Predictive Analysis for Detecting Serializability Violations through Trace Segmentation

Arnab Sinha, Sharad Malik
Princeton University

Chao Wang, Aarti Gupta
NEC Laboratories America

***Abstract*—We address the problem of detecting serializability violations in a concurrent program using predictive analysis, where a violation is detected either in an observed trace or in an alternate interleaving of events in that trace. Under the widely used notion of conflict-serializability, checking whether a given execution is serializable can be done in polynomial time. However, when all possible interleavings are considered, the problem becomes intractable. We address this in practice through a graph-based method, which for a given atomic block and trace, derives a smaller segment of the trace, referred to as the *Trace Atomicity Segment (TAS)*, for further systematic exploration. We use the observed write-read pairs of events in the given trace to consider a set of events that guarantee feasibility, i.e., each interleaving of these events corresponds to some real execution of the program. We call this set of interleavings the *almost view-preserving (AVP)* interleavings. We show that the TAS is sufficient for finding serializability violations among *all* AVP interleavings. Further, the TAS enables a simple static check that can prove the absence of a violation. This check often succeeds in practice. If it fails, we perform a systematic exploration over events in the TAS, where we use dynamic partial order reduction with additional pruning to reduce the number of interleavings considered. Unlike previous efforts that are less precise, when our method reports a serializability violation, the reported interleaving is guaranteed to correspond to an actual execution of the program. We report experimental results that demonstrate the effectiveness of our method in detecting serializability violations for Java and C/C++ benchmark programs.**

I. Introduction

The atomicity of a set of operations is a desired correctness condition for concurrent programs. Informally, atomicity refers to the non-interference between shared accesses residing outside and inside an atomic region. A recent study shows that 69% of concurrency bugs are atomicity violations [1] and there is significant interest in the research community to address this problem [2–5].

There are different correctness notions associated with atomicity such as conflict-serializability and linearizability. Informally, an execution is conflict-serializable if it is equivalent, in some sense, to a serial execution where the individual atomic regions are executed sequentially [6, 7]. Consider the program execution shown in Figure 1. It has two concurrent threads T_1 and T_2, a globally visible pointer variable p and two thread-local variables a and b. The relevant events of interest are the read/writes to the shared variable p. Let p=0 initially. The three statements in thread T_1 should be executed atomically as they belong to a user transaction indicated by `atomic{...}`. Note that this is

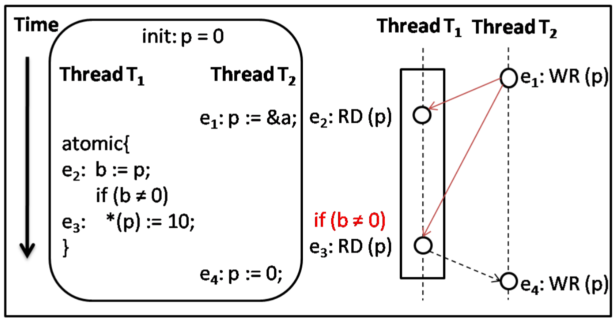

Figure 1. The program execution trace on the left can be graphically abstracted as a sequence of reads (RD) and writes (WR) of global variables on the right.

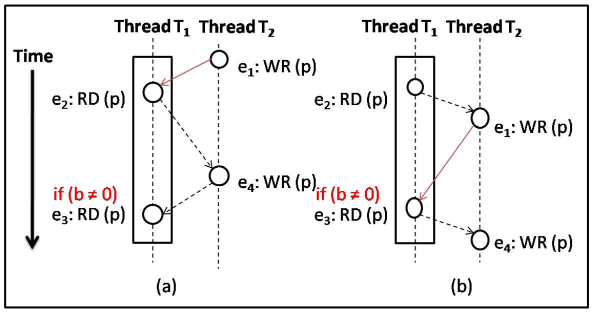

Figure 2. Two alternative, unserializable, interleavings of the same events as in Figure 1. The interleaving in (a) is guaranteed to be possible, hence it is a real violation, while the one in (b) is not guaranteed and thus a bogus violation.

a serial execution, since the shared pointer p is accessed atomically by T_1 between events e_1 and e_4. However, Figure 2(a) shows another execution order where event e_4 is interfering with the atomic block and hence breaks the conflict-serializability. Henceforth, we interchangeably refer to this kind of violation as a conflict-serializability violation or an atomicity violation.

Checking conflict-serializability of a given trace can be done in polynomial-time [5, 7, 8]. This problem is referred to as the *monitoring problem* [4, 8]. For concurrent programs, one may also be interested in checking conflict-serializability for all possible interleavings of a given trace. This is referred to as the *prediction problem*. In predictive analysis, a violation is detected either in the observed trace or in an alternate interleaving of events in that trace.

Broadly speaking, existing predictive analysis methods can be classified into two categories based on their precision. Methods in the first category detect *must-violations*, i.e. the reported violation must be a real violation. Methods in the second category detect *may-violations*, i.e. the reported violation may be a real violation. They also differ in the coverage they target when exploring alternate interleavings, as shown in Figure 3.

Methods in the first category often use under-approximated analysis, e.g. based on Lamport's happens-before causality relation and its extensions [9–11], the maximal causal model [12], SideTrack [13], and our proposed method TAS. A notable exception, which uses precise rather than under-approximated analysis, is the concurrent trace program (CTP) model [14]. However, achieving the best precision and high coverage is costly. Furthermore, in practical situations, it is not always possible to heavily instrument the program to extract a precise predictive model.

