Precisely Deciding Control State Reachability in Concurrent Traces with Limited Observability

Chao Wang and Kevin Hoang

Department of ECE, Virginia Tech, Blacksburg, VA 24061, USA

Abstract. We propose a new algorithm for precisely deciding a *control state reachability (CSR)* problem in runtime verification of concurrent programs, where the trace provides only limited observability of the execution. Under the assumption of limited observability, we know only the type of each event (read, write, lock, unlock, etc.) and the associated shared object, but not the concrete values of these objects or the control/data dependency among these events. Our method is the first *sound and complete* method for deciding such CSR in traces that involve more than two threads, while handling both standard synchronization primitives and *ad hoc* synchronizations implemented via shared memory accesses. It relies on a new *polygraph* based analysis, which is provably more accurate than existing methods based on lockset analysis, acquisition history, universal causality graph, and a recently proposed method based the *causally-precedes* relation. We have implemented the method in an offline data-race detection tool and demonstrated its effectiveness on multithreaded C/C++ applications.

1 Introduction

The idea of using an offline analysis of the trace log to predict subtle bugs of a concurrent system has been the focus of intense research in recent years. The core problem in this analysis is to decide an instance of the *control state reachability (CSR)* problem: given a valid execution trace ρ, decide whether there exists an *alternative interleaving* ρ' of the events of the trace that can lead to a bad system state, e.g. one that manifests a data-race, a deadlock, or an atomicity violation. Although there exists a large body of work on trace based analysis for predicting concurrency bugs, e.g. using either under-approximation [19, 20, 7, 24] or overapproximation [18, 26, 31, 4], none of the existing methods can precisely decide CSR for input traces with *limited observability*.

Under the assumption of limited observability, the input trace records only the *global* operations but not *thread-local* operations. Even for the global operations, such as thread synchronizations and shared memory accesses, we only know the event types and the associated shared objects, but not the concrete values of these objects or the control/data dependency among the events. For instance, executing the code `tmp:=X; Y:=tmp+10;` would produce events `READ(X)` and `WRITE(Y)` in the trace log. However, we would not know the concrete values of X and Y, or whether `WRITE(Y)` is data-dependent on `READ(X)`. As another example, executing the code `if(X==10) Y:=0` would produce the same `READ(X)` and `WRITE(Y)` events in the trace log.

Precisely deciding CSR under limited observability is challenging for two reasons. First, we need to characterize the set of bugs that can be predicted, with certainty, by

analyzing only the input trace under limited observability. Second, we need to design a new algorithm that can produce the exact set of *predictable* bugs. In other words, the algorithm should report a bug in the trace if and only if the bug is guaranteed to show up in some actual program execution. However, to the best of our knowledge, there does not exist any method that can precisely decide this CSR problem. For example, classic methods based on lockset analysis [18] may produce false alarms due to overapproximation, whereas classic methods based on the happens-before causality relation [13] may miss real bugs due to underapproximation.

For two threads synchronizing via nested locks only, Kahlon *et al.* [9] proposed the first sound and complete algorithm for deciding CSR based on a lock acquisition history (LAH) analysis. They subsequently proposed a lock causality graph (LCG) analysis [8], which generalizes LAH to handle also non-nested, but finite-length, lock chains. The theoretical significance of LAH and LCG is that they prove the decidability of CSR under certain synchronization patterns even for concurrently running recursive procedures. However, neither LAH nor LCG considers synchronization primitives other than locks. Kahlon and Wang [10] proposed a universal causality graph (UCG) analysis, which generalizes LCG to handle non-lock synchronization primitives as well. However, UCG is sound and complete for deciding CSR in traces that involve only two threads. For more than two threads, UCG may produce false alarms.

In this paper, we propose the first sound and complete algorithm for deciding CSR for input traces that involve more than two threads, while precisely handling both standard synchronizations and *ad hoc* synchronizations via shared memory accesses. We introduce a new *polygraph* based analysis framework, which is provably more accurate than the existing UCG analysis. The polygraph abstraction allows us to strengthen the CSR decision procedure to make it sound and complete for traces that involve an arbitrary, but fixed, number of threads. Note that, being part of a testing procedure, our method do not guarantee to detect all bugs in the program. Instead, our *sound and complete* argument is restricted to the set of *predictable* bugs for the given input execution trace.

We also introduce a new *predictive model* to more accurately model not only standard synchronization primitives such as locks and wait/notify (as in UCG), but also *ad hoc* synchronizations implemented using shared memory accesses. This can significantly increase the precision of bug detection. When being applied to data-race detection, for instance, our method will be provably more accurate than both UCG and a more recent *causally precedes* (CP [24]) analysis. Similar to ours, the CP method does not report false alarms. However, it accomplishes this by aggressively dropping valid interleavings, in a way that may lead to missed bugs. Our method does not have this problem.

We have implemented the new method in an offline data-race detection tool based on the LLVM platform. Our experiments conducted on a set of multithreaded C/C++ applications show that the new method is indeed more accurate that the existing ones.

To sum up, we have made the following contributions:

- We propose a new *polygraph* based analysis for precisely deciding CSR in execution traces with only limited visibility. The method is sound and complete for an arbitrary, but fixed, number of concurrent threads.

– We implement the new method inside an offline data-race detection tool and demonstrate that, in addition to being provably more accurate, it actually reports more real bugs than CP and fewer false alarms than UCG.

2 Preliminaries

The control state reachability (CSR) problem arises from trace based analysis of multithreaded programs for detecting subtle concurrency bugs. Given a concrete execution trace ρ of the program, this analysis typically consists of three steps:

1. Identify a set of *potential* bugs by scanning the trace ρ for known error patterns.
2. For each potential bug, create an error condition EC and check for *feasibility*, i.e. EC can be satisfied by some valid interleaving ρ' of the concurrent events in ρ.
3. Compute a thread schedule for each bug found in Step 2 for deterministic replay.

Our main contribution lies in step 2, where our new algorithm can precisely decide the feasibility of EC. This is equivalent to deciding the CSR problem, where we are concerned with the simultaneous reachability of some locations in the concurrent threads communicating through standard and *ad hoc* synchronization operations.

Let the input trace be a sequence $\rho = e_1, \ldots, e_n$, where each event e_i $(1 \leq i \leq n)$ models an operation in thread T_i of one of the following types:

– $fork(thrd)$ for the creation of child thread $thrd$; and
– $join(thrd)$ for the join back of child thread $thrd$.
– $acq(lk)$ for acquiring lock lk;
– $rel(lk)$ for releasing lock lk;
– $signal(cv)$ for sending signal via condition variable cv;
– $wait(cv)$ for receiving signal via condition variable cv;
– $R(sh)$ for reading from shared variable sh;
– $W(sh)$ for writing to shared variable sh.

For example, the wait operation in POSIX threads is of the form $wait(cv, lk)$, where cv is a condition variable and lk is a lock. Based on the POSIX standard, this operation consists of three substeps. First, the thread releases lk to allow other threads to acquire lk and execute $signal(cv)$. Next, the thread enters the sleep mode. Finally, after the signal operation is executed by another thread, the thread wakes up and acquires lk again. There, the entire wait operation is equivalent to $rel(lk)$; $wait(cv)$; $acq(lk)$.

