Efficient Predictive Analysis for Detecting Nondeterminism in Multi-Threaded Programs

Arnab Sinha, Sharad Malik
Princeton University
{sinha,sharad}@princeton.edu

Aarti Gupta
NEC Laboratories America
agupta@nec-labs.com

***Abstract*—*Determinism* is often a desired property in multi-threaded programs. A multi-threaded program is said to be deterministic if for a given input, different thread interleavings result in the same system state in the execution of the program. This, in turn, requires that different interleavings preserve the values read by each read operation. A related, but less strict condition is for the program to be *race-free*. A deterministic program is race-free but the converse may not be true. There is much work done in the *static analysis* of programs to detect races and nondeterminism. However, this can be expensive and may not complete for large programs in reasonable time. In contrast to static analysis, *predictive analysis* techniques take a given program trace and explore other possible interleavings that may violate a given property – in this case the property of interest is determinism. Predictive analysis can be sound, but is not complete as it is limited to a specific set of program runs. Nonetheless, it is of interest as it offers greater scalability than static analysis. This work presents a predictive analysis method for detecting nondeterminism in multi-threaded programs. The key feature of our graph-based method is that it searches over a reduced set of sufficient interleavings. On the average, the number of graphs analyzed is almost equal to the number of potential cases of non-determinism, thereby ensuring that it is efficient. We demonstrate its application on some benchmark Java and C/C++ programs.**

I. INTRODUCTION

Writing correct and efficient multi-threaded programs is widely accepted as a challenging task. The wide range of possible concurrency errors makes it inherently harder than writing sequential programs [15], [25], [27]. Given the same input, the different runs of a multi-threaded program may produce different outputs because the threads interleave in different ways. This makes it hard to replicate and debug errors through traditional testing methods. These errors are referred to as "Heisenbugs" [2]. The potential *nondeterminism* of multi-threaded programs lies at the core of these Heisenbugs. For this and other reasons, *determinism* is often a desired property in multi-threaded programs. A multi-threaded program is said to be deterministic if for a given input, different thread interleavings result in the same system state in the execution of the program. It is important to consider when the system state is observed. If it is observed only at the end of the program execution, then individual read events may not need to read the same value across different interleavings. However, if the system state is continuously observed, then each read event must read the same value in all possible interleavings. We consider this case. Further, for ease of analysis we consider the stricter condition that each read event reads the value from the *same* write event in all interleavings. This restriction is consistent with other work in predictive analysis [8], [33], and can be supplemented with program analysis to consider specific values rather than specific events, if desired.

A related but less strict condition is a *datarace*. A pair of shared memory accesses are said to be *conflicting* if they are performed by different threads and at least one of them is a write. Also, the events are *unsynchronized* if the threads do not use an explicit mechanism such as locks to prevent the accesses from being simultaneous. A datarace is defined as two conflicting and unsynchronized data accesses. *A deterministic program is race-free but the converse may not be true*. The following example in Fig. 1 illustrates this further.

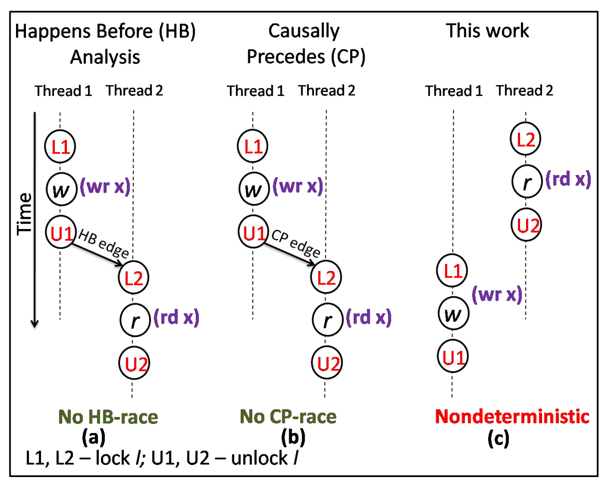

Figure 1. A deterministic program is race-free but the converse may not be true. ('Causally precedes' is defined in [34].)

Consider the example in Fig. 1. In this example, there is a pair of conflicting shared memory accesses, each under the lock-scope of the same lock variable l. Let events L1 and L2 be '*acquire lock*' events on l. Similarly, let events U1 and U2 represent '*release lock*' events on l. In (a), we show a standard Happens Before (HB) analysis for lock operations. The two events U1 and L2 are ordered by HB, as indicated by the (U1, L2) edge and hence there is no race. Next, in (b), we consider the *causally precedes* (CP) analysis proposed by Smaragdakis et al. [34]. Due to the presence of conflicting accesses (w and r) within the lock-scopes, U1 causally precedes L2 introducing the CP edge from U1 to L2. Hence, there is no *CP-race*. However, observe that in another interleaving (c), where the lock-scopes swap order, the following different order is possible in an interleaving: U2 happens before L1. Thus, while the program is race free, it is nondeterministic because the read event (`rd` x) reads from a different write event in the interleaving (c) compared to the interleaving in (a) and (b).

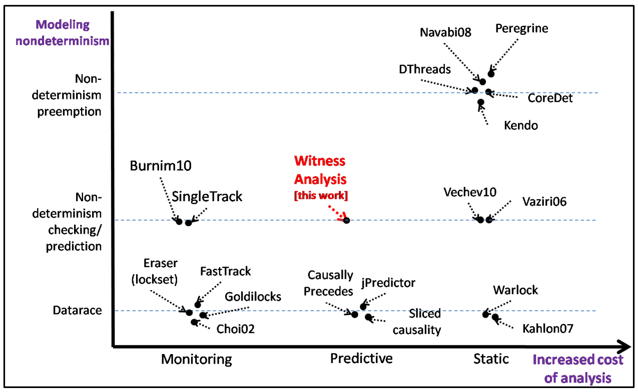

Figure 2. Classification of race and nondeterminism detection techniques based on cost of analysis: Burnim10 [6], SingleTrack [31], Eraser [32], FastTrack [13], GoldiLocks [11], Sliced Causality [7], jPredictor [8], Causally precedes [34], CoreDet [3], Kendo [28], DThreads [24], Navabi08 [26], Peregrine [10], Vaziri06 [36], Warlock [35], Kahlon07 [20], Choi02 [9], Vechev10 [37]

There is much work done in *static analysis* of programs to detect races and nondeterminism [35], [20], [9], [3], [28], [24], [26], [10], [36] as shown in Fig. 2. Among these, *deterministic multi-threading* (DMT) has attracted a lot of interest recently [24], [10]. DMT deterministically schedules the threads such that the values read by the read operations are preserved. The static analyses for detection or finding schedules can be expensive and may not complete for large programs within reasonable time.

The other end of the spectrum is *monitoring*-based solutions [32], [13], [11]. Although monitoring-based solutions are scalable and sound, the analysis is based only on the runs that are actually executed. In contrast, *predictive analysis* techniques take a given program trace and explore other possible interleavings that may violate a given property [34], [8], [7]. This helps to enhance coverage of a given test input to a larger set of thread interleavings. Predictive analysis can be sound but it is not complete as it may not cover the entire program.

In this work, we adopt a predictive analysis technique for detecting nondeterminism. This provides an effective trade-off between cost and coverage. Our technique is based on the partial order permitted by a trace combined with the reasoning for locks. This technique is fast because it searches a reduced set of sufficient interleavings. Potential cases of non-determinism are checked by constructing a causality graph from the thread events and confirming that this is acyclic. We demonstrate its application on some benchmark Java and C++ programs. Our results show that the average number of graphs analyzed per benchmark is one per potential case of nondeterminism.