Methods in the second category often use over-approximated analysis, e.g. the Eraser-style lockset analysis [15], the meta-analysis based on lock acquisition histories [16], and the UCG analysis based on universal causality graph [17]. Note that in both the meta-analysis and the UCG-analysis, a reported erroneous schedule is not guaranteed to appear in some program execution, i.e. the reported violation may be bogus. The reason is that these methods track only the control flow and synchronizations while ignoring the data flow. This is in sharp contrast to our proposed method, which tracks data flow.

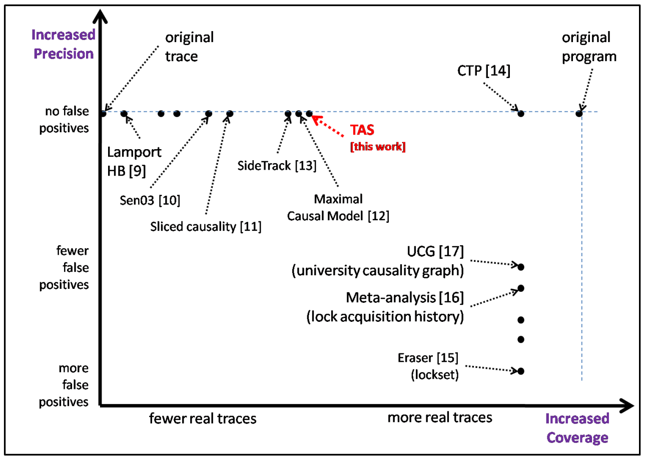

Figure 3. Classification of predictive analysis techniques based on precision and coverage.

A. Motivation

The motivation for our current work is to explore a different trade off point in this space of coverage and precision. We want to design a predictive model that admits interleavings likely to exhibit errors, while sticking with feasible interleavings and using scalable analysis. In some sense, we want to explore the limits of coverage one can achieve with feasible traces extracted from the observed trace. In particular, we want to avoid the post-analysis runtime check for feasibility. While other methods [12, 13] also have similar goals, the key ingredients of our method are different – the predictive model, the target set of interleavings, and the reductions we use.

We target general conflict-serializability violations, since they are very challenging and involve reasoning over all possible shared variables over all threads. Most previous efforts do not address such violations for more than two threads. Indeed, our predictive model and techniques can be easily adapted for detection of simpler concurrency violations (e.g. serializability violations for a single variables in two threads, or dataraces, etc.).

Consider the example shown in Figure 2 which motivates our approach. Although both executions (a) and (b) are unserializable interleavings of events in the trace shown in Figure 1, the violation in (a) is real, while (b) is not guaranteed to be real. In execution (b), event e_2 precedes e_1, assigning a different value to the shared variable p than in the original trace. This leads to the local variable b in T_1 being assigned a different value, thereby affecting its control flow. Specifically, when b equals zero, the program cannot execute event e_3, and thus there is no real atomicity violation.

B. Overview of our Approach

We use a graph-based predictive model with clock vectors to track events in the observed trace, where events correspond to visible operations in the program, i.e. read/write accesses on shared variables and synchronization operations. The clock vectors capture the causality (happen-before) constraints in the usual way [9]. In particular, we *couple* each read event on a shared variable with the last write event on the same variable that happened before in the trace. Note that this coupling may be inter-thread or intra-thread. Assume that during our exploration of alternate interleavings, all synchronizing constraints are respected (details provided later). Now, consider an interleaving of thread events, where the read events respect their coupling. Clearly this interleaving is guaranteed to correspond to a real execution. This is related to the well-known notions of view equivalence and view serializability [7].

However, if we explore complete view-preserving interleavings only, then we cannot hope to find view-serializability violations, and hence no conflict-serializability violations either. The idea is to consider interleavings where any thread is allowed to *break* its read-coupling, i.e. read from a different write event, but then to *skip* all subsequent events in this thread and its dependent events in other threads since they can no longer be guaranteed. We call this set of interleavings the *almost view-preserving (AVP) set*, since within each thread we preserve the view upto the broken read-coupling i.e. maintain the *view preserving prefix*.

In most practical cases, the program traces are very large with an even larger number of interleavings. We address this by designing a graph-based algorithm that focuses on a given atomic block (at a time), to derive a smaller segment of the trace for exploration. *We show that the segment we derive is sufficient to detect serializability violations for the given atomic block under all AVP interleavings*. We refer to this algorithm as *Trace Segmentation* and the segment as a *Trace Atomicity Segment* (TAS). We can further *refine* the TAS by shrinking it iteratively, depending on the access patterns in the given atomic block. The TAS also allows us to perform a simple static check before performing a search over all AVP interleavings. If the static check passes, then no violation is possible among the AVP set. If the static check fails, i.e. a violation is possible, we perform systematic exploration on events in the TAS, where we use known techniques such as dynamic partial order reduction (DPOR) [18, 19], with additional pruning and heuristics optimized for our setting, to reduce the number of interleavings to be checked. For each interleaving, the final step involves checking whether it violates conflict-serializability. We adopt the approach of [5], which builds a conflict (DSR) graph [7] for the interleaving and checks for a cycle in the graph.

We have implemented our method to work on traces generated from C/C++ and Java benchmark programs. The results show that our TAS-based method is effective at finding real violations in practice, while the TAS reduction and static check greatly help to improve performance (in comparison to a systematic search over all feasible traces).

This work makes the following contributions.

- We focus on finding real violations. Therefore, we explore a set of *almost view-preserving (AVP)* interleavings, where violations consist only of those thread-local prefixes that are guaranteed to correspond to some real program execution.
- We use a graph-based model for predictive analysis, where we derive a smaller segment of the trace called a *trace atomicity segment (TAS). We prove that the TAS we derive is sufficient for finding conflict-serializability violations among all AVP interleavings.*
- The TAS also enables a quick static check, to prove easy cases where no violations are possible among the target set of interleavings. This check often succeeds in practice.
- We propose a systematic exploration of the TAS when the static check fails. This utilizes dynamic partial order reduction and additional search pruning to further reduce the number of interleavings to be checked.
- We have implemented our TAS-based method and show promising results on C/C++ and Java benchmark programs.