We assume that the input trace ρ is feasible because it represents a real execution. If ρ itself exposes a bug, we are done. Otherwise, we check whether there exists a permutation ρ' that exposes a bug. Permutation ρ' is feasible (or real) if ρ' can appear in some actual execution of the program. Furthermore, we assume that it is not possible to run the original program again to test the validity of ρ'. Instead, we define a statically checkable condition over ρ, under which permutation ρ' is guaranteed to be feasible.

First, we define the condition for threads synchronizing via standard synchronization primitives. Let $\rho = e_1, \ldots, e_n$ be the input trace and $\rho' = e'_1, \ldots, e'_n$ be a permutation, where for all $1 \leq i, j \leq n$, each event e'_i maps to a unique e_j and vice versa. Let $e_i \rightarrow e_j$ denote that e_i appears before e_j when the two events are in different threads.

Let $e_i <_{PO} e_j$ denote that e_i appears before e_j in the same thread. The conditions for ρ' is to be feasible are as follows (c.f. [10]):

1. **program order**: events within each thread must follow their program order. That is, $e_i <_{PO} e_j$ if e_i appears before e_j in ρ and both events are from the same thread.
2. **fork/join order**: events in a child thread t must appear after the $fork(t)$ event, but before the $join(t)$ event, of the parent thread.
3. **signal/wait order**: events in each matching $signal(cv)$ and $wait(cv)$ pair must appear in the same order as they appear in the input trace ρ.
4. **acq/rel order**: events from two matching lock/unlock pairs, e.g. (acq_1, rel_1) and (acq_2, rel_2) over the same lock, must be mutually exclusive. Since critical sections should not interleave, either $rel_1 \rightarrow acq_2$ or $rel_2 \rightarrow acq_1$.

For threads that do not synchronize via shared memory accesses, these are both sufficient and necessary conditions for ρ' to be feasible. However, in the general case, they are only necessary conditions. That is, if any condition is violated, ρ' is guaranteed to be infeasible. But even if all of them are satisfied, ρ' may still be infeasible. It is worth pointing out that, since the input trace ρ is feasible, every lock acquired by a thread in ρ must have been released by the same thread. To overcome the problem, we add the following condition:

5. **write/read order**: events from two matching write/read pairs, e.g. (W_1, R_1) and (W_2, R_2), where $R_1(x)$ reads the value set by $W_1(x)$ and $R_2(x)$ reads the value set by $W_2(x)$, must not interfere. They should satisfy $W_1 \rightarrow R_1$, $W_2 \rightarrow R_2$, and in addition, either $R_1 \rightarrow W_2$ or $R_2 \rightarrow W_1$.

This condition may not hold in all execution traces, but instead, is imposed by the given input trace ρ. The condition ensures that, as long as ρ is feasible, ρ' is also feasible, even if we do not have any information about the program that generates the input trace ρ. If there exist a write event in ρ that does not have any matching read event, we add a dummy read event immediately after the write event.

We assume that ρ provides limited observability of the program. For example, we do not know whether the read event $R(x)$ comes from `a:=x+5` or `if(x>10)` or `if(x<0)`. Similarly, we do not know whether the write event $W(x)$ comes from `x:=10` or `x:=0`. Given the sequence $R(x) \ldots W(sh)$, we do not know whether $W(sh)$, which may come from `a:=sh`, is control-dependent on $R(x)$, which may come from `if(x>0)`. Assume that `a` and `b` are thread-local, traces from the following programs are indistinguishable:

- **program 1:** `if (x==0) { b:=0; } a:=sh;` or
- **program 2:** `if (x!=0) { a:=sh; }` or
- **program 3:** `{ b:=x; a:=sh; }`

Nevertheless, we show that, by requiring each $R_1(x)$ in ρ' to read from the same $W_1(x)$ as in ρ, we can ensure that $R(sh)$ will be executed in ρ', regardless of the expression in the if-condition, and whether the if-condition is guarding $R(sh)$.

We want to stress that the core analysis procedure proposed in this paper is not tied up to whether the write/read order (Condition 5) is used or not. To make our subsequent presentation clear, we define the following two types of predictive models:

– The **CSR model** includes Conditions 1, 2, 3, and 4, but not Condition 5.
– The **CDSR model** includes Conditions 1, 2, 3, 4, and 5.

The CSR model considers only standard synchronizations, whereas the CDSR model also considers *ad hoc* synchronizations implemented by using shared memory accesses. In the sequel, we shall present our new polygraph based analysis method for the CSR model first, and then extend it the CDSR model.

3 Polygraph Based Causality Analysis

Deciding the feasibility of an error condition EC is challenging mainly due to *interleaving explosion* – the number of possible interleavings is often exponentially large. Therefore, naively enumerating the feasible interleavings and checking them against EC is not practical. Instead, we rely on checking a new *polygraph* where deciding the feasibility of EC is equivalent to deciding the absence of cycles in the graph.

3.1 From Input Trace ρ to Polygraph G_ρ

A *polygraph* is a generalization of a directed graph that we use to capture all feasible interleavings of the events of an input trace. The term was coined by Papadimitriou [16] while studying view serializability: nodes in his polygraph are requests and responses of database transactions, whereas in our case, they are events of a multithreaded program. Let the input trace be ρ. The ρ-induced polygraph, denoted $G_\rho = (V, E, E_{poly})$, consists of a set V of nodes, a set E of edges, and a set E_{poly} of polyedges:

– Each node in V models an event in ρ.
– Each edge in E, denoted $a \to b$, means a must appear before b. Initially, these edges come from the program order, fork/join, and signal/wait as defined in Section 2.
– Each polyedge in E_{poly}, denoted $(a \to b, c \to d)$, represents an *either-or* choice, meaning that either a appears before b, or c appears before d.

For the CSR model (Section 2), the polyedges come from the **acq-rel** event pairs:

– For any two *acq-rel* event pairs over lock lk, say (acq_1, rel_1) and (acq_2, rel_2), their mutual exclusion demands that either rel_1 appears before acq_2, or rel_2 appears before acq_1. In G_ρ, this is modeled by polyedge $(rel_1 \to acq_2, rel_2 \to acq_1)$.

This defines the set of interleavings in the CSR model, for which we set out to design a sound and complete algorithm for checking the feasibility of EC.

Later, in Section 5, we will add another type of polyedges for modeling the non-interference properties of the write-read pairs (the CDSR model in Section 2). If the threads synchronize solely via standard synchronization primitives – without using shared variable accesses – the CSR model would be precisely for predicting all the real bugs (w.r.t. the input trace). Otherwise, we need to consider the CDSR model. Regardless, our core analysis procedure works for both models.