This work makes the following contributions:

- It presents a sound and complete[1] predictive analysis technique for checking determinism of multi-threaded programs. It reports only feasible violations of nondeterminism and thus avoids false bugs that would require additional test execution after the analysis.
- The proposed technique requires search over a reduced set of sufficient interleavings and hence is fast.
- The technique has been implemented and experimental results on C/C++ and Java benchmark programs are very promising.

II. Preliminaries

We consider a multi-threaded program consisting of a set of *threads* $T_1, T_2 \ldots, T_k$ and a set of *shared variables*. Let $\{1, \ldots, k\}$ be the set of thread indices. The remaining aspects of the program, including the control flow and the expression syntax, are intentionally left unspecified for generality.

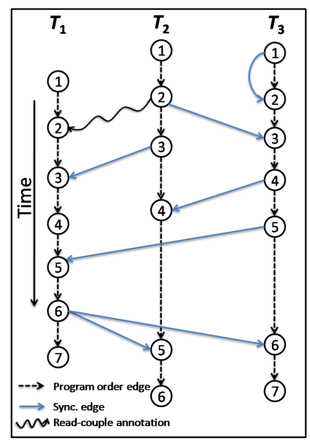

Figure 3. The partial order graph with vertices representing events and the dashed and solid edges are program order and sync. edges respectively. The read-couple annotations are indicated by the squiggly arrows.

Program Trace Model: An execution trace $\rho = e_1$, e_2,$\ldots e_n$ is a sequence of events, e_i, $i \in \{1, \ldots, n\}$, each of which is an instance of a *visible* operation during the execution of the program. The visible operations are: read/write accesses to shared variables and synchronization operations such as wait, notify, notifyall, lock acquire/release and thread fork/join. An event is represented as a 5-tuple $(tid, eid, type, var, child)$, where tid is the thread index ($tid \in \{1, \ldots, k\}$), eid is the event index (that starts from 1, and increases sequentially within a thread), $type$ is the event type, var is either a shared variable (in read/write operations) or a synchronization object, $child$ is the child thread index (in thread create/join). The event type is one of $\{read$, $write$, $fork$, $join$, $acquire$, $release$, $wait$, $notify$, $notifyall\}$.

An execution trace ρ is the observed interleaving of events across the threads and provides a total order on these

[1]over all interleavings of events in the given trace, not over the entire program

events. We derive the required partial order for this trace by retaining only the set of must-happen-before constraints as described below.

Partial Order Graph: Let $G(V,E)$ be a partial order graph such that $V(G)$ is the set of vertices, each of which represents an event in the trace (we use vertices and events interchangeably when the context is clear). Fig. 3 is an example partial order graph with three threads. The number inside each vertex is the eid within the thread. A directed edge (a,b) in $E(G)$ (the set of edges) is either a program order edge, or a synchronization (sync.) edge[2]. Program order edges are indicated by dotted arrows and sync. edges by solid arrows in Fig. 3. An edge in $E(G)$ is referred to as a *partial order edge*.

We note that locks are not added as sync. edges in $E(G)$. The mutual exclusion due to locks is considered separately by our analysis. We also give special consideration to write-read pairings. If event b reads the value written by event a, then the pair (a,b) is defined as a *read-couple*. A read-couple is indicated by a squiggly arrow annotation in the partial order graph G. Note that this is also not included in the edge set $E(G)$. In a different interleaving τ, if b reads from a different event c, we say that the read-couple for b, and the read b in ρ is *broken* in τ.

Locked Scope: A locked scope, denoted as $[e_i \dots e_j]_l$, is defined as the sequence of events $e_i \dots e_j$ after an '`acquire lock` l' event and before a '`release lock` l' event, where l is a lock-variable. Note that the sequence of events $e_i \dots e_j$ and lock acquire/release events belong to the same thread.

III. Predictive Analysis of Nondeterminism

We assume that the shared variables are implicitly written (or initialized) at the beginning of the execution. Similarly, they are all implicitly read at the end of the program execution. Given the same inputs, if a read instruction of a shared variable reads the value from the same write operation in all interleavings, it is referred to as a *view-preserving* read. Otherwise, the read is *non-view-preserving*. This is related to the well-known notion of view equivalence in database transactions [29].

Definition 1: **[Program Nondeterminism]** We define a multi-threaded program to be nondeterministic iff there exists at least one non-view-preserving read.

Writer, Readers and Challengers: In the given trace, there can be several read operations reading the value written by a single write operation, w. w is referred to as the *writer*. Any read event that reads the value written by w is denoted as *reader* of w. Let $R(w)$ be the set of readers of w. Any write operation c, other than w that writes the same shared variable is denoted as a *challenger* of w. It is named so since it challenges the set of read-couples induced by w (i.e. $\{(w,r) \text{ where, } r \in R(w)\}$) as in an alternate interleaving r may read from c instead of w, thus breaking the read couple (w,r). Let, $C(w)$ be the set of all challengers of writer w.

Problem Formulation: We aim to detect nondeterminism over alternate interleavings of events of a given trace ρ. Thus, we address the following problem: *given a trace ρ and a read-couple (w,r) in ρ, is there a challenger c such that it breaks (w,r) in another interleaving τ?*

For a pair of events e_1 and e_2 and an interleaving, let $e_1 \mapsto e_2$ represent "*e_1 precedes e_2 in the interleaving*". For a given triplet (w,r,c) and a partial order graph G, where $r \in R(w)$ and $c \in C(w)$, the read-couple is broken in an interleaving τ, when any of the following orders is present in τ: (1) $c \mapsto r \mapsto w$, or (2) $w \mapsto c \mapsto r$, or (3) $c \mapsto r$ and w does not occur in τ. In each case, r does not read from w in τ. We refer to these orders as *witnesses of nondeterminism* and the interleaving containing a witness as a *witness interleaving*. In cases (1) and (2), w, r and c are the events of the witness and in case (3), c and r are the events of the witness. A triplet is said to be nondeterministic if it can provide a witness of nondeterminism.

Central Idea: There are two phases in our analysis for each witness. For a certain witness ω to exist in an interleaving τ, τ must satisfy the orderings between the members of ω in addition to the HB constraints imposed by program-order, synchronization and possibly between locked scopes. Let $G'(\omega)$ be the graph after incorporating all the mentioned constraints to G in the form of ordering edges. $G'(\omega)$ cannot contain a cycle since τ must be a total order of events satisfying the ordering constraints imposed by $G'(\omega)$. Thus, in the first phase of our analysis, we check for a cycle in $G'(\omega)$. Presence of cycle in $G'(\omega)$ entails the witness to be infeasible (*necessary condition* for feasibility of witness). (This phase is similar to a Universal Causality Graph (UCG)-based analysis [23]. We provide a detailed comparison later.) However, absence of a cycle in $G'(\omega)$ does not guarantee feasibility of witness. This is because for each pair of mutually exclusive locked scopes LS_1 and LS_2, either LS_1 HB LS_2 or LS_2 HB LS_1. Since this holds for each pair of mutually exclusive locked scopes, we need to consider all possible combinations of such HB constraints. For d such pairs, there will be 2^d combinations. These choices need to be explored by augmenting G' with each of these 2^d combinations of HB constraints. In the second phase of our analysis, we construct all such possible graphs obtained by augmenting $G'(\omega)$. Let $G''(\omega)$ be one such graph. The witness is infeasible if and only if all 2^d $G''(\omega)$ graphs contain cycles (*sufficient condition* for feasibility of witness). We now describe these two phases in detail below.