II. Preliminaries and Problem Formulation

We consider a concurrent program as consisting of a set of *threads* $T_1, \ldots, T_k$ and a set of *shared variables*. Let $tid = \{1, \ldots, k\}$ 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.

A. Program Trace Model

An execution trace $\rho = e_1,\ e_2, \ldots e_n$ is a sequence of events, each of which is an instance of a *visible* operation during the execution of the concurrent program. For Java programs as well as C/C++ programs using POSIX Threads, the read/write accesses to shared variables and the synchronization operations are regarded as visible operations. All other operations are regarded as thread-local and are omitted in the execution trace. An event is represented as a 5-tuple $(tid, eid, type, var, child)$, where tid is the thread index, 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) or a synchronization object, $child$ is the child thread index (in thread creation/join). The event type is one of $\{read$, $write$, $fork$, $join$, $acquire$, $release$, $wait$, $notify$, $notifyall$, $atomic\_begin$, $atomic\_end\}$. They can be classified into four categories:

1) $read$ and $write$ denote the read and write accesses to a shared variable var;
2) $fork$ and $join$ denote the creation and termination of a child thread, where event (tid, eid, $fork$, -, $child$) creates a child thread whose index is $child$, and event (tid, eid, $join$, -, $child$) joins the child thread back;
3) $atomic\_begin$ and $atomic\_end$ denote the beginning and end of an atomic block, respectively. The marker events ($atomic\_begin$ and $atomic\_end$) may come from manual annotation of the program source code, or may be mined automatically [2, 20].
4) The rest correspond to synchronization operations over locks and condition variables. The `synchronized` keyword in Java is translated into a pair of $acquire$ and $release$ events over the lock implicitly associated with an object.

An execution trace ρ provides a total order on the events appearing in ρ. We derive a partial order by retaining only the set of must-happen-before constraints, which collectively are sufficient to guarantee feasibility of the serializations of this partial order.

B. The Partial Order Graph

Let $G(V, E)$ be the partial order graph which is constructed from an execution trace as follows. $V(G)$ 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). The directed edge (u,v)$\in E(G)$, the set of edges, if and only if at least one of the following conditions is true:

1) Program Order edge: when $u.tid = v.tid$ and $u.eid = v.eid - 1$, i.e. event u immediately precedes event v in program order.
2) Read-After-Write edge: when v reads the value of a shared variable written by u in the given trace ρ. This may be an intra- or an inter-thread edge.
3) Sync. edge: We consider the following sync. events.

- u is a $fork$ event and v is the first event in the child thread.
- u is terminating event for a thread and v is the corresponding $join$ event.
- u is a $wait$ event and v is the corresponding $notify$ event.

The PO (program order) edges enforce the program order within each thread. (For ease of exposition, we assume a sequential consistency memory model; other weaker memory models can be handled by relaxing these constraints.) The RAW (read-after-write) edges are needed to explore the AVP interleavings, which guarantee that the interleavings are feasible. Note that there are no WAW (write-after-write) and WAR (write-after-read) edges in G itself, but these edges are tracked while detecting atomicity violations in the interleavings. Sync. edges are also used for generating feasible interleavings. Note there are no edges for lock events - lock semantics are guaranteed during the exploration of interleavings.

Definition of a read-couple: If there is a RAW edge from a to b in G, then the pair (a, b) is a *read-couple*. If in a different interleaving, b reads from a different event c, we say that the read-couple for b in G is broken.

Definition of vector: Each vertex v in $G(V, E)$ has a vector vec_v of k integers, where the i^{th} integer denotes the eid of the latest event in thread i that must occur before v [9]. (We claim no novelty in defining the vector but provide our definition for clarity.) The vectors are computed in a topological order on the vertices. The vector of a vertex is defined as follows:

$$vec_v[i] = \begin{cases} v.eid - 1 & \text{if } (i = v.tid) \\ max(\mathsf{set}_1 \cup \mathsf{set}_2 \cup \{0\}) & \text{otherwise.} \end{cases} \quad (1)$$

where,

$$\mathsf{set}_1 = \{vec_u[i] \mid (u.tid \neq i) \wedge (u, v) \in E(G)\}, \text{ and,}$$

$$\mathsf{set}_2 = \{u.eid \mid (u.tid = i) \wedge (u, v) \in E(G)\}$$

Example: An example of a partial order graph with 3 threads is shown in Figure 4. The rectangular block in this figure represents an atomic block. The number inside each vertex is the eid. The vectors are shown in square brackets next to the vertices. For convenience, we shall refer to vertex 1 in the 2^{nd} thread as vertex 2.1.

Whether two vertices u and v in graph $G(V, E)$ may happen in parallel (MHP) can be conveniently checked by comparing their vectors. Formally, we define the following *event precedence relations* based on the vectors. For two vertices u and v,

- **MHB**: u must happen before v, denoted $u < v$, iff $vec_v[u.tid] \geq u.eid$.
- **MHP**: u may happen in parallel with v, denoted $u \mid v$, iff $vec_v[u.tid] < u.eid$, and $vec_u[v.tid] < v.eid$.