3.2 From Error Condition EC to Polygraph $G_\rho(EC)$

Given an error condition EC, e.g. representing a potential data-race, we construct a new polygraph $G_\rho(EC)$, which is a specialization of G_ρ for checking EC. We model EC by adding a set of new edges to G_ρ, and then perform a slicing of G_ρ, to remove all polyedges that are irrelevant to satisfying EC. Below are examples for modeling some typical concurrency bugs:

- *Data-race*: Let (a, b) be the events that have a potential data-race. These potential data-races may be computed using standard lockset analysis [18]. That is, we compute the set of locks held by a thread at each of its program locations. Then, for any two events a and b that access the same memory from different threads without holding a common lock, there is a potential data-race. Let a' and b' be the immediate preceding events of a and b in the two threads, respectively. The error condition EC consists of edges $a' \rightarrow b$ and $b' \rightarrow a$.
- *Atomicity violation*: Let a and b be two events that are intended to execute atomically in one thread, and c be an interfering event in another thread. Here, c interferes with a (and b) if and only if they access the same memory location with at least one write event. In such case, the event order $a < c < b$ indicates a violation. The error condition EC consists of edges $a \rightarrow c$ and $c \rightarrow b$.
- *Order violation*: Let the event sequence $a_1, \ldots, a_k$ be an unintended execution order. The error condition EC consists of edges $a_1 \rightarrow a_2$, $a_2 \rightarrow a_3$ ..., and $a_{k-1} \rightarrow a_k$, since the violation is exposed when all these edges are satisfied.

Slicing $G_\rho(EC)$ with respect to EC broadens the coverage and allows more real bugs to be detected. Recall that our goal is to find a valid interleaving that can lead to the satisfaction of all edges in EC. However, the valid interleaving does not have to be a permutation of the whole input trace ρ. Instead, a subsequence or prefix, from the initial state to EC, would suffice. Therefore, we remove from $G_\rho(EC)$ any happens-before obligation (edges and polyedges) that are not needed for satisfying EC.

3.3 Resolving the Polyedges to Detect Cycles

Since each edge in $G_\rho(EC)$ represents a *precedence* relation that must be satisfied, a cycle in this graph means that no valid interleaving exists. Therefore, we decide the feasibility of EC by detecting cycles in $G_\rho(EC)$. Sometimes, $G_\rho(EC)$ has no cycle initially, but existing edges may force the *either-or* decisions of some polyedges, which in turn lead to cycles. This *polyedge resolution* process is often triggered by the addition of the EC edges. It is iterative because resolving one polyedge may introduce a new regular edge that triggers the resolution of another polyedge.

For the CSR model (Section 2), we use the following polyedge resolution rule:

Rule 1: For any polyedge $(rel_1 \rightarrow acq_2, rel_2 \rightarrow acq_1)$, if there already exists a regular edge $s \rightarrow t$ such that $acq_1 <_{PO} s$ and $t <_{PO} rel_2$, we remove the polyedge and add regular edge $rel_1 \rightarrow acq_2$ (see the three cases in Fig. 1). The reason is that, the other choice $rel_2 \rightarrow acq_1$ would have formed a cycle with $s \rightarrow t$.

Therefore, our procedure starts by resolving all polyedges whose choices are forced by some existing edges. It repeats the process until (1) a cycle is detected, meaning that

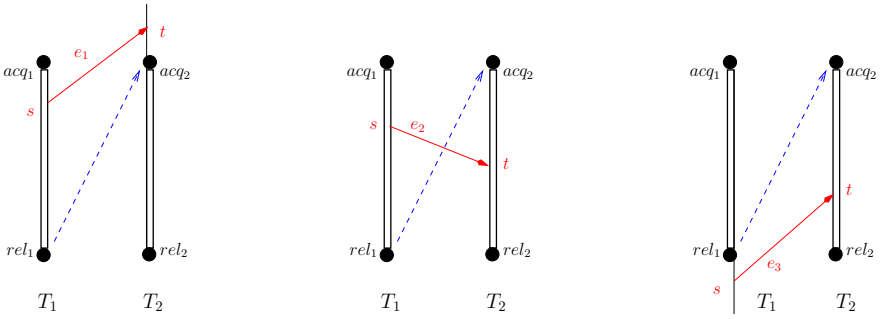

Fig. 1. Polyedge resolution due to existing edge $s \rightarrow t$, which adds $rel_1 \rightarrow acq_2$.

no feasible interleaving exists in $G_\rho(EC)$; (2) all polyedges are resolved, meaning that a feasible interleaving is found; or (3) there are still some unresolved polyedges. In the third case, the problem remains undecided.

By now, our new polygraph based procedure matches the precision of the UCG [10], although the underlying decision mechanisms are drastically different. UCG does not rely on polygraph, but instead on a set of custom made inference rules. Our use of polygraph allows the new analysis framework to be easily extended to handle not only standard synchronization primitives but also *ad hoc* synchronizations such as shared memory read/write accesses.

If the goal is to design an over-approximated analysis, the undecided EC in Case 3 may be reported as a potential bug. If the goal is to design an under-approximated analysis, the undecided EC in Case 3 may be dropped. Both would lead to a loss of precision. In the next section, we shall propose a new method for resolving the third case, thereby avoiding the precision loss.

4 Generalizing the Algorithm to *k* Threads

We start by proving that the polyedge resolution algorithm in Section 3.3 (Rule 1) is sound and complete for two threads communicating via standard synchronizations. Then, we show why it does not work for traces with more than two threads. Finally, we extend Rule 1 to make it sound and complete for traces with more than two threads.

Theorem 1. *If our algorithm defined in Section 3.3 generates a cycle in $G_\rho(EC)$, there does not exist any valid interleaving in the CSR predictive model that satisfies EC.*

The proof is straightforward because all edges in $G_\rho(EC)$ represent *precedence* relations that must hold at all time. Thus, a cycle meaning that EC is not satisfiable. □

Theorem 2. *If our algorithm defined in Section 3.3 does not generate a cycle in $G_\rho(EC)$, for 2 threads, a valid interleaving always exists in the CSR predictive model.*

The proof consists of two cases. First, if all polyedges are resolved at the end of the iterative process and there is no cycle, since nondeterminism is removed completely,

EC is satisfiable. Second, if some polyedges still remain un-resolved at the end of the iterative process, we show that they can always be resolved as follows:

1. Pick a polyedge arbitrarily and replace it with one of the *either-or* edges.
2. Apply Rule 1 to resolve the affected polyedges.
3. Repeat the above steps until all polyedges are resolved.

We prove, by contradiction, that the above process would not create any cycle for two threads. Assume that resolving polyedge $\langle rel_1 \rightarrow acq_2, rel_2 \rightarrow acq_1 \rangle$ into regular edge $rel_1 \rightarrow acq_2$ creates a cycle subsequently. With two threads, the cycle must involve edges $rel_1 \rightarrow acq_2 <_{PO} s \rightarrow t <_{PO} rel_1$, where $s \rightarrow t$ is an existing edge. However, based on Rule 1, since edge $s \rightarrow t$ already exists, it should have resolved the polyedge already. Therefore, our assumption is not correct. The theorem is proved. □

A byproduct is that, for two threads, the *schedule reconstruction* procedure as described above can always generate a valid thread interleaving in polynomial time.

4.1 From 2 Threads to 3 Threads

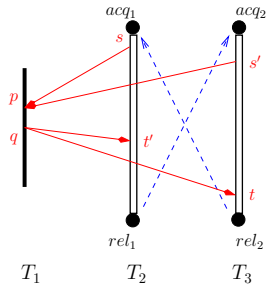

Fig. 2. Example (3 threads): There is no valid interleaving and the polygraphs have no cycle.