A. Necessary Condition for Witness: Witness Order Graph

Let ω be a witness in an interleaving τ. We consider ordering constraints imposed by ω on τ. Note that G already

[2] HB edges between $fork$ event in parent thread and first event in child thread, between $wait$ and $notify$ events, and between last event in child thread and $join$ event in parent thread are sync. edges.

contains program order and synchronization constraints that τ must obey. We now augment G to $G'(\omega)$ to include additional ordering constraints imposed by the witness ω. $G'(\omega)$ is referred to as the *witness order graph*. The orders imposed by ω are reflected by adding additional edges to $G'(\omega)$ denoted as *witness order edges*. These edges need to consider the mutual exclusion property of locked scopes of the witness events.

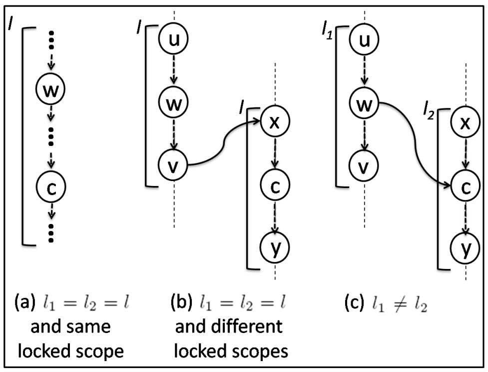

Figure 4. Additional ordering constraints imposed by locked scopes and witness order edges in witness order graph $G'(\omega)$.

We construct $G'(\omega)$ specific to witness ω as follows. (Henceforth, we refer to $G'(\omega)$ as G' when ω is clear from the context.) Without loss of generality consider ω to be of the type $w \mapsto c \mapsto r$. Consider the ordering constraints in τ due to $w \mapsto c$. Let $[u \dots w \dots v]_{l_1}$ be a locked scope of w. w may be in multiple nested locked scopes. Note that event u may be the same as w, and/or w may be the same as event v. Note also that l_1 may be nil, i.e. w is not in a locked scope. Similarly, let $[x \dots c \dots y]_{l_2}$ be a locked scope of c. Again event x may be the same as c, and/or c may be the same as y, and l_2 may be nil, i.e. c is not in a locked scope.

We consider the ordering constraint $w \mapsto c$ imposed by the witness in τ, under the following three cases of locked scopes.

- **Case 1:** $l_1 = l_2 = l$ and w and c are in the same locked scope $[\dots w \dots c \dots]_l$. The program order edges already capture the ordering constraint between w and c. No additional constraint is added (Fig. 4(a)).
- **Case 2:** $l_1 = l_2 = l$ and w and c are *not* in the same locked scope. The $w \mapsto c$ order imposes an additional order that v precedes x in τ due to the mutual exclusion imposed by l. We reflect this by adding the witness order edge (v, x) to G' (Fig. 4(b)).
- **Case 3:** $l_1 \neq l_2$. There is no mutual exclusion constraint. However we need to add the witness order edge (w, c) to G', as shown in Fig. 4(c).

Note that the above needs to be done for each locked scope pair l_1 and l_2 that w and c are in.

Similarly, we need to add witness order edges for $c \mapsto r$ and $w \mapsto r$, based on locked scopes for w, c, and r. Note that the edge due to $w \mapsto r$ does not transitively follow from the edges due to $w \mapsto c$ and $c \mapsto r$ due to the locked scopes.

For each observed read-couple (a, b) in ρ besides (w, r) in ω, we add a read-couple edge (a, b) to $E(G')$. In addition, the *induced edges* [23] are added to $E(G')$ as described below. For a pair of locked scopes guarded by same lock-variable ($[u, \dots, v]_l$ and $[x, \dots, y]_l$, say), we add an induced edge (v, x) if there is path from u to y in G'[3]. However, if neither v precedes x and nor y precedes u in G' i.e. the locked-scopes are unordered, then the locked-scopes are said to have a *choice* between edges (v, x) and (y, u) in terms of the HB relation between them. This choice will be dealt with later.

This is illustrated in Fig. 5. The partial order graph G in Fig. 5(b) corresponds to the multi-threaded program trace in Fig. 5(a). Further, Fig. 5(c) shows the witness order graph for the same program trace and witness $c \mapsto r \mapsto w$. The edge (v, x) is induced by (w_2, r_2) and the presence of locked scopes $[u, \dots, v]_l$ and $[x, \dots, y]_l$. Note that insertion of one induced edge can trigger insertion of another induced edge if the locked-scopes are nested or overlapping.

G' now contains the following four kinds of ordering constraints due to G (program order edges + sync. edges), witness order edges (including locked scope analysis), read-couple edges except (w, r), and the induced edges due to mutual exclusion of locked scopes. Locked scope analysis enforces mutual exclusion constraint. However, when combined with the ordering enforced by a specific witness, the mutual exclusion constraint can lead to an ordering constraint which can be added to the partial order ordering constraints [23].

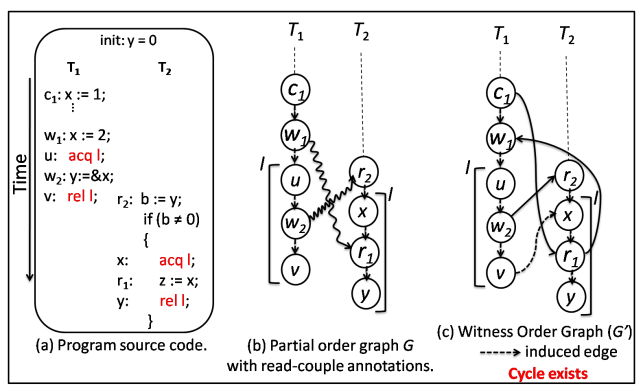

Figure 5. The partial order graph G and the witness order graph $G'(\omega)$, where ω is $(c \mapsto r \mapsto w)$ for the example program source code in (a).

In Fig. 5(c), G' has a cycle $(r_1 \rightarrow w_1 \rightarrow u \rightarrow v \rightarrow x \rightarrow r_1)$. Since this cycle represents orderings corresponding to the edges in G', at least one of these orders is not possible. Specifically, in this case, the (w_2, r_2) read-couple will be broken in τ. Therefore, the read for r_2 in τ may result in

[3]Presence of a path from u to y in G' implies that $[u, \dots, v]_l$ must be entirely executed before starting the execution of $[x, \dots, y]_l$.

a different value from the read in the original trace ρ. This may alter the program flow so that the event r_1 may not even happen in τ. In this case the witness is said to be infeasible as τ may not contain r_1.

Let (w', r') be a read-couple in ρ that is broken in τ. Let x be an event in witness ω. The witness ω is *infeasible* if there is a path from r' to x in G. Intuitively, for ω to be feasible, all the views must be preserved until the events in ω in the interleaving τ. (If (w', r') is broken in τ then r' is not view preserving.) Otherwise ω is deemed infeasible in G'. The following theorem provides the necessary condition for feasibility.

Theorem 1: [WITNESS ORDER GRAPH THEOREM] A witness is infeasible if there is a cycle in G'.

A proof sketch is provided in the appendix. The reverse direction (infeasibility⇒cycle) is not true. This has to do with the ordering choice between unordered locked scopes and is considered next.

Consider a pair of locked scopes $[a_1, \ldots, b_1]_l$ and $[a_2, \ldots, b_2]_l$ in different threads guarded by the same lock variable l, such that there does not exist a path between a_1 and b_2 or between a_2 and b_1 in G'. In this case the locked scopes are defined to be an *unordered* pair of locked scopes. Moreover due to the mutual exclusion between the two locked scopes one must be ordered before the other. Thus, there exists a *choice* between edges (b_1, a_2) and (b_2, a_1). The edges (b_1, a_2) and (b_2, a_1) are defined as *choice edges* and the pair $\{(b_1, a_2), (b_2, a_1)\}$ is a *choice edge pair*.