In our example (Figure 4), note that vertex 2.4 may happen in parallel with vertex 1.4 because, based on the two vectors $vec_{1.4}$ = [3,3,0] and $vec_{2.4}$ = [0,3,4], neither

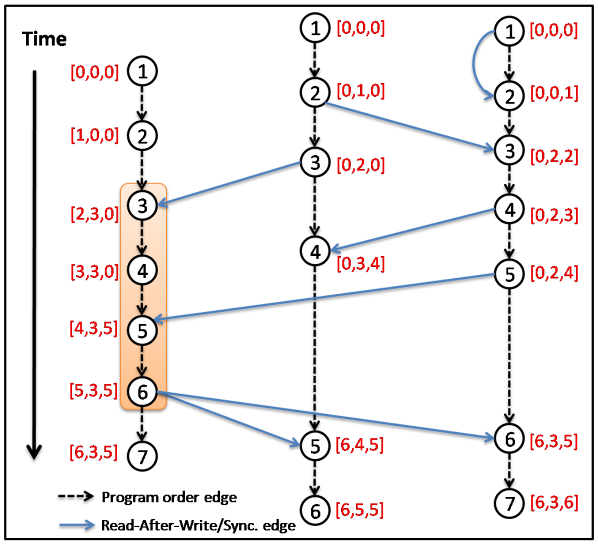

Figure 4. The partial order graph with vectors (computed according to Eq. 1). Each vertex is an event from the execution trace. The dashed edges are PO edges, and the solid edges are RAW/Sync. edges. Note that RAW edges can both be inter- and intra-thread edges.

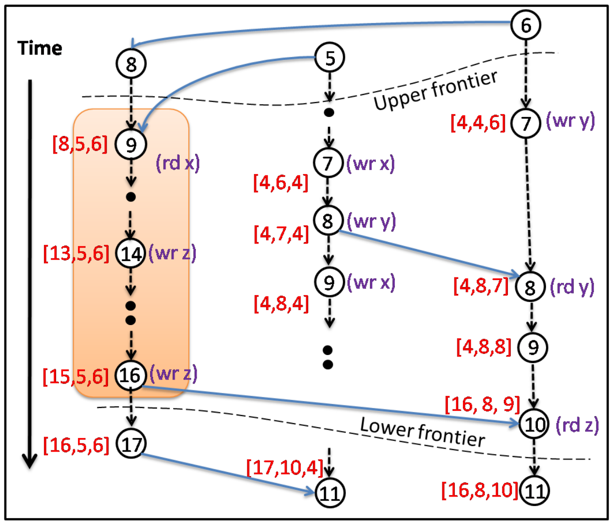

Figure 5. The TAS in a partial order graph G, with respect to the atomic block (shaded rectangular region). The upper and lower frontiers of the TAS are given by {8,5,6} and {17,11,11}, respectively.

vertex must happen before the other. In contrast, vertex 2.2 (with vector [0,1,0]) must happen before vertex 1.4 (with vector [3,3,0]).

Definition of atomic block: An atomic block $\mathcal{A}$ is a subgraph of G such that $V(\mathcal{A})$ is the set of events between the special marker events *atomic_begin* and *atomic_end*, all within the same thread that includes the marker events. The edge set is defined as $E(\mathcal{A}) = \{(u,v) \mid u,v \in V(\mathcal{A})$ & $u.eid = v.eid - 1\}$. Let, $T_{\mathcal{A}}$ be thread containing the atomic block and $T_{\mathcal{A}}.tid$ denote the thread index of vertices in $V(\mathcal{A})$. In a concurrent execution trace, there may be multiple atomic blocks in different threads. Our method performs violation detection on a single atomic block at a time.

C. Almost View Preserving (AVP) Interleavings

Let ρ be the given trace. We are targeting $AVP(\rho)$ as the maximal set of interleavings that are guaranteed feasible without interpreting the data values, and without a post-analysis/runtime check. This makes it an interesting trade-off point between search complexity and coverage. It is derived as follows. Let G' be a partial order graph derived from ρ similar to G except it contains only the program order and sync. edges. Let $t \in T$, the set of all interleavings consistent with G'. Let v be a read event in t. For each read event v in t if the read couple for v in ρ is broken in t then all vertices w for which v must-happens-before w in G are deleted from t resulting in t'. $AVP(\rho)$ is the set of all t' s.t. $t \in T$.

In Figure 1, $\rho = e_1, e_2, e_3, e_4$. Therefore, $AVP(\rho) = \{(e_1, e_2, e_3, e_4), (e_1, e_2, e_4, e_3), (e_1, e_4, e_2), (e_2, e_1, e_4)\}$. As shown in our experimental results, the AVP set constitutes a large number of interleavings in practice. Serbănută et al. [12] also proposed a maximal model based on actual observed values, but they allow only one value mismatch. Thus their set of interleavings and the AVP set are incomparable.

D. A Violation Path

It is well-known that there is a conflict-serializability violation if there exists a cycle in the D-serializable (DSR) graph [5, 7]. In our setting, for any alternate interleaving we *conceptually* construct a conflict graph G_C, (similar to DSR graph) where vertices are the read/write events and edges represent conflicting accesses and program order. There is an atomicity violation if we can find a path that starts and terminates within the atomic block, and visits at least one vertex outside the atomic block. For example, for the trace in Figure 1, $e_2 \rightarrow e_4 \rightarrow e_3$ is such a path in the alternate interleaving $\{e_1, e_2, e_4, e_3\}$. Next we formalize this notion.

Let, $V_{RW}(G) \subseteq V(G)$ be the set of vertices which read/write global variables. Similarly, $V_{RW}(\mathcal{A}) = V(\mathcal{A}) \cap V_{RW}(G)$. Let, $\rho' \in AVP(\rho)$. In the conflict-graph $G_C(V(G_C), E(G_C))$, $V(G_C)$ is the set of read/write events in ρ' and $E(G_C)$ consists of program order edges between the vertices in $V(G_C)$ and conflicting accesses: read-after-write (RAW), write-after-read (WAR) and write-after-write (WAW).