The proof in Theorem 2 does not work for trees with 3 threads, as shown in Fig. 2. Here, we have four regular edges ($s \rightarrow p$, $s' \rightarrow p$, $q \rightarrow t'$, and $q \rightarrow t$) and one polyedge $\langle rel_1 \rightarrow acq_2, rel_2 \rightarrow acq_1 \rangle$. While there is no feasible interleaving, there is no cycle either. The reason why there is no feasible interleaving is due to the transitive precedence constraints $s \rightsquigarrow t$ and $s' \rightsquigarrow t'$. However, notice that none of the regular edges alone can resolve the polyedge based on Rule 1. Furthermore, both choices of the polyedge would create a cycle.

A straightforward way to strengthen the algorithm is to include transitive edges such as $s \rightsquigarrow t$ in Rule 1. For example, in Fig. 2, one transitive edge is $s \rightarrow p <_{PO} q \rightarrow t$ and another is $s' \rightarrow p <_{PO} q \rightarrow t'$. With the modified Rule 1, these two transitive edges would lead to a cycle. Unfortunately, although this fix works for traces with 3 threads, it does not work for traces with 4 threads, as shown by Fig. 3.

Fig. 3 has two polyedges that cannot be resolved by the existing edges and their transitive edges. Therefore, the polygraph is cycle-free. However, there does not exist

a feasible interleaving either. Consider all four cases for resolving the two polyedges – all would lead to contradictions (cycles):

- If we select $rel'_1 \rightarrow acq'_2$ from the second polyedge, transitive edge $s \rightarrow q_1 <_{PO} rel'_1 \rightarrow acq'_2 <_{PO} p_2 \rightarrow t$ would induce $rel_1 \rightarrow acq_2$; bug transitive edge $s' \rightarrow q_1 <_{PO} rel'_1 \rightarrow acq'_2 <_{PO} p_2 \rightarrow t'$ would induce $rel_2 \rightarrow acq_1$.
- If we select $rel'_2 \rightarrow acq'_1$ from the second polyedge, transitive edge $s \rightarrow q_2 <_{PO} rel'_2 \rightarrow acq'_1 <_{PO} p_1 \rightarrow t$ would induce $rel_1 \rightarrow acq_2$; but transitive edge $s' \rightarrow q_2 <_{PO} rel'_2 \rightarrow acq'_1 <_{PO} p_1 \rightarrow t'$ would induce $rel_2 \rightarrow acq_1$.

Therefore, we need to strengthen the algorithm further for traces with 4 or more threads.

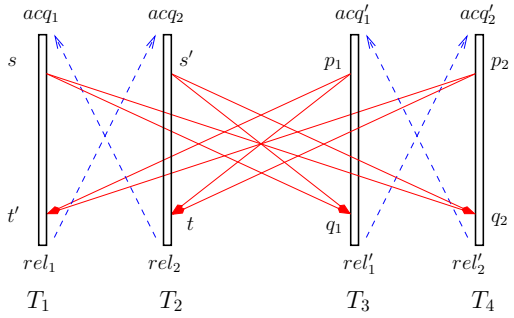

Fig. 3. Example (4 threads): There is no valid interleaving and the polygraphs have no cycle.

4.2 Heuristics for Resolving the Remaining Polyedges

Before continuing our effort on strengthening Rule 1, let us pause for a moment to consider the current polygraph $G_\rho(EC)$, which (1) has no cycle, and also (2) has some unresolved polyedges. Therefore, either it has a feasible interleaving, or it does not. Recall that Rule 1 is geared toward proving infeasibility (by finding a cycle). What if EC actually is feasible? In this case, the best strategy is not to find a cycle (since it does not exist) but to find a valid interleaving. Toward this end, we must resolve the remaining polyedges in a consistent fashion, for example, by employing our *schedule reconstruction* algorithm proposed as part of the proof for Theorem 2.

Recall that in the proof for Theorem 2, for each unresolved polyedge, we arbitrarily pick one of the either-or choices and then apply Rule 1 to propagate its impact on the remaining polyedges. If we can resolve all polyedges without creating a cycle, we have proved the feasibility of the EC. However, since the *schedule reconstruction* algorithm resolves polyedges arbitrarily, it may not be the best strategy for finding a feasible interleaving if there exists one.

Fig. 4 shows that, if we make the wrong decision, by choosing $rel_1 \rightarrow acq_2$ first to resolve the polyedge on the left-hand side, it would lead to a cycle regardless of how we resolve the second polyedge. Notice that in this example, there actually exists a feasible

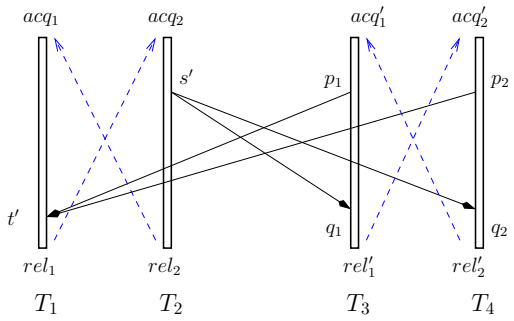

Fig. 4. There exists a valid interleaving, but arbitrarily selecting edges from unresolved polyedges, e.g. $rel_1 \rightarrow acq_2$, may lead to a cycle. Note that $s' \rightsquigarrow t'$ always holds.

interleaving: it is possible to resolve both polyedges, e.g. by picking $rel_2 \rightarrow acq_1$ and $rel'_2 \rightarrow acq'_1$, while avoiding the creation of any cycle.

Therefore, we use a *causally precedes (CP)* relation [24] as guidance to increase the success rate of the schedule reconstruction. That is, we impose a strict precedence order between any two critical sections that share conflicting data accesses – two accesses of the same memory location and at least one of them is a write. Enforcing the CP relation takes a polynomial time w.r.t. the trace length, and the main advantage is that, if the graph remains acyclic, EC is guaranteed to be feasible (c.f. [24]). More specifically, our use of the CP relation based heuristic is as follows:

– For any unresolved polyedge $\langle rel_1 \rightarrow acq_2, rel_2 \rightarrow acq_1 \rangle$, if $rel_1 <_{CP} acq_2$, where $<_{CP}$ means *causally-precedes* [24], we replace it with edge $rel_1 \rightarrow acq_2$;

If the above heuristic search finds a valid interleaving, we are done. Otherwise, we resolve it in the the next step. Therefore, our overall method is at least as accurate as the CP-only method [24] for detecting bugs. In Section 6, we will show that in practice, our method is often significantly better.

4.3 Generalizing the Resolution Rule for k Threads

When the CP heuristic based search in Section 4.2 encounters a cycle, it does not mean that EC is infeasible, since the cycle may be created by a wrong decision. Therefore, we should backtrack and try again. However, a naive backtracking algorithm can be expensive: for $|E_{poly}|$ polyedges, it may take $O(2^{|E_{poly}|})$ time.

Instead, we propose a bounded lookahead search whose complexity remains polynomial in $|E_{poly}|$. Let k be the number of threads. We propose to strengthen Rule 1 with an exhaustive lookahead of k polyedges, to explore both of the either-or choices of all k polyedges. The goal is to identify hidden implications such as the ones in Fig. 3. In particular, Rule 1 is modified as follows:

Rule 1 (strengthened): For any polyedge $\langle rel_1 \rightarrow acq_2, rel_2 \rightarrow acq_1 \rangle$, we also check if there exists a path from s to t such that (1) it involves $\leq k$ polyedges along the

way and (2) all the 2^k ways of revolving this polyedges lead to $s < t$. If such path exists ($s \rightarrow t$ is only a special case), we remove the polyedge and add regular edge $rel_1 \rightarrow acq_2$, since it is implied.