Consider G' shown in Fig. 6. Let there be a witness order edge from y to x (not shown in Fig. 6 for clarity). Let $[a_1, \ldots, b_1]_{l_1}$ and $[a_2, \ldots, b_2]_{l_1}$ be an unordered pair of locked scopes guarded by variable l_1. Similarly, let $[a_3, \ldots, b_3]_{l_2}$ and $[a_4, \ldots, b_4]_{l_2}$ be an unordered pair of locked scopes guarded by variable l_2. Let e_1 and e_2 be choice edges $e_1 \in\{(b_1, a_2),(b_2, a_1)\}$ and $e_2 \in\{(b_3, a_4),(b_4, a_3)\}$. Let the edges shown in Fig. 6 represent paths in G'. *For finding a feasible witness, we need at least one combination of choice edges e_1 and e_2 such that their addition to G' leads to no cycle.* In this example, every combination of e_1 and e_2 results in a path from x to y. This combined with the witness order edge (y, x) leads to a cycle for each combination. In general, if there are d choice edge pairs then we need to check 2^d combinations in conjunction with G'. The number of combinations that actually need to be considered can be reduced as shown in the next subsection.

We would like to point out that this example also illustrates that UCG analysis [23] is incomplete in general, since it does not consider choice edges that may result in cycles with more than two threads.

B. Sufficient Condition for Witness: Choice Graph

We first define a *lock abstraction graph* denoted as $G''_a(\omega)$. (Henceforth, we refer to the lock abstraction graph as G''_a when ω is clear from the context.) All vertices within a locked scope in G' are replaced by a single meta-vertex in G''_a. Any edge originating from or terminating into the locked scope, originates from or terminates into the meta-vertex, respectively. Further, for each unordered pair of locked scopes present in G', an undirected edge connects the corresponding meta-vertices in G''_a referred to as the *abstract choice edge*. The abstract choice graph for the example shown in Fig. 6 is shown in Fig. 7(a). The vertices $m_1, \ldots, m_4$ are the meta-vertices and the undirected edges (m_1, m_2) and (m_3, m_4) represent the abstract choice edges in G''_a.

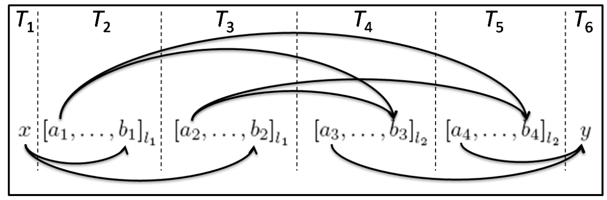

Figure 6. All combinations of choice edges e_1 and e_2 give a path from x to y, where $e_1 \in\{(b_1, a_2),(b_2, a_1)\}$ and $e_2 \in\{(b_3, a_4),(b_4, a_3)\}$

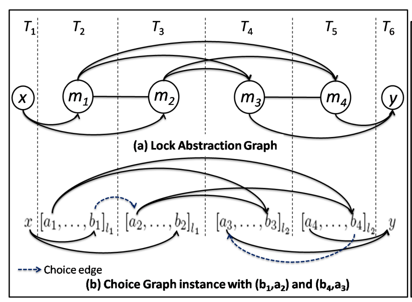

Figure 7. (a) Lock abstraction graph for the example shown in Fig 6. (b) One of the choice graphs with choice edges (b_1, a_2) and (b_4, a_3).

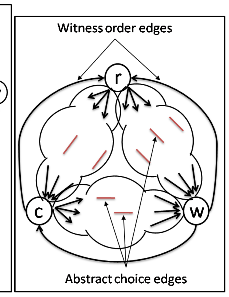

Figure 8. The undirected edges shown in $G''_a(\omega)$ are the abstract choice edges that constitute S_{choice} for witness $w \mapsto c \mapsto r$.

We compute S_{choice} as the set of choice edge pairs such that their exploration is sufficient to detect feasibility of ω. We construct S_{choice} by collecting all the abstract choice edges present in all paths from x to y in G''_a, for all x and y, where (y, x) is a witness order edge in G'. Fig. 8 illustrates this for a witness $w \mapsto c \mapsto r$. Let $|S_{choice}| = d'$. Usually $(d' << d)$. This reduction can be viewed as a form of witness-based slicing of G''_a.

Next, we define the choice graph $G''(\omega)$ as follows. (Henceforth, we refer to the choice graph as G'' when ω is clear from the context.) The vertex set $V(G'') = V(G')$. The edge set $E(G'')$ is $E(G')$ augmented with exactly one choice edge per choice edge pair in S_{choice}. Formally,

$$E(G'') = E(G') \cup \left(\bigcup_{\substack{\forall\{e_{c1}, e_{c2}\} \in S_{choice}, \\ e_c \in \{e_{c1}, e_{c2}\}}} \{e_c\} \right)$$

For instance, in the example shown in Fig 6, there are two choice edge pairs that belong to S_{choice}: $\{(b_1, a_2), (b_2, a_1)\}$ and $\{(b_3, a_4), (b_4, a_3)\}$. Each choice graph must choose exactly one edge from each pair. As there are 2^2 combinations possible, there exists four choice graphs for this example. Fig. 7(b) shows one of those choice graphs with a choice edge combination (b_1, a_2) and (b_4, a_3).

Theorem 2: [CHOICE GRAPH THEOREM] A witness is infeasible iff all the choice graphs have cycles.

A proof-sketch is provided in the appendix.

C. The Nondeterminism Checking Algorithm

We now summarize the overall algorithm. We first compute the set of possible witnesses, based on challengers for each read event in a trace. For each such witness ω, in the first phase of our analysis, we construct the witness order graph ($G'(\omega)$) and check for a cycle. The witness is infeasible if there is a cycle in $G'(\omega)$. However, if there is no cycle we proceed to the second phase of our analysis. We compute the set S_{choice}. If S_{choice} is empty, the witness is feasible. Otherwise, we construct $2^{d'}$ choice graphs, where $|S_{choice}| = d'$, and check for a cycle until we find a choice graph with no cycle. If an acyclic choice graph exists, the witness is declared feasible. If all choice graphs contain cycles, then the witness is declared infeasible. *In practice, we need to explore only a handful (mostly one) of these choice graphs to find one without a cycle.*

The complete algorithm is shown in Fig. 9. It generates all feasible witnesses of nondeterminism for a given interleaving ρ. Let, x_i, $i = 1 \ldots m$ be the shared variables in the observed trace ρ. Further, for each shared variable x_i, let L_{x_i} be the list of read-couples, i.e. $L_{x_i} = \{(w, R(w)) \mid w \text{ writes } x_i\}$.

ReportFeasibleWitnesses (interleaving ρ)
1. Construct partial order graph G from ρ
2: Visit each vertex and if it accesses shared variable x_i
   a. label vertex with locked scopes.
   b. populate L_{x_i}.
3: for each L_{x_i}, $(i = 1 \ldots m)$
4: for each write w_j in L_{x_i}
5: for each read $r_k \in R(w_j)$
6: for each write c_l in L_{x_i} such that $w_j \neq c_l$
7: Let (w_j, r_k, c_l) be the triplet
8: for each possible witness ω for (w_j, r_k, c_l)
   //Witness Order Graph Check
9: Construct $G'(\omega)$ and check for cycle in $G'(\omega)$.
10: If cycle found in $G'(\omega)$, report ω is infeasible.
11: Else construct $G''_a(\omega)$ and compute S_{choice}.
12: Report feasible witness if S_{choice} is empty.
13: Construct choice graphs until acyclic graph is found and report ω is feasible.
14: If all choice graphs are cyclic, report ω is infeasible.