A *violation path* $P(\rho')$ is a sequence of alternating vertices and edges in G_C such that it originates from $\mathcal{A}$, visits at least one vertex outside $\mathcal{A}$ and terminates within $\mathcal{A}$, with no repetition of vertices. The vertices and edges in $P(\rho')$ are denoted by $V(P(\rho')) \subseteq V(G_C)$ and $E(P(\rho')) \subseteq E(G_C)$, respectively. There exists an atomicity violation for the atomic block $\mathcal{A}$ in ρ iff there exist a violation path $P(\rho')$. The vertex at which the path leaves (enters) the atomic block is referred to as $s(P(\rho'))$ ($t(P(\rho'))$). A violation path $P(\rho')$, if it exists, is characterized as follows:

- $s(P(\rho')).eid < t(P(\rho')).eid$ where, $s(P(\rho'))$, $t(P(\rho')) \in V_{RW}(\mathcal{A})$
- $V(P(\rho')) \cap \overline{V_{RW}(\mathcal{A})} \neq \{\}$, where $\overline{V_{RW}(\mathcal{A})}$ is the set of read/write vertices outside $\mathcal{A}$, i.e., $\overline{V_{RW}(\mathcal{A})} = V(G_C) \backslash V_{RW}(\mathcal{A})$.

The predictive analysis problem considered here is to detect the existence of a violation path $P(\rho')$ for all $\rho' \in AVP(\rho)$.

III. The Trace Atomicity Segment (TAS)

For an atomic block $\mathcal{A}$, we identify the TAS, i.e. a subgraph $\mathcal{Z}_{\mathcal{A}} \subseteq G$ that is sufficient for the purpose of detecting the serializability violations among $AVP(\rho)$. *Intuitively, the TAS captures the events that may happen in parallel with events in $\mathcal{A}$ till any broken reads are encountered.* Next, we define the boundaries of this TAS referred to as the *frontiers*.

A. The Frontiers

Let vertices u_0 and u_∞ be the 'first' and 'last' read/write vertices in $V_{RW}(\mathcal{A})$ respectively. Therefore, $u_0.eid$=min$\{v.eid \mid v \in V_{RW}(\mathcal{A})\}$ and $u_\infty.eid$= max$\{v.eid \mid v \in V_{RW}(\mathcal{A})\}$.

A *frontier* is a k-tuple, i.e. a vector, where the i^{th} integer represents the *eid* of some event in i^{th} thread. A TAS is bounded by two frontiers (*upper* and *lower* with respect to G) defined as follows (see Figure 5):

- **Upper Frontier**: $UF_{\mathcal{A}}$ is a frontier such that $UF_{\mathcal{A}} = vec_{u_0}$. Any coupled read that has its writer (wr) below the upper frontier (i.e. $wr.eid > UF_{\mathcal{A}}[wr.tid]$) is referred to as *read coupled below $UF_{\mathcal{A}}$*.
- **Lower Frontier**: $LF_{\mathcal{A}}$ is a frontier such that

$$LF_{\mathcal{A}}[i] = \begin{cases} u_\infty.eid + 1 & \text{if } i = T_{\mathcal{A}}.tid \\ min\{v.eid \mid v \in F\} & \text{if } i \neq T_{\mathcal{A}}.tid \text{ and } F \neq \{\}, \\ & \text{where, } F = \{v \mid v \in V(G), \\ & v.tid = i,\ vec_v[T_{\mathcal{A}}.tid] \geq \\ & u_\infty.eid, \text{ and } v \text{ is not read} \\ & \text{coupled below } UF_{\mathcal{A}}\} \\ \phi.eid & \text{if } i \neq T_{\mathcal{A}}.tid \text{ and } F = \{\}. \end{cases} \tag{2}$$

where ϕ denotes a special terminating event, added to every thread at the end.

The subgraph of G between the upper frontier and the lower frontier is called the TAS $\mathcal{Z}_{\mathcal{A}}$. Formally,

$$\begin{aligned} V(\mathcal{Z}_{\mathcal{A}}) = & \{u \mid u \in V(G) \ \wedge \ (u.eid > UF_{\mathcal{A}}[u.tid]) \\ & \wedge \ (u.eid < LF_{\mathcal{A}}[u.tid])\} \\ E(\mathcal{Z}_{\mathcal{A}}) = & \{(u,v) \mid u,v \in V(\mathcal{Z}_{\mathcal{A}}) \text{ and } (u,v) \in E(G)\} \end{aligned}$$

A vertex v is said to be 'on the $LF_{\mathcal{A}}$' ('below the $LF_{\mathcal{A}}$')

iff $LF_{\mathcal{A}}[v.tid] = v.eid$ ($LF_{\mathcal{A}}[v.tid] < v.eid$). Similarly, a vertex v is said to be 'on the $UF_{\mathcal{A}}$' ('above the $UF_{\mathcal{A}}$') iff $UF_{\mathcal{A}}[v.tid] = v.eid$ ($v.eid < UF_{\mathcal{A}}[v.tid]$). The intuition behind the frontiers is that no vertex above the upper frontier (below lower frontier) may appear after (before) any vertex $v \in V_{RW}(\mathcal{A}) \subseteq V(\mathcal{Z}_{\mathcal{A}})$ in any interleaving in $AVP(\rho)$. Also, note that a read coupled below $UF_{\mathcal{A}}$ is not allowed to be on the $LF_{\mathcal{A}}$, since a read couple can be broken in an alternate interleaving in $AVP(\rho)$.