This strengthened rule directly leads to the proof of the following theorem.

Theorem 3. *If our strengthened algorithm as defined in this section does not generate a cycle in $G_\rho(EC)$, a valid interleaving always exists in the CSR predictive model.*

It is polynomial in $|E_{poly}|$ for two reasons. First, in a graph with $|V|$ nodes, there are at most $O(|V|^2)$ edges to add, which bounds the number of iterations. Furthermore, adding one such edge requires a graph analysis which takes $O(|V|+|E|)$ time. Second, with k threads, we only need to check for cycles that involve at most k threads and therefore k polyedges, because larger cycles can be decomposed into these smaller cycles. Checking all possible combinations of k polyedges takes $O(2^k)$ time. With $|E_{poly}|$ polyedges, there are at most $O(|E_{poly}|^k)$ distinct cycles that need to be inspected in the k-step lookahead.

Therefore, the overall method takes $O(|V|^2\ (|V| + |E|)\ 2^k\ |E_{poly}|^k)$ time, which is polynomial in $|V|$ and E. Note that in an *offline* trace based analysis, the number of threads k is fixed, and often small, whereas the trace length $|V|$ can be arbitrarily large. Therefore, it is advantageous to have a worst-case complexity polynomial in $|V|$.

4.4 The Overall Flow

The overall algorithm is illustrated in Fig. 5. Given an input trace ρ, we first construct a polygraph G_ρ and compute a set of potential error conditions. For each error condition EC, we construct the specialized polygraph $G_\rho(EC)$ and resolve polyedges. If there exists a cycle in $G_\rho(EC)$, we conclude that the error condition cannot be satisfied. Otherwise, we search for a valid schedule using the CP heuristic. If a valid schedule is found, it is a real bug. Otherwise, we switch to the lookahead search.

When the number of events $|V|$ is large and the number of threads is more than 2, the lookahead search may become expensive. In such case, other practical *tricks* may be needed, together with our new algorithm, to control the execution time. For example, a popular technique in runtime verification of large applications is to restrict the analysis to a sliding window of say, 5000 events, as opposed to the entire trace.

Even in such case, our generalized algorithm is valuable for two reasons. First, it provides useful insight for us to understand the various sources of precision loss in the CSR analysis. Second, it provides a unified framework for us to progressively increase the precision of the CSR analysis in the practical implementation. For example, when we set the lookahead depth to 1, 2, 3, ..., our algorithm would become precise automatically for traces that involve an increasing number of threads.

5 Applying the New Algorithm to CDSR Model

For the second predictive model defined in Section 2, namely the CDSR model, our polygraph G_ρ contains not only the **acq-rel** polyedges (Section 3.2) but also another type of polyedges, called the **write-read** polyedges:

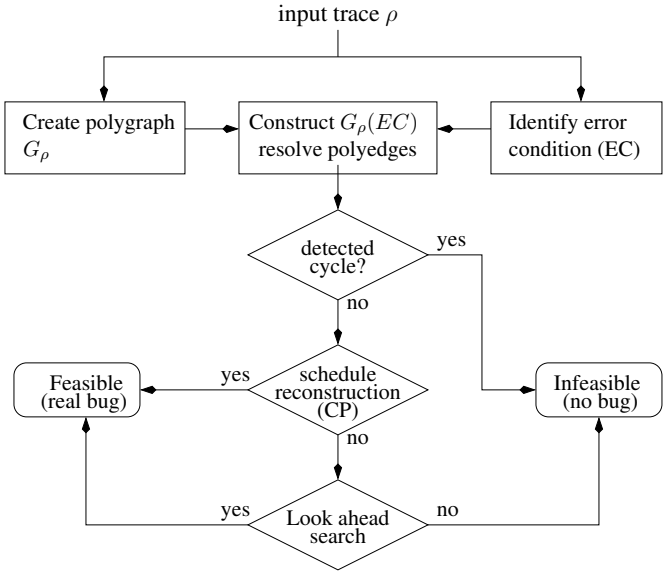

Fig. 5. Overall flow of our polygraph based prediction method.

– For any two *write-read* event pairs, denoted (W_1, R_1) and (W_2, R_2), where $R_1(x)$ reads from $W_1(x)$, and $R_2(x)$ reads from $W_2(x)$ in ρ, we maintain their write-to-read correspondence by requiring that either R_1 appears before W_2, or R_2 appears before W_1. The polyedge is denoted $\langle R_1 \to W_2, R_2 \to W_1 \rangle$.

The new polyedges ensure that all the $R(x)$ events get the same $W(x)$ events as in the input trace ρ. Although this is not required by the program semantics, it is required by the CDSR model to ensure that ρ' is feasible whenever ρ is feasible.

Interestingly, there is an analogy between the *write-read* polyedge and the *acq-rel* polyedge. The non-interference property of write-read event pairs is similar in its form – although not in its meaning – to the mutual exclusion property of the critical sections defined by the acq-rel event pairs.

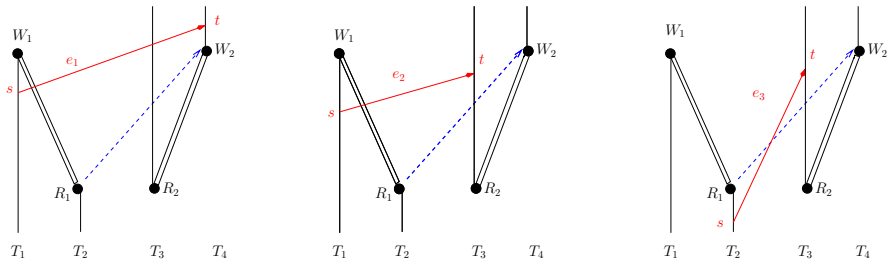

Fig. 6. Polyedge resolution due to existing edge $s \to t$, which adds edge $R_1 \to W_2$.

Therefore, similar to Rule 1 in Section 3.3, we define a new polyedge resolution rule as follows:

Rule 2. For any polyedge $\langle R_1 \rightarrow W_2, R_2 \rightarrow W_1 \rangle$, we check if there exists an edge $s \rightarrow t$ in the graph that forces the resolution of the either-or choice.

- When each write-read event pair comes from the same thread, non-interference is similar to mutual exclusion – hence the new rule is the same as Rule 1 (see the three cases in Fig. 1), except for substituting acq with W and rel with R.
- When the events from a write-read pair come from two different threads, the rule is slightly different (see the three cases in Fig. 6). In all of the three cases, if there exists an edge $s \rightarrow t$ in the graph that forces the resolution of the either-or choice, we replace the polygraph with edge $R_1 \rightarrow W_2$. More specifically, we look for edge $s \rightarrow t$ such that $W_1 < s$ and $t < R_2$. There are two ways to satisfy $W_1 < s$. One is $W_1 <_{PO} s$ as illustrated by the source nodes of edges e_1 and e_2 in Fig. 6. The other is $R_1 <_{PO} s$, which together with $W_1 \rightarrow R_1$ leads to $W_1 < s$ as illustrated by the source node of edge e_3.