Figure 9. Algorithm for reporting feasible witnesses

Optimization: Note that the partial order edges ($V(G)$) and all the induced edges due to locks and read-couple edges except (w, r) are present in all the choice graphs for a given witness ω. Therefore, we add all the read-couple edges and their induced edges to G before line 3 in Fig. 9. Next, for each witness we do the following: (1) delete the read-couple (w, r) and the appropriate induced edges corresponding to the read-couple (w, r), and, (2) insert the witness order edges and the edges induced by them to produce $G'(\omega)$. Moreover, we use vector clocks for keeping track of the causality relationships necessary for incremental addition or removal of an induced edge.

Complexity Analysis: The symbols introduced for complexity analysis are described in Fig. 10. The complexity of step 1 in procedure **ReportFeasibleWitnesses** is $O(M + N) = O(M)$ as $N \leq M$. The locked scope analysis requires two passes over the trace to label each read/write event with eid's of acquire/release lock events guarding the event. This is $O(m(p + q)L)$. Populating L_{x_i}, for $i = 1 \ldots m$ requires one pass over the trace ($O(N)$). Next we consider the complexity for a single witness. To construct G', we add the read-couple edges ($O(mp)$) and the witness edges ($O(1)$). The induced edges order the locked scopes. Therefore, the number of induced edges added is $O(L^2)$. The number of read-couples in G is $O(mp)$. Thus, $|E(G')| = O(M + mp + L^2)$. Then cycle checking in G' is $O(M + N + mp + L^2) = O(M + L^2)$ (since $N \leq M$ and $mp \leq N$). The number of witnesses is $O(mpq^2)$. Note that in our implementation, the construction of G' is done between step 2 and step 3 of procedure **ReportFeasibleWitnesses** for efficiency, with some simple book-keeping which is omitted here for brevity. Since, the number of choice edge pairs (d) is $O(L^2)$, $d' = O(L^2)$. Therefore, the number of choice graphs is $O(2^{L^2})$. Checking a cycle in a choice graph is $O(M + L^2)$. Therefore, the overall complexity: $O(M + m(p+q)L + mp + mpq^2(M+L^2) + 2^{L^2}(M + L^2)) = O(mpL + mqL + (mpq^2 + 2^{L^2})(M + L^2))$.

Symbol description	Symbol
Number of threads	k
Number of vertices in G	N
Number of edges in G	M
Number of lock events in G	L
Number of variables	m
Max. number of events per thread	n
Max. number of reads per variable	p
Max. number of writes per variable	q

Figure 10. Symbol table

IV. RESULTS

We have implemented our technique in a prototype tool. This tool is capable of logging/analyzing execution traces generated by both Java programs and multi-threaded C/C++ programs using pthreads. The program traces used are all available online [18]. The C++ benchmark is available online [16]. All the Java benchmarks are publicly available [12], [14], [17], [19], [30].

The tool logs execution traces at runtime from C++ source code instrumented using the commercial front end from Edison Design Group (EDG). For Java programs, we used execution traces logged at runtime by a modified Java Virtual Machine (JVM). For each test case, we first executed the

Table I
SUMMARY OF THE EXPERIMENTAL DATA ON THE WITNESSES OF NONDETERMINISM IN TRACES OF MULTI-THREADED PROGRAMS.

1. Categories	2. #Benchmarks	3. #Possible witnesses	Witness Order Graph Analysis 4. Witnesses with cycles in G' (infeasible) (%)	5. Witnesses with no choice edges (feasible) (%)	6. Witnesses with choice edges (%)	Choice Graph Analysis 7. Possible choice graphs per witness in column 6	8. Choice graphs explored per witness in column 6	9. Total time taken (sec)	10. Total feasible witnesses (%)	11. Total infeasible witnesses (%)	12. Avg. number of graphs analyzed per witness in column 3
Phase I sufficient	9	104178	85789 (82.35)	18389 (17.65)	0 (0)	–	–	44.3	18389 (17.65)	85789 (82.35)	1
Both phases required	16	5604552	4477107 (79.88)	943516 (16.84)	183929 (3.28)	7.53	1.03	8597	1127445 (20.11)	4477196 (79.88)	1.001

program using the default OS thread scheduling and logged the execution trace. Next we applied our algorithm to detect the feasible witnesses. We would like to highlight here that we originally implemented exploration of the combination of choice edges using an SMT solver, but the cost was prohibitive, failing to finish on several benchmarks. This motivated our current purely graph-based approach.

All our experiments were conducted on an Intel i7 machine (2.67 GHz, 3 GB memory) running Ubuntu 2.6.31-14-generic. Detailed experimental results are reported in the appendix (Table A1) and a summary is presented in Table I. We make the following observations.

- In 9 out of 25 traces, Phase I alone was sufficient (row 1 in Table I) for our analysis.
- Around 80% of the witnesses in the majority of the traces are found to be infeasible due to the presence of a cycle in the witness order graph G' (Column 4). Since this is a quick check, most of the witnesses are handled quite expeditiously.
- Among the rest of the witnesses, a majority of them do not have choice edges ($\sim$17% of the total witnesses) (Column 5). For the traces in row 2, $\sim$3% of the witnesses have choice edges to be explored (Column 6).
- For the witnesses left with choice edges, even when the average number of possible choice graphs per witness is large (Column 7), the number of choice graphs actually explored per witness is close to 1 (Column 8). This is because the exploration stops as soon as an acyclic choice graph is detected. Thus, overall the average number of graphs explored per witness is very close to 1 also (Column 12).
- The time required for witness order graph analysis is much lower than that of choice graph analysis.

V. RELATED WORK

We have already discussed the broad categories of efforts in detecting dataraces and nondeterminism in Section I (Figure 2). We highlight specific related aspects below.

Datarace detection: Broadly, the approaches can be classified into three groups – (1) monitoring [32], [11], [13], [9], (2) predictive analysis [7], [8], [34] and (3) static analysis [35], [20], [22]. Like many of these techniques, we too use happens-before analysis and reasoning about locks. However, our focus is on detecting nondeterminism that is related to, but distinct from, datarace detection. Specifically, we do not have to provide witnesses with unsynchronized memory accesses, which may involve subtle reasoning about locks, e.g. by using lock acquisition histories [21] or causally-precedes relationships [34]. Rather, we consider witnesses with all possible orderings of related events (w, r, and c), where lock reasoning is used only to ensure mutual exclusion. We use a simple notion of lock scopes to enforce mutual exclusion. Chen et al. [8] used a related notion called lock atomicity sets, but they provide a richer abstraction (lock atomicity equivalence) for their purpose of predicting sound interleavings. UCG-based analysis [23] also used cycle-based infeasibility checks, but their analysis is incomplete for more than two threads where choice edges need to be considered. Our lock abstraction graph can be used to identify choice edge pairs in witness-based slicing for other checkers that may use UCG analysis.