Example: Figure 5 shows an example of a TAS for the atomic region in a partial order graph with three threads. The upper frontier is $UF_{\mathcal{A}}$={8,5,6}. The lower frontier is $LF_{\mathcal{A}}$={17,11,11}. Note that the vertices 1.8, 2.5 and 3.6 are on the $UF_{\mathcal{A}}$. Similarly, vertices 1.17, 2.11 and 3.11 are on the $LF_{\mathcal{A}}$. The subgraph in between the frontiers is the TAS $\mathcal{Z}_{\mathcal{A}}$. Note that $V_{RW}(\mathcal{A}) \subseteq V(\mathcal{Z}_{\mathcal{A}})$. Although, the frontiers are simple vectors, they are represented as cuts in Figure 5 as the frontiers demarcate the boundaries of the TAS. Next, we show the sufficiency of TAS for detecting the existence of a violation path for a given atomic block.

B. Sufficiency of TAS

Theorem 1: [TAS THEOREM] Let, v,$v' \in V_{RW}(G)$ such that $v.eid \leq UF_{\mathcal{A}}[v.tid]$, $v'.eid \geq LF_{\mathcal{A}}[v'.tid]$. $\forall \rho' \in AVP(\rho)$, there does not exist a violation path $P(\rho')$ s.t. $P(\rho')$ goes through v or v'.

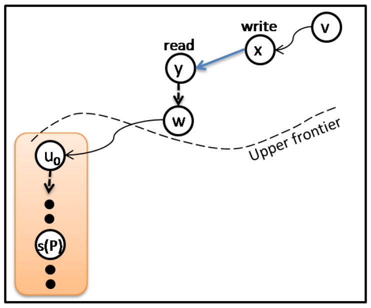

Figure 6. Case 1 of Theorem 1: v lies above $UF_{\mathcal{A}}$. Let, (x,y) be the broken read-couple on path connecting v and u_0 in G, such that $s(P(\rho'))$ precedes v in $P(\rho')$.

Proof Sketch: If a violation path goes through v or v' then (a) $s(P(\rho'))$ precedes v in $P(\rho')$, or (b) $t(P(\rho'))$ follows v' in $P(\rho')$. We consider both cases separately.

Case 1: Let there be a violation path $P(\rho')$ such that $s(P(\rho'))$ precedes v in $P(\rho')$. Since v lies above $UF_{\mathcal{A}}$, there must exist at least one path in G connecting v and u_0 and in all those paths u_0 follows v (see Figure 6). Moreover, u_0 precedes (or is the same as) $s(P(\rho'))$ in G. Therefore, at least one RAW edge in all paths in G connecting v and u_0 must be broken in ρ', otherwise we would have a cycle connecting u_0, $s(P(\rho'))$ and v in ρ'. Let, (x,y) be a broken read-couple in G which lies on a path (say τ) connecting v and u_0 in G. Consider the cases for y.

Case 1.1: y=u_0. All the events following u_0 in $\mathcal{A}$ will be skipped in G_C for ρ'. Hence, the violation path cannot terminate in $\mathcal{A}$ leading to contradiction.

Case 1.2: $y \neq u_0$. Since, y is a read-event there cannot be an outgoing inter-thread edge from y in G. Therefore, (x,y) must be immediately followed by a program order edge (y,w) in τ. However, if (x,y) is a broken read-couple, then w must be skipped (as w follows y in program order) in G_C for ρ'. Therefore τ discontinues after y in G_C. Thus u_0 and all following vertices including $s(P(\rho'))$ and $t(P(\rho'))$ in $V(\mathcal{A}) \notin V(G_C)$ and thus $P(\rho')$ cannot be a violation path.

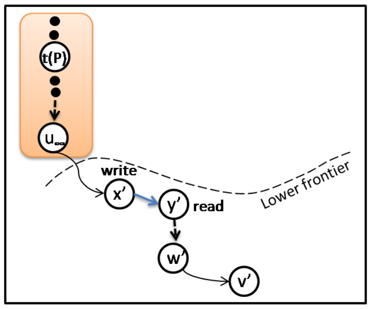

Figure 7. Case 2 of Theorem 1: v' lies below $LF_{\mathcal{A}}$. Let, (x',y') be the broken read-couple on path connecting u_∞ and v' in G, such that $t(P(\rho'))$ follows v' in $P(\rho')$.

Case 2: Let there be a violation path $P(\rho')$ such that $t(P(\rho'))$ follows v' in $P(\rho')$. Since v' lies below $LF_{\mathcal{A}}$, there must exist at least one path in G connecting u_∞ and v' and in all those paths u_∞ precedes v' (see Figure 7). Moreover, u_∞ follows $t(P(\rho'))$ in G. Therefore, at least one RAW edge in all paths in G connecting u_∞ and v' must be broken in ρ', otherwise we would have a cycle connecting $t(P(\rho'))$, u_∞ and v' in ρ'. Let, (x',y') be the broken read-couple in G which lies on a path (say τ') connecting u_∞ and v' in G. Consider the cases on y'.

Case 2.1: $y' = v'$. From the premise, $v'.eid \geq LF_{\mathcal{A}}[v'.tid]$. There are two possibilities.

Case 2.1.1: $v'.eid = LF_{\mathcal{A}}[v'.tid]$. The RAW edge ($x'$,$v'$) in $E(G)$ implies that $x' < v'$. If v' is on the $LF_{\mathcal{A}}$, then x' must be within $\mathcal{Z}_{\mathcal{A}}$. But, then v' is a read coupled below $UF_{\mathcal{A}}$. Therefore, by Eq. 2, v' cannot be on the $LF_{\mathcal{A}}$.

Case 2.1.2: $v'.eid > LF_{\mathcal{A}}[v'.tid]$. There must exist w such that $w.eid = LF_{\mathcal{A}}[v'.tid]$ (i.e. w is on $LF_{\mathcal{A}}$). Therefore, there must exist at least one path connecting u_∞ and w in G and in all those paths u_∞ precedes w. Moreover, w precedes v' in all interleavings in $AVP(\rho)$ since both events belong to the same thread. Therefore, in all interleavings in $AVP(\rho)$, u_∞ must precede v' and this precludes a path from v' to $t(P(\rho'))$.