Similar to the original Rule 1, the above rule works only for two threads. To handle traces with more than two threads, we strengthen Rule 2 in the same way as we strengthen Rule 1 in Section 4.3. That is, in the **Strengthened Rule 2**, we check for not only an edge $s \rightarrow t$, but also a path from s to t such that (1) it involves $\leq k$ polyedges along the way and (2) all the 2^k ways of revolving these polyedges lead to $s < t$. If such a path exists, we must replace the polyedge $\langle R_1 \rightarrow W_2, R_2 \rightarrow W_1 \rangle$ with the regular edge $R_1 \rightarrow W_2$ since it is implied.

The proof of correctness is almost identical to the one for the CSR model and therefore is omitted. We give the two theorems as follows:

Theorem 4. *If our algorithm defined in this section generates a cycle in $G_\rho(EC)$, there does not exist any valid interleaving in the CDSR model that satisfies EC.*

Theorem 5. *If our algorithm does not generate a cycle in $G_\rho(EC)$, a valid interleaving always exists in the CDSR model.*

By now, our new polygraph based analysis is already provably more accurate than the UCG analysis [10] in the following sense. First, it works not only for two threads – as in UCG – but also for an arbitrary (but fixed) number of threads. Second, the CDSR model is more accurate than the one used in UCG. Recall that UCG has only inference rules that are equivalent to our **Rule 1**, whereas our new method also has **Rule 2**.

6 Running Examples

In this section, we demonstrate the application of our new decision procedure in an offline predictive analysis for detecting data-races. We use a popular programming idiom in POSIX threads to illustrate some application specific optimizations that we made during our implementation in contrast to both UCG and CP. The program in Fig. 7 (left) shows a typical scenario for using condition variable c, which ensures that assignment

`x:=1` in thread T_1 always appears before `a:=x` in thread T_2. Here, lock l protects the concurrent accesses to the condition variable c since it is shared by both threads.

According to the POSIX standard, if T_2 enters the critical section ($l_7 \dots l_{12}$) first, the execution of `wait(c,l)` would release lock l and block. At this time, thread T_1 has to execute `signal(c)` and then `unlock(l)` at l_6. After that, T_2 wakes up from `wait(c,l)`, re-acquires lock l and then continues. This thread interleaving is shown by Trace 1 in the middle, where the $wait(c, l)$ operation splits into three distinct events $rel(l); wait(c); acq(l)$.

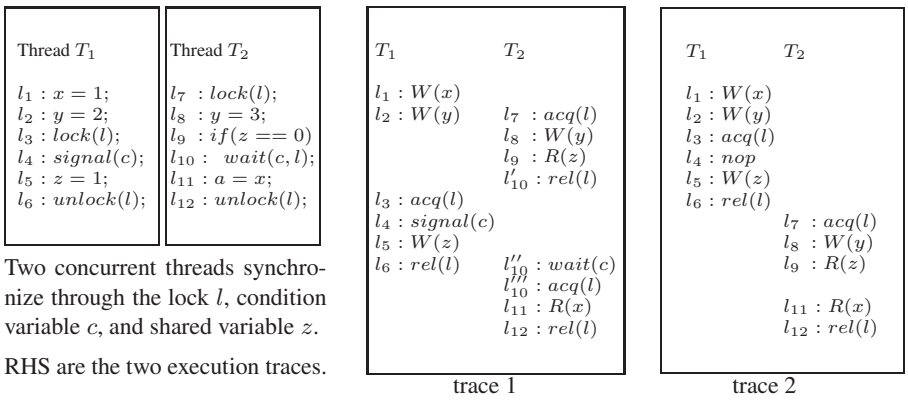

Fig. 7. UCG would report a bogus race on x on trace 2; CP would miss the real race on y.

The use of variable z is crucial in ensuring that `wait(c,l)` is executed only when T_1 has not yet executed `signal(c)`. If T_1 enters the critical section ($l_3 \dots l_6$) and executes `signal(c)` first, since T_2 is not waiting, the signal sent by T_1 would be lost. In this case, T_2 must skip `wait(c,l)` based on checking z's value. Otherwise, executing `wait(c,l)` would cause T_2 to hang. This second interleaving is shown in Trace 2.

Also note that the two input traces provide only limited observability. That is, we know l_8 is a write to y in trace 1 but not the value written (or the right-hand-side expression). We know that both l'_{10} and l_{11} happen after l_9, but do not know that l_{10} is guarded by l_9 but l_{11} is not. These are reasonable assumptions in many real-world applications, where getting more detailed program information is expensive or impossible, e.g. when the execution trace logs are generated during production runs on the client site.

There are two potential data-races on variables x and y, respectively. Here a data-race refers to the simultaneous accesses of the same memory location by two concurrent threads, where at least one access is a write. These potential data-races can be identified by a standard lockset analysis, because the thread locations $(1_1, l_{11})$ and (l_2, l_8) are not protected by the same lock.

When the input is Trace 1, both UCG and CP would correctly decide that the data-race on y is real and the data-race on x is bogus, due to the signal-to-wait edge $l_4 \rightarrow l''_{10}$. However, when the input is Trace 2, UCG would incorrectly classify (l_1, l_{11}) as being reachable (bogus) and CP would incorrectly classify (l_2, l_8) as unreachable (missed).

The imprecision of UCG for Trace 2 is due to the fact that UCG considers only standard synchronization operations while ignoring *ad hoc* synchronization events such as $W(z)$ and $R(z)$. Since l_1 and l_{11} are not protected by a common lock, and not ordered by signal/wait, UCG assumes that they can be executed simultaneously.

The imprecision of CP for Trace 2 is due to its dropping of valid interleavings too aggressively. That is, whenever two critical sections share conflicting data accesses – such as $W(z)$ and $R(z)$ in Trace 2 – it imposes a *causally-precedes* relation, effectively adding $l_6 <_{CP} l_7$. Although this removes the bogus data-race on x, it also removes the real data-race on y because it forbids any interleaving in which the order of the two critical section is swapped. Indeed, the constraint $l_5 <_{CP} l_6$ is too strong, because according to the semantics of the locks, l_6 can actually happen before l_5.

Our method, in contrast, can correctly classify both cases in Fig. 7. It improves over UCG by adding the new Rule 2. The addition of write/read consistency, in particular, is responsible for the correct classification of the data-race on x. It also improves over CP by applying the goal (EC) directed polyedge slicing (Section 3.2), goal directed polyedge resolution (Section 3.3), before resorting to the use of CP heuristic (Section 4.2). Therefore, it may be able to find a feasible solution without using the (often more restrictive) CP relation at all. Indeed, this is the reason why we can still detect the data-race on y whereas the original CP method cannot.

7 Experiments

We have implemented the new procedure in an offline data-race prediction tool based on the LLVM platform. The tool is capable of analyzing traces generated by arbitrary concurrent C/C++ programs written using the POSIX threads. It is worth pointing out that other instrumentation tools, such as the PIN binary instrumentation tool, may also be used to generate the input traces.