Nondeterminism detection: Ensuring deterministic programs has received a lot of attention lately [5]. Vechev et al. proposed a static analysis for verifying determinism in structured parallel programs, based on checking non-overlapping memory accesses in parallel sections [37]. There is some work on specification and dynamic checking for determinism also [6], [31]. Burnim et al. proposed an assertion framework for specifying that programs should behave deterministically and used it to detect nondeterministic behavior [6]. Sadowski et al. proposed a new non-interference specification for deterministically-parallel code, and used a dynamic analysis tool called SideTrack to enforce it [31]. Many other efforts focus on adding synchronization or deterministic scheduling to preempt nondeterministic behavior or related bugs. Vaziri et al. associate synchronization constraints with fields of a class in object-oriented programs, and use static analysis to automatically infer synchronization points to avoid concurrency-related bugs [36]. Navabi et al. insert lightweight synchronization primitives at potential violation points [26]. DThreads replaces the `pthreads` library with an efficient deterministic multi-threading system [24]. CoreDet is a compiler and runtime system for general-purpose software deterministic multi-threading [3]. Other such systems are Determinator [1], Kendo [28] and dOS [4]. In contrast to these efforts, our work does not target specifying or enforcing determinism, but only to check it under standard synchronization and scheduling semantics. Any enforcements (using synchronization or deterministic thread scheduling) can be easily accounted for by adapting the partial orders we consider in our analysis. To the best of our knowledge, our work is the first to use predictive analysis for detecting nondeterminism.

VI. Conclusion

We have proposed a graph-based predictive analysis method for detecting nondeterminism in multi-threaded programs. We analyze each read-couple with all other writes to the same shared variable and determine the conditions for nondeterminism. When these conditions are satisfied, we generate a witness of nondeterminism. Further, we ensure no false positives by ensuring that our witness is feasible, i.e. there exists an interleaving where this witness will be observed. A key property of our method is that we provide a sound and complete[4] predictive technique that explores a reduced set of sufficient interleavings, thereby ensuring that it is efficient. Our experimental results demonstrate the effectiveness of our proposed method on several C/C++ and Java benchmark programs.

References

[1] Aviram, A., chun Weng, S., Hu, S., Ford, B.: Efficient System Enforced Deterministic Parallelism. In: OSDI (2010)

[2] Ball, T., Burckhardt, S., de Halleux, J., Musuvathi, M., Qadeer, S.: Deconstructing Concurrency Heisenbugs. In: ICSE. pp. 403–404. IEEE (2009)

[3] Bergan, T., Anderson, O., Devietti, J., Ceze, L., Grossman, D.: CoreDet: A Compiler and Runtime System for Deterministic Multi-threaded Execution. In: ASPLOS. pp. 53–64 (2010)

[4] Bergan, T., Hunt, N., Ceze, L., Gribble, S.D.: Deterministic Process Groups in dOS. In: OSDI. pp. 1–16. OSDI'10 (2010)

[5] Bocchino, Jr., R.L., Adve, V.S., Adve, S.V., Snir, M.: Parallel Programming Must Be Deterministic By Default. In: HotPar. pp. 4–4. HotPar'09 (2009)

[6] Burnim, J., Sen, K.: Asserting and Checking Determinism For Multi-threaded Programs. Commun. ACM 53 (Jun 2010)

[7] Chen, F., Rosu, G.: Parametric and Sliced Causality. In: CAV. pp. 240–253 (2007)

[8] Chen, F., Serbănută, T., Rosu, G.: jPredictor: A Predictive Runtime Analysis Tool for Java. In: ICSE. pp. 221–230 (2008)

[9] Choi, J.D., Lee, K., Loginov, A., O'Callahan, R., Sarkar, V., Sridharan, M.: Efficient and Precise Datarace Detection for Multithreaded Object-oriented Programs. In: PLDI. pp. 258–269. PLDI '02 (2002)

[10] Cui, H., Wu, J., Gallagher, J., Guo, H., Yang, J.: Efficient Deterministic Multithreading Through Schedule Relaxation. In: SOSP. pp. 337–351. SOSP '11 (2011)

[11] Elmas, T., Qadeer, S., Tasiran, S.: Goldilocks: A Race-Aware Java Runtime. Commun. ACM 53 (Nov 2010)

[12] Farchi, E., Nir, Y., Ur, S.: Concurrent Bug Patterns and How to Test Them. In: IPDPS. p. 286 (2003)

[13] Flanagan, C., Freund, S.N.: FastTrack: Efficient and Precise Dynamic Race Detection. In: PLDI. PLDI '09 (2009)

[4]complete for predicting from the given trace, not for the entire program

[14] Havelund, K.: Using Runtime Analysis to Guide Model Checking of Java Programs. In: SPIN. pp. 245–264 (2000)

[15] Havender, J.W.: Avoiding deadlock in multitasking systems

[16] http://incubator.apache.org/thrift/:

[17] http://research.microsoft.com/qadeer/cav_issta.htm: Joint CAV/ISSTA special event on specification, verification, and testing of concurrent software

[18] http://www.princeton.edu/∼sinha/FMCAD12_Traces.zip:

[19] http://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/index_1.html: Java grande forum benchmark suite

[20] Kahlon, V., Yang, Y., Sankaranarayanan, S., Gupta, A.: Fast and accurate static data-race detection for concurrent programs. In: CAV. pp. 226–239. Springer (2007), LNCS 4590

[21] Kahlon, V., Ivancic, F., Gupta, A.: Reasoning About Threads Communicating via Locks. In: Computer Aided Verification. pp. 505–518 (2005), LNCS 3576

[22] Kahlon, V., Sankaranarayanan, S., Gupta, A.: Semantic Reduction of Thread Interleavings in Concurrent Programs. In: TACAS. TACAS '09 (2009)

[23] Kahlon, V., Wang, C.: Universal Causality Graphs: A Precise Happens-Before Model for Detecting Bugs in Concurrent Programs. In: CAV. pp. 434–449. Springer (2010)

[24] Liu, T., Curtsinger, C., Berger, E.D.: DThreads: Efficient Deterministic Multithreading. In: SOSP. pp. 327–336. SOSP '11 (2011)

[25] Mcdowell, C.E., Helmbold, D.P.: Debugging Concurrent Programs. ACM Computing Surveys 21, 593–622 (1989)

[26] Navabi, A., Zhang, X., Jagannathan, S.: Quasi-Static Scheduling For Safe Futures. In: Chatterjee, S., Scott, M.L. (eds.) PPoPP. pp. 23–32. ACM (2008)

[27] Netzer, R.H.B., Miller, B.P.: What Are Race Conditions?: Some Issues and Formalizations. ACM Lett. Program. Lang. Syst. 1 (March 1992)

[28] Olszewski, M., Ansel, J., Amarasinghe, S.: Kendo: Efficient Deterministic Multithreading in Software. SIGPLAN Not. 44, 97–108 (Mar 2009)

[29] Papadimitriou, C.H.: The Serializability of Concurrent Database Updates. J. ACM 26(4), 631–653 (1979)

[30] von Praun, C., Gross, T.R.: Static Detection of Atomicity Violations in Object-Oriented Programs. Object Technology 3(6) (2004)

[31] Sadowski, C., Freund, S.N., Flanagan, C.: SingleTrack: A Dynamic Determinism Checker for Multithreaded Programs. In: ESOP. ESOP '09 (2009)

[32] Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: Eraser: A Dynamic Data Race Detector for Multithreaded Programs. ACM Trans. Comput. Syst. 15(4), 391–411 (1997)

[33] Sinha, A., Malik, S., Wang, C., Gupta, A.: Predictive Analysis for Detecting Serializability Errors through Trace Segmentation. In: MEMOCODE (2011)

[34] Smaragdakis, Y., Evans, J., Sadowski, C., Flanagan, J.Y.C.: Sound Predictive Race Detection in Polynomial Time. In: POPL (2012)

[35] Sterling, N.: WARLOCK - A Static Data Race Analysis Tool. In: USENIX Winter. pp. 97–106 (1993)