Case 2.2: $y' \neq v'$ (shown in Figure 7). Let, (y',w') be the program order edge which follows (x',y') in τ'. Therefore, if (x',y') is the broken read-couple in ρ', then w' must be skipped. Thus, τ' discontinues after y' in ρ'

and v' is not present in $P(\rho')$. Hence the contradiction.

Corollary 1: [LOCALITY COROLLARY] The serializability of atomic block $\mathcal{A}$ over all AVP interleavings cannot be violated due to any vertex $v \notin V(\mathcal{Z}_{\mathcal{A}})$.

From Theorem 1, we infer that, for any vertices v and v' outside $\mathcal{Z}_{\mathcal{A}}$ (defined in the context of Theorem 1), no violation path is possible where v follows $source(P(\rho'))$ or v' precedes $target(P(\rho'))$), where $\rho' \in AVP(\rho)$. Therefore, a violation path P originating from and terminating into $\mathcal{A}$ can neither cross $UF_{\mathcal{A}}$ nor cross $LF_{\mathcal{A}}$. Hence, there exists no violation path that includes vertices outside $\mathcal{Z}_{\mathcal{A}}$.

C. Static Checking of TAS

Once the TAS $\mathcal{Z}_{\mathcal{A}}$ is identified, it is often possible to statically determine the absence of any AVP interleaving that violates atomicity. For a violation path, there must exist at least two events within the atomic block that conflict with other access(es) outside the atomic block. Formally, the necessary condition for existence of a violation path P is: there exist e_1, $e_2 \in V_{RW}(\mathcal{A})$ such that $\exists e'_1, e'_2 \in \overline{V_{RW}(\mathcal{A})}$, and e_1 (e_2) conflicts (RAW, WAR, WAW) with e'_1 (e'_2).

If such events e_1 and e_2 do not exist, then no violation is possible. Note that the conflicting accesses $\{e'_1, e'_2\}$ outside the atomic block may or may not correspond to the same event. It is important to note that the TAS frequently makes this check succeed, whereas it would typically fail for the full trace since it is likely that a shared variable (that is accessed within $\mathcal{A}$) is accessed outside the TAS. While the static check is straight-forward, it is this beneficial result of the TAS that we would like to highlight.

D. Exploration of TAS

If the static check fails for a TAS, i.e. an atomicity violation is possible, we systematically explore events in the TAS to generate the AVP interleavings. These interleavings are then checked for existence of a violation path. The exploration of TAS can be performed by any systematic search method (explicit or symbolic). In our current implementation, we employ explicit search inspired by dynamic partial order reduction [18, 19]. Interleavings with the same relative order between the conflicting events are defined as *conflict-equivalent* interleavings. The DPOR algorithm avoids generation of conflict-equivalent interleavings. However, off-the-shelf application of DPOR cannot generate AVP interleavings efficiently and several heuristics need to be used in our context. These have been omitted here for brevity.

E. Complexity

Let, $|V(G)| = N$, $|E(G)| = M$. The vertices in G needs to be topologically sorted for the task of vector assignment ($O(N + M)$). Once the vectors are assigned, derivation of TAS requires identification of upper ($O(1)$) and lower ($O(N)$) frontiers. Next, the static check requires a scanning of the vertices in the TAS ($O(N)$). Therefore, the total time complexity of deriving the TAS is $O(N + M)$. The complexity of search-space exploration depends on the method chosen, in our case this is the complexity of DPOR based search.

IV. RESULTS

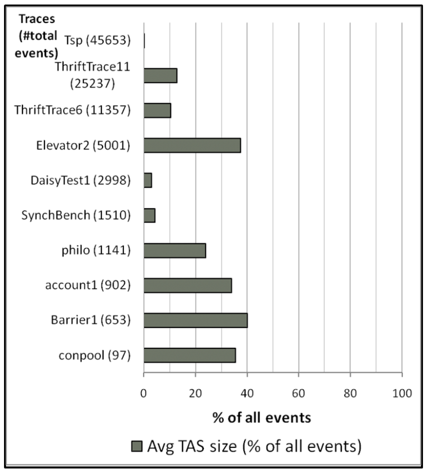

Figure 8. The relative TAS size tends to be significantly smaller than the trace size for the larger traces.

We have implemented our AVP predictive analysis technique in a prototype tool. This tool is capable of logging/analyzing execution traces generated by both Java programs and multithreaded C/C++ programs using Pthreads. The C++ benchmark used is available online [21]. All the Java benchmarks are also publicly available [22–26].

Next, we describe how the tool logs execution traces and analyzes those traces. 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 use execution traces logged at runtime by a modified Java Virtual Machine (JVM). For each test case, we first execute the program using the default OS thread scheduling and log the execution trace. Next we apply our algorithm to detect the serializability violations. For Java traces, we assume that all synchronized blocks are intended to be atomic, unless the synchronized block has a wait. For the C++ application, we assume that all blocks using scoped locks (monitors implemented using Pthreads locks and condition variables) are intended to be atomic.

All our experiments were conducted on an Intel Xeon machine with a 2.8 GHz Intel processor and 1GB memory running Linux. We want to answer the following questions through experimentation.

- How large is the TAS relative to the trace?
- How often does the static check on the TAS succeed?
- What is the computation time needed to detect the violations?

We considered the following scenarios.

- "no TAS": This shows the results of experiments where plain DPOR (without our heuristics) is applied to the entire trace for checking each atomic block.