We have conducted experiments on two sets of benchmarks. The first set is a collection of traces from small multithreaded programs in the recent literature (e.g. [10, 24]). The main purpose is to confirm that our implementation is indeed more accurate than existing methods. For example, the benchmark *cp7* comes from an example used in [24] to illustrate why there is no *CP-predictable* data-race and how CP can avoid false alarms. Our tool discovers that it actually has a real and predictable data-race under the CDSR predictive model. We were surprised initially by the data-race reported by our tool, but confirmed subsequently that this was indeed a real data-race, although it could not be detected by using the CP method.

The second set of benchmarks is a collection of traces from various open-source projects with known bugs, which we use to demonstrate the effectiveness of our method. The set contains bug samples extracted from applications such as Mozilla and MySQL, as well as two open-source projects (`aget-0.4` and `pfscan-1.0`) downloaded from the *sourceforge.net* website. Some of the extracted examples are kindly provided by the authors of [32], whereas others are used in some prior publications (e.g. [15]). All benchmarks are accompanied by test cases to facilitate concrete execution.

We compare the performance of three methods: UCG [10], CP [24], and Poly. All methods were implemented in the same data-race prediction tool to facilitate a fair

comparison. (Recall that the original CP algorithm [24] was for Java.) Our experiments were conducted on a workstation with 2.8 GHz Pentium D processor and 2GB memory.

Table 1. Comparing our new method with UCG [10] and CP [24] for predicting data-races.

Test Program						Partial Trace				Num. Data-races				Time (sec/race)		
name	LOC	thr	lks	cvs	vars	evs	lkevs	coevs	rwevs (r / w)	r-p	r-ucg	r-poly	r-cp	t-ucg	t-new	t-cp
cp1	79	3	1	0	2	21	4	0	6 (2/4)	5	1	0	0	0.021	0.029	0.002
cp2b	113	3	1	0	4	27	4	0	12 (5/7)	11	3	1	1	0.020	0.043	0.005
cp2	113	3	1	0	4	27	4	0	12 (5/7)	11	3	1	1	0.030	0.044	0.002
cp4	67	3	1	0	1	18	4	0	3 (0/3)	3	1	1	1	0.015	0.019	0.002
cp5	168	4	3	0	3	45	14	0	14 (4/10)	12	1	0	0	0.042	0.049	0.006
cp5b	144	4	3	0	3	42	14	0	11 (2/9)	9	1	1	0	0.029	0.035	0.003
cp6	255	5	6	0	5	71	24	0	23 (8/15)	19	1	0	0	0.043	0.059	0.005
cp7	277	5	7	0	6	80	30	0	25 (6/19)	20	1	1	0	0.073	0.080	0.008
cp8	119	3	3	0	2	33	12	0	8 (2/6)	7	1	1	1	0.030	0.035	0.004
vt1	53	2	1	0	1	13	4	0	2 (1/1)	1	1	1	1	0.007	0.009	0.002
vt2	59	2	1	0	2	15	4	0	4 (2/2)	1	1	0	0	0.015	0.021	0.004
vt2b	60	2	1	0	2	15	4	0	4 (2/2)	1	1	1	0	0.008	0.012	0.003
vt2c	56	2	1	0	1	14	4	0	3 (1/2)	1	1	1	0	0.015	0.020	0.005
vt2d	56	2	1	0	1	14	4	0	3 (1/2)	1	1	1	0	0.017	0.023	0.005
vtex4	72	2	1	1	2	19	4	1	6 (2/4)	3	1	0	0	0.016	0.020	0.003
vtex5	73	2	1	1	2	19	4	1	6 (2/4)	3	1	1	0	0.032	0.011	0.005
vtex6	92	3	1	1	2	26	6	2	6 (2/4)	5	0	0	0	0.027	0.008	0.007
Total											**20**	**11**	**5**			
UpdateTimer	266	2	1	0	4	50	32	0	11 (6/5)	6	1	1	1	0.026	0.034	0.006
SeekToItem	246	3	1	0	3	49	28	0	10 (6/4)	7	1	1	1	0.021	0.031	0.004
TimerThread	240	2	2	1	4	51	30	2	10 (5/5)	0	0	0	0	0.020	0.024	0.004
NodeState	176	2	1	0	3	32	20	0	5 (2/3)	1	1	1	1	0.047	0.052	0.004
MysqlLog	181	2	1	0	2	32	20	0	5 (2/3)	2	2	2	2	0.033	0.039	0.004
Loadscript	227	3	2	0	3	50	30	0	8 (4/4)	1	0	0	0	0.048	0.055	0.006
FileTransport	184	2	1	0	2	33	20	0	6 (3/3)	2	2	2	2	0.039	0.048	0.009
CreateThread	178	2	1	0	4	32	16	0	9 (5/4)	3	2	1	1	0.041	0.046	0.003
thrift-1606	44	2	0	0	1	9	0	0	3 (1/2)	2	1	1	1	0.002	0.003	0.000
apache-21285	484	3	1	0	8	69	8	0	50 (35/15)	35	8	2	2	0.019	0.043	0.012
apache-25520	82	3	0	0	2	16	0	0	6 (3/3)	5	3	1	1	0.019	0.033	0.001
maple-cir-list	1393	3	2	0	38	208	56	0	140 (60/80)	43	2	1	1	0.137	0.243	0.047
mysql2011	231	3	2	0	10	53	4	0	37 (18/19)	33	3	3	3	0.011	0.021	0.004
pfscan-1.0-r3	4431	4	4	3	324	791	66	29	678 (288/390)	117	6	3	0	0.007	0.158	0.064
aget-0.4.comb	9277	3	1	0	785	1294	40	0	1243(1089/154)	513	405	11	10	0.132	0.357	0.042
Total											**430**	**30**	**26**			

Table 1 shows the results. In this table, the first six columns show the statistics of the test programs, including the name, the number of lines of code, the number of threads, lock variables, condition variables, and shared variables. The next four columns show the statistics of the input trace, including the number of events (evs), the number of lock events (lkevs), the number of condition variable events (coevs), and the number of read/write events (rwevs). We also provide a break-down of the read and write events.

The next four columns show the statistics of the data-race prediction algorithm. Column *rp* shows the number of *potential data-races*. These are the data-races found by our implementation of a standard lockset based analysis. Column *r-ucg* shows the number of data-races reported by the UCG algorithm. Both lockset and UCG may re-

port spurious data-races but miss no real data-races that are *predictable* from the given traces. Column *r-poly* shows the number of real data-races found by our new algorithm. Column *r-cp* shows the number of data-races found by CP. The last three columns show the runtime performance, which is the average time (seconds) per feasibility check.

In the first set of examples, our new method found more real data-races than CP (11 versus 5), while avoiding all false alarms generated by UCG (a total of 9). In the second set of examples, our new method also found more real data-races than CP (30 versus 26), while avoiding all false alarms generated by UCG (a total of 406).

In terms of runtime performance, our method on average takes longer time than both UCG and CP. This is as expected due to its more involved causality analysis. The benefit of the extra effort is that our method always returns the precise result, without false alarms and missed bugs. Furthermore, the runtime numbers of all three methods are very small – practically negligible – for the targeted application. Therefore, we conclude that our for offline applications, our new method is competitive in that it provides a much more *in-depth* analysis of the concurrent execution traces.