[36] Vaziri, M., Tip, F., Dolby, J.: Associating Synchronization Constraints With Data In An Object-Oriented Language. In: POPL (2006)

[37] Vechev, M., Yahav, E., Raman, R., Sarkar, V.: Automatic Verification of Determinism For Structured Parallel Programs. In: SAS. SAS'10 (2010)

APPENDIX

A. DETAILED EXPERIMENTAL RESULTS

The detailed experimental results are given in Table A1. For each benchmark, column 1 presents various statistics of the logged program traces: threads (thrds), number of events (evs), number of lock events (l-evs) and lock variables (l-vars), number of read/write events (rw-evs) and shared variables (rw-vars) and number of wait-notify events (wn-evs). Column 2 shows the total number of possible witnesses in the observed trace. Columns 3-5 and 6-7 show the results of analyses based on witness order graphs and choice graphs, respectively. For the witness order graphs, we report the number of infeasible witnesses (i.e. cycle found) (column 3), number of feasible witnesses (no choice edges and no cycle) (column 4) and witnesses left with choice edges (column 5). Similarly, for the choice graphs, we report the number of possible choice graphs per witness in column 5 that have choice edges (column 6) and number of choice graphs explored per witness in column 5 that have choice edges (column 7). Column 8 shows the total time taken for the analysis. Columns 9 and 10 show the total feasible witnesses and the total infeasible witnesses, respectively. Column 11 reports the average number of graphs analyzed per witness in column 2.

B. PROOF SKETCH OF THEOREM 1

Proof Sketch of Witness Order Graph Theorem:

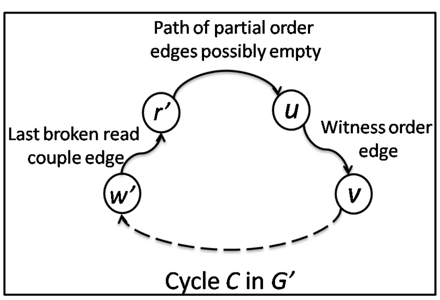

Figure B1. Case 2 of the proof of Theorem 1

Let C be the cycle in G'. The partial order edges/induced edges led by partial order edges and read-couples/induced edges led by read-couples only cannot constitute C[5] (otherwise this contradicts the total order of ρ). Therefore, C must contain a witness order edge.

Case 1: (C contains witness order edge and partial order edge/induced edge led by partial order edge only. C does not contain read-couple/induced edge led by read-couple.) Let τ be an interleaving that contains ω. Due to C, τ is cyclic. Then there does not exist a valid interleaving τ (since it must be a total order of events) that contains ω. Hence, ω is infeasible.

[5]either a partial order edge or a read-couple edge leads to an induced edge

Case 2: (C contains witness order edge, partial order edge/induced edge led by partial order edge and at least one read-couple/induced edge led by read-couple.) In interleaving τ, at least one read-couple in C must be broken for τ to be a total order since the witness order edges must be observed for τ to be a witness interleaving. What we now need to show is that such broken read couple can alter program flow so that some event x, where x is an event of ω, may not occur. Thus ω will be infeasible.

Let (w', r') be the last read-couple that is broken in C before a witness order edge and let (u, v) be the first witness order edge in C after (w', r') (Fig. B1). Note that there cannot be any unbroken read-couple (w'', r'') between (w', r') and (u, v) in C because all such reads after a broken read are not guaranteed to happen.

From the construction of the witness order edge (u, v) we know there are two possible cases.

1) Different lock variable case (Fig. 4(c)): In this case, u is an event in the witness ω. Since the read-couple (w', r') is broken and there are only partial order edges between r' and u, u is not guaranteed to happen in τ, and thus ω is infeasible.
2) Same lock variable case (Fig. 4(b)): In this case, there is a vertex x, where $x \in \omega$ and $[x \dots u]_l$, i.e. x and u are in the same locked scope. Since there is a path through partial order edges from r' to u and (w', r') is broken, u may not occur. If u does not occur, then a) either the entire scope $[x \dots u]_l$ is not executed, in which case ω is infeasible as x is an event in ω, or b) x occurs, but u does not occur and thus the witness ω cannot continue along (u, v). Thus ω is infeasible.□

C. PROOF SKETCH OF THEOREM 2

Proof Sketch of Choice Graph Theorem: ($\Leftarrow$) All choice graphs represent traces that are consistent with the witness order graph $G'(\omega)$. We know that G' does not have a cycle, otherwise it would have been detected before. In a choice graph, all the unordered pairs of locked scopes represented in S_{choice} are ordered. The presence of cycle in a choice graph $G''(\omega)$ implies that the witness ω is infeasible with respect to the particular ordering of locked scopes present in $G''(\omega)$. Similarly, the presence of cycles in all choice graphs implies that the witness is infeasible with respect to all the orderings of locked scopes represented in S_{choice}. Hence, ω is infeasible.

($\Rightarrow$) It is known that G' is acyclic (otherwise it would have been detected earlier). Therefore, ω is infeasible implies that there does not exist an (acyclic) interleaving τ that is consistent with any of the 2^d combinations of choice edges. Then all those 2^d graphs are cyclic. The cycles in these 2^d graphs can be divided into two categories, (1) cycles that do not contain any choice edge outside S_{choice}, and, (2) cycles that contain at least one choice edge outside S_{choice}. All cycles of the first category are present in $2^{d'}$ choice graphs. We are done if we can prove that (1) there is no cycle in the second category, and, (2) each of the choice graphs contain at least one cycle from the first category.

Subproof 1: We prove by contradiction. Let the witness be infeasible and there exists a cycle C in one of 2^d possible graphs that contains at least one choice edge e_{c1} from a choice edge pair $t = \{e_{c1}, e_{c2}\}$ outside S_{choice}. C must contain at least one witness order edge (y, x) (otherwise ρ is inconsistent). This choice edge e_{c1} in C is on the path between x and y. Therefore, by definition the choice edge pair t must be within S_{choice} leading to contradiction.

Subproof 2: We prove by contradiction. Let the witness ω be infeasible and there exists an acyclic choice graph G''. This implies that the particular combination of d' choice edges present in G'' does not lead to a cycle (since, by subproof 1, we know that there does not exist a cycle in 2^d combinations that contain choice edges from pairs outside S_{choice}). Then there exists an (acyclic) interleaving τ consistent with choice graph G'' containing ω. Then ω is feasible. This leads to the contradiction.

Hence, if the witness is infeasible, then all the choice graphs must have cycles in them.$\square$

Table A1

EXPERIMENTAL DATA ON THE WITNESSES OF NONDETERMINISM IN TRACES OF MULTI-THREADED PROGRAMS.

