages*, pages 110–121, 2005.

[6] C. Flanagan, K. Leino, M. Lillibridge, G. Nelson, J. Saxe, and R. Stata. Extended static checking for java. In *PLDI*, pages 234–245, 2002.

[7] M. Ganai, S. Kundu, and R. Gupta. Partial order reduction for scalable testing of SystemC TLM designs. In *Design Automation Conference*, 2008.

[8] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*. Springer, 1996. LNCS 1032.

[9] P. Godefroid. Software model checking: The VeriSoft approach. *Formal Methods in System Design*, 26(2):77–101, 2005.

[10] G. Holzmann, E. Najm, and A. Serhrouchni. SPIN model checking: An introduction. *STTT*, 2(4):321–327, 2000.

[11] V. Kahlon, Y. Yang, S. Sankaranarayanan, and A. Gupta. Fast and accurate static data-race detection for concurrent programs. In *Computer Aided Verification*, pages 226–239. Springer, 2007. LNCS 4590.

[12] N. Leveson and C. Turner. Investigation of the therac-25 accidents. *IEEE Computer*, 26(7):18–41, 1993.

[13] A. W. Mazurkiewicz. Trace theory. In *Advances in Petri Nets*, pages 279–324. Springer, 1986. LNCS 255.

[14] M. Musuvathi and S. Qadeer. CHESS: Systematic stress testing of concurrent software. In *Symposium on Logic-Based Program Synthesis and Transformation*, pages 15–16. Springer, 2006. LNCS 4407.

[15] G. Necula, S. McPeak, S. Rahul, and W. Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *International Conference on Compiler Construction*, pages 213–228. Springer, 2002. LNCS 2304.

[16] N. Nethercote and J. Seward. Valgrind: A program supervision framework. *Electr. Notes Theor. Comput. Sci.*, 89(2), 2003.

[17] P. Pratikakis, J. Foster, and M. Hicks. LOCKSMITH: context-sensitive correlation analysis for race detection. In *PLDI*, pages 320–331. ACM, 2006.

[18] J. P. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the Fifth Annual Symposium on Programming*, 1981.

[19] A. Salcianu and M. Rinard. Pointer and escape analysis for multithreaded programs. In *Principles and Practices of Parallel Programming*, pages 12–23. ACM Press, 2001.

[20] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.

[21] K. Sen, G. Rosu, and G. Agha. Detecting errors in multithreaded programs by generalized predictive analysis of executions. In *Formal Methods for Open Object-Based Distributed Systems*, pages 211–226. Springer, 2005. LNCS 3535.

[22] J. Voung, R. Jhala, and S. Lerner. RELAY: static race detection on millions of lines of code. In *Foundations of Software Engineering*, pages 205–214. ACM, 2007.

[23] Y. Yang, X. Chen, and G. Gopalakrishnan. Inspect: A Runtime Model Checker for Multithreaded C Programs. Technical Report UUCS-08-004, University of Utah, 2008.

[24] Y. Yang, X. Chen, G. Gopalakrishnan, and R. Kirby. Efficient stateful dynamic partial order reduction. In *SPIN Workshop on Model Checking Software*, 2008.