8 Related Work

We have reviewed existing trace-based predictive analysis methods including the lock-set analysis and the lock causality based methods for deciding control state reachability (LAH [9, 11], LCG [8], UCG [10], and CP [24]). Our new procedure is more generally applicable and accurate than these existing methods.

Beyond offline bug prediction, there is a large body of work on *online bug detection*, e.g. for detecting data-races [6, 1, 14] and atomicity violations [3, 30]. The distinction between these two types of methods is fairly large. Typically, online methods focus primarily on reducing the runtime overhead, often at the expense of losing precision (false alarms) or decreasing coverage (missed bugs). Whereas offline analysis methods focus primarily on improving the precision and coverage. If spending a few extra seconds or even minutes on analyzing a trace log can lead to the discovery of a few more real bugs, it would considered worthwhile. Our polygraph based method is by far the most accurate method for offline analysis of execution traces with limited observability.

A closely related work is PENELOPE [25, 5], which can predict and subsequently confirm atomicity violations and null pointer violations. A core predictive analysis in PENELOPE is the LAH [9] analysis. Since our method improves over LAH, in principle, it may also be used to enhance the analysis in PENELOPE. In addition, PENELOPE confirms the predicted buggy interleavings by trying to re-execute them. This approach works well when it is possible to re-execute in the original environment. Our new method, in contrast, is better suited for applications where re-running the program is impossible, e.g. when the traces are generated on the client site.

Another related work is jPredictor [2], which requires the analysis of a complete execution trace consisting of all global and local instructions involved in the execution. A similar limitation exists in the SAT/SMT based predictive methods [27, 29, 12, 21, 23, 17, 22, 28], which require the program source code in conjunction to a trace to construct the prediction model. Although in general, these methods are more powerful, the detailed program code information may not be available in in many application settings.

Our new method, in contrast, relies on only a simple trace, which is better suited for applications where detailed information about the program is not available.

9 Conclusions

We have presented a new polygraph based procedure for deciding control state reachability properties in simple execution traces generated by multithreaded programs. We have customized our core analysis procedure for an offline data-race detection tool. In this context, our method is provably more accurate than the existing ones, including the more recent causally-precedes method and the universal causality graph method. We have implemented and evaluated our method through experiments. The results confirm that our procedure is indeed more accurate than the existing ones.

Acknowledgments

We thank the anonymous reviewers for their feedback. This work is supported in part by the NSF grant CCF-1149454 and the ONR grant N00014-13-1-0527.

References

1. M. D. Bond, K. E. Coons, and K. S. McKinley. PACER: proportional detection of data races. In *Programming Language Design and Implementation*, pages 255–268, 2010.
2. F. Chen, T. Serbanuta, and G. Rosu. jPredictor: a predictive runtime analysis tool for java. In *International Conference on Software Engineering*, pages 221–230, 2008.
3. A. Farzan and P. Madhusudan. Monitoring atomicity in concurrent programs. In *International Conference on Computer Aided Verification*, pages 52–65, 2008.
4. A. Farzan and P. Madhusudan. Meta-analysis for atomicity violations under nested locking. In *International Conference on Computer Aided Verification*, pages 248–262, 2009.
5. A. Farzan, P. Madhusudan, N. Razavi, and F. Sorrentino. Predicting null-pointer dereferences in concurrent programs. In *Foundations of Software Engineering*, page 47, 2012.
6. C. Flanagan and S. N. Freund. FastTrack: efficient and precise dynamic race detection. *Commun. ACM*, 53(11):93–101, 2010.
7. C. Flanagan, S. N. Freund, and J. Yi. Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. In *Programming Language Design and Implementation*, pages 293–303, 2008.
8. V. Kahlon. Boundedness vs. unboundedness of lock chains: Characterizing decidability of pairwise cfl-reachability for threads communicating via locks. In *International Symposium on Logic in Computer Science*, pages 27–36, 2009.
9. V. Kahlon, F. Ivancic, and A. Gupta. Reasoning about threads communicating via locks. In *International Conference on Computer Aided Verification*, pages 505–518, 2005.
10. V. Kahlon and C. Wang. Universal Causality Graphs: A precise happens-before model for detecting bugs in concurrent programs. In *International Conference on Computer Aided Verification*, pages 434–449, 2010.
11. V. Kahlon and C. Wang. Lock removal for concurrent trace programs. In *International Conference on Computer Aided Verification*, pages 227–242, 2012.

12. S. Kundu, M. K. Ganai, and C. Wang. CONTESSA: Concurrency testing augmented with symbolic analysis. In *International Conference on Computer Aided Verification*, pages 127–131, 2010.
13. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.
14. D. Li, W. Srisa-an, and M. B. Dwyer. SOS: saving time in dynamic race detection with stationary analysis. In *ACM Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 35–50, 2011.
15. S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: detecting atomicity violations via access interleaving invariants. In *Architectural Support for Programming Languages and Operating Systems*, pages 37–48, 2006.
16. C. H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, 1979.
17. M. Said, C. Wang, Z. Yang, and K. Sakallah. Generating data race witnesses by an SMT-based analysis. In *NASA Formal Methods*, pages 313–327, 2011.
18. 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.
19. K. Sen, G. Rosu, and G. Agha. Runtime safety analysis of multithreaded programs. In *Foundations of Software Engineering*, pages 337–346, 2003.
20. 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, 2005.
21. A. Sinha and S. Malik. Using concurrency to check concurrency: Checking serializability in software transactional memory. In *Parallel and Distributed Processing Symposium*, 2010.
22. A. Sinha, S. Malik, C. Wang, and A. Gupta. Predictive analysis for detecting serializability violations through trace segmentation. In *Formal Methods and Models for Codesign*, pages 99–108, 2011.
23. N. Sinha and C. Wang. On interference abstractions. In *ACM Symposium on Principles of Programming Languages*, pages 423–434, 2011.
24. Y. Smaragdakis, J. Evans, C. Sadowski, J. Yi, and C. Flanagan. Sound predictive race detection in polynomial time. In *ACM Symposium on Principles of Programming Languages*, pages 387–400, 2012.
25. F. Sorrentino, A. Farzan, and P. Madhusudan. PENELOPE: weaving threads to expose atomicity violations. In *Foundations of Software Engineering*, pages 37–46, 2010.
26. C. von Praun and T. R. Gross. Object race detection. In *ACM Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 70–82, 2001.
27. C. Wang, S. Chaudhuri, A. Gupta, and Y. Yang. Symbolic pruning of concurrent program executions. In *Foundations of Software Engineering*, pages 23–32, 2009.
28. C. Wang and M. Ganai. Predicting concurrency failures in generalized traces of x86 executables. In *International Conference on Runtime Verification*, Sept. 2011.
29. C. Wang, R. Limaye, M. Ganai, and A. Gupta. Trace-based symbolic analysis for atomicity violations. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 328–342, 2010.
30. C. Wang, M. Said, and A. Gupta. Coverage guided systematic concurrency testing. In *International Conference on Software Engineering*, pages 221–230, 2011.
31. L. Wang and S. D. Stoller. Runtime analysis of atomicity for multithreaded programs. *IEEE Trans. Software Eng.*, 32(2):93–110, 2006.
32. J. Yu and S. Narayanasamy. A case for an interleaving constrained shared-memory multiprocessor. In *International Symposium on Computer Architecture*, pages 325–336, 2009.