1. Benchmark	2. #Possible witnesses	Witness Order Graph Analysis			Choice Graph Analysis		8. Total time taken	9. Total feasible witnesses (%)	10. Total infeasible witnesses (%)	11. Avg. number of graphs analyzed per witness in column 2
		3. Witnesses with cycles in G' (infeasible) (%)	4. Witnesses with no choice edges (feasible) (%)	5. Witnesses with choice edges (%) (ϵ=$\sim$0)	6. Possible choice graphs per witness in column 5	7. Choice graphs explored per witness in column 5				
conpool - thrds: 4, evs: 97, l-evs: 16, l-vars: 1, rw-evs: 53, rw-vars: 5, wn-evs: 3	252	221 (88)	31 (12)	0 (0)	–	–	0.007s	31 (12.3)	221 (87.7)	1
liveness - thrds: 7, evs: 283, l-evs: 44, l-vars: 9, rw-evs: 163, rw-vars: 12, wn-evs: 6	855	709 (83)	146 (17)	0 (0)	–	–	0.064s	146 (17)	709 (83)	1
SynchBench - thrds: 16, evs: 1510, l-evs: 306, l-vars: 2, rw-evs: 533, rw-vars: 15, wn-evs: 0	47526	39474 (83)	8052 (17)	0 (0)	–	–	8.85s	8052 (17)	39474 (83)	1
Barrier1 - thrds: 10, evs: 653, l-evs: 108, l-vars: 2, rw-evs: 262, rw-vars: 12, wn-evs: 7	3975	3231 (81)	744 (19)	0 (0)	–	–	0.62s	744 (18.7)	3231 (81.3)	1
Barrier2 - thrds: 13, evs: 805, l-evs: 136, l-vars: 2, rw-evs: 340, rw-vars: 16, wn-evs: 7	6516	5305 (81)	1211 (19)	0 (0)	–	–	1.17s	1211 (18.6)	5305 (81.4)	1
account1 - thrds: 11, evs: 902, l-evs: 146, l-vars: 21, rw-evs: 430, rw-vars: 42, wn-evs: 10	1416	1042 (73.6)	326 (23)	48 (3.4)	36.33	2.83	1.352s	374 (26.1)	1058 (73.9)	1.06
account2 - thrds: 21, evs: 1747, l-evs: 282, l-vars: 41, rw-evs: 850, rw-vars: 82, wn-evs: 20	2796	2063 (73.8)	647 (23)	86 (3.2)	6.84	1.65	3.819s	733 (26)	2091 (74)	1.02
DaisyTest1 - thrds: 3, evs: 2998, l-evs: 422, l-vars: 10, rw-evs: 2003, rw-vars: 45, wn-evs: 15	383007	305635 (79.8)	61852 (16.1)	15520 (4.1)	6.35	1.12	244s	77372 (20.2)	305650 (79.8)	1.005
DaisyTest2 - thrds: 3, evs: 4998, l-evs: 699, l-vars: 14, rw-evs: 3330, rw-vars: 64, wn-evs: 25	1100814	871935 (79.2)	183794 (16.7)	45085 (4.1)	6.41	1.04	1130s	228879 (20.8)	871950 (79.2)	1.001
DaisyTest3 - thrds: 3, evs: 7999, l-evs: 1096, l-vars: 18, rw-evs: 5341, rw-vars: 81, wn-evs: 36	3025419	2386632 (78.9)	516746 (17.1)	122041 (4)	6.19	1.02	6393s	638787 (21.1)	2386647 (78.9)	1.0006
Elevator1 - thrds: 4, evs: 3004, l-evs: 370, l-vars: 11, rw-evs: 1795, rw-vars: 70, wn-evs: 0	3249	2671 (82)	578 (18)	0 (0)	–	–	0.8s	578 (17.8)	2671 (82.2)	1
Elevator2 - thrds: 4, evs: 5001, l-evs: 610, l-vars: 11, rw-evs: 3668, rw-vars: 117, wn-evs: 0	12285	9999 (81)	2286 (19)	0 (0)	–	–	8.1s	2286 (18.6)	9999 (81.4)	1
Elevator3 - thrds: 4, evs: 8004, l-evs: 1128, l-vars: 11, rw-evs: 5601, rw-vars: 94, wn-evs: 0	24627	20061 (81)	4566 (19)	0 (0)	–	–	24s	4566 (18.5)	20061 (81.5)	1

(Continued to next page)

1. Benchmark	2. #Possible witnesses	Witness Order Graph Analysis 3. Witnesses with cycles in G' (infeasible) (%)	4. Witnesses with no choice edges (feasible) (%)	5. Witnesses with choice edges (%) (ϵ=∼0)	Choice Graph Analysis 6. Possible choice graphs per witness in column 5	7. Choice graphs explored per witness in column 5	8. Total time taken	9. Total feasible witnesses (%)	10. Total infeasible witnesses (%)	11. Avg. number of graphs analyzed per witness in column 2
philo - thrds: 6, evs: 1141, l-evs: 126, l-vars: 6, rw-evs: 857, rw-vars: 23, wn-evs: 22	4893	4118 (84)	775 (16)	0 (0)	–	–	0.65s	775 (15.8)	4118 (84.2)	1
ThriftTrace1 - thrds: 4, evs: 2406, l-evs: 226, l-vars: 12, rw-evs: 869, rw-vars: 62, wn-evs: 53	618	361 (58)	96 (16)	161 (26)	105.6	1	1.6s	257 (41.6)	361 (58.4)	1
ThriftTrace2 - thrds: 4, evs: 2452, l-evs: 234, l-vars: 12, rw-evs: 873, rw-vars: 62, wn-evs: 55	618	361 (58)	96 (16)	161 (26)	105.6	1	1.6s	257 (41.6)	361 (58.4)	1
ThriftTrace3 - thrds: 10, evs: 4844, l-evs: 404, l-vars: 18, rw-evs: 1739, rw-vars: 129, wn-evs: 91	2811	2374 (84.5)	435 (15.5)	2 (ϵ)	34	1	1.2s	437 (15.5)	2374 (84.5)	1
ThriftTrace4 - thrds: 4, evs: 6761, l-evs: 586, l-vars: 48, rw-evs: 2785, rw-vars: 171, wn-evs: 125	35610	29441 (82.7)	5804 (16.3)	365 (1)	202	1	21.5s	6169 (17.3)	29441 (82.7)	1
ThriftTrace5 - thrds: 20, evs: 8950, l-evs: 722, l-vars: 28, rw-evs: 3320, rw-vars: 223, wn-evs: 165	9615	8058 (83.8)	1537 (16)	20 (0.2)	480.2	1	5.9s	1557 (16.2)	8058 (83.8)	1
ThriftTrace6 - thrds: 4, evs: 11357, l-evs: 1384, l-vars: 48, rw-evs: 3184, rw-vars: 171, wn-evs: 324	35610	29441 (82.7)	5804 (16.3)	365 (1)	296.7	1	28s	6169 (17.3)	29441 (82.7)	1
ThriftTrace7 - thrds: 20, evs: 13561, l-evs: 1520, l-vars: 28, rw-evs: 3727, rw-vars: 223, wn-evs: 372	9609	8055 (83.8)	1536 (16)	18 (0.2)	524.6	1	8.1s	1554 (16.2)	8055 (83.8)	1
ThriftTrace8 - thrds: 40, evs: 15975, l-evs: 1239, l-vars: 48, rw-evs: 5485, rw-vars: 429, wn-evs: 259	35748	29887 (83.6)	5809 (16.3)	52 (0.1)	261.6	1	35s	5861 (16.4)	29887 (83.6)	1
ThriftTrace9 - thrds: 10, evs: 16040, l-evs: 2981, l-vars: 18, rw-evs: 1345, rw-vars: 107, wn-evs: 1408	2577	2194 (85.1)	382 (14.8)	1 (0.1)	64	1	2.9s	383 (14.9)	2194 (85.1)	1
ThriftTrace10 - thrds: 6, evs: 20640, l-evs: 1724, l-vars: 158, rw-evs: 8818, rw-vars: 519, wn-evs: 349	479142	399814 (83)	79326 (16.5)	2 (ϵ)	18	1	243s	79328 (16.6)	399814 (83.4)	1
ThriftTrace11 - thrds: 6, evs: 25237, l-evs: 2522, l-vars: 158, rw-evs: 9218, rw-vars: 519, wn-evs: 549	479142	399814 (83)	79326 (16.5)	2 (ϵ)	18	1	477s	79328 (16.6)	399814 (83.4)	1