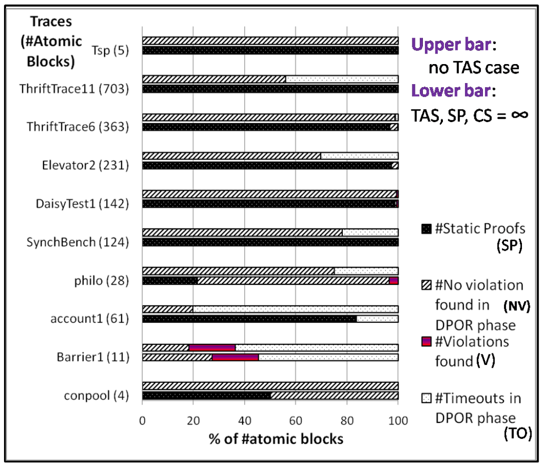

Figure 9. In larger traces, the timeouts decrease as the static checks increasingly succeed.

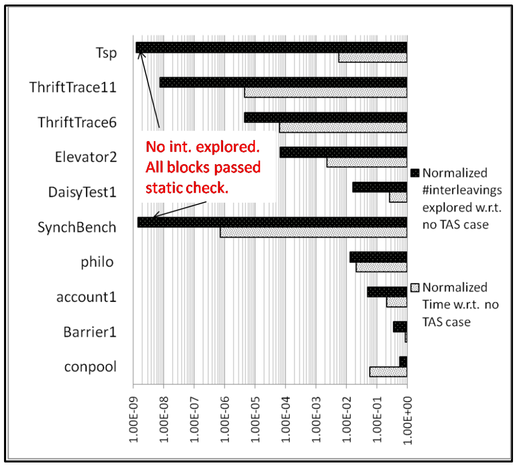

Figure 10. Our method catches violations exploring much fewer interleavings in significantly less time.

- "TAS, no SP": TAS and search heuristics but no static check.
- "TAS, SP, $CS_{max} = 5$": Alongwith TAS, static check and our optimized DPOR search technique we have considered context switch bounding [27] to 5 context switches ($CS_{max} = 5$) to reduce the number of interleavings considered).
- "TAS, SP, $CS_{max} = \infty$": TAS, static check, search heuristics with no bound on context switches.

Note that since the previous research on these Java benchmarks, including Fusion [14], are limited by their ability to check serializability violations involving at most 1 variable and 2 threads, we cannot provide an experimental comparison against them.

For ease of representation, we present the experimental results for a sample of traces in Figures 8, 9 and 10 for 'no TAS' and 'TAS, SP, $CS_{max} = \infty$' experiments. Henceforth, for brevity we refer to the 'TAS, SP, $CS_{max} = \infty$' case as the 'TAS' case. Additional details for these traces are presented in the Table I. The remaining traces in the benchmark set show similar characteristics. These figures support the claims we make next.

In Figure 8, the average TAS size (X-axis) is shown as a percentage of the total number of events. The Y-axis depicts the traces with the number of events shown within parenthesis (in increasing order of the number of events). Note that, the average TAS size of `Tsp` is 0.2% of the total events and hence the corresponding bar is virtually absent in the figure. Moreover, observe that *the relative TAS size is significantly smaller than the trace size, and tends to be more so for the larger traces for this set of benchmarks.*

For the same traces, Figure 9 shows that the static checks using the TAS often succeed which results in fewer timeouts. A timeout of 10 minutes per atomic block is chosen for the search phase. There are two horizontal bars per trace. The upper and the lower bar show the results of 'no TAS' and 'TAS' cases respectively. In the 'no TAS' case, there are three possible outcomes for each atomic block - 'no violation possible' (NV), 'violation found' (V) or 'timeout' (TO). We show the distribution of NV-V-TO for the 'no TAS' case in the upper bar. Similarly, in the 'TAS' case, the possible outcomes are: 'static check passed' (SP), NV, V, TO. The distribution of SP-NV-V-TO is shown for the 'TAS' case in the lower bar. Observe that in traces `philo`, `SynchBench`, `Elevator2`, `ThriftTrace11` the timeouts in the 'TAS' case disappear. Moreover, in traces `account1` and `Barrier1` the number of timeouts decreased significantly. *This is due to the large number of static checks that succeed in the 'TAS' case, which effectively bring down the number of timeouts.*

Figure 10 compares the execution time and the number of interleavings explored in the 'TAS' case normalized with respect to the 'no TAS' case. In all the traces, 'TAS' case outperformed 'no TAS' case in terms of number of interleavings explored and time taken. For instance in traces `SynchBench` and `Tsp`, all the TAS-es passed the quick static check in negligible time, demonstrating the efficiency of our static check. Moreover, we also found that for all these experiments, the time needed to compute the TAS was insignificant (less than 1 sec). Hence, we find that *our method catches violations in significantly less time by exploring much fewer interleavings.*

Discussion:

- In our 'TAS Search w/ Static Check' experiments, we detect 23 atomicity violations in 26 traces. This underscores our intuition that the predictive search-space of AVP interleavings has value in detecting atomicity violations.
- We find that the static check on TAS is very effective. For example, in the `ThriftTrace` benchmarks, more than 90% of the atomic blocks are statically determined not to have a violation. This drastically reduces the total time, which is dominated by the time spent in search. On average, 81.75% of the atomic blocks pass this test.

- Usually the 'no TAS' experiments run longer compared to experiments involving TAS. This shows that the trace reduction enabled by TAS is effective. On average only 23.88% (best case 0.2%) of the entire trace is included in a TAS. *Moreover, the TAS enables us to handle non-terminating/streaming applications, which cannot be analyzed by any technique that needs a complete trace.*
- The results indicate less than 1% timeouts over all checks in 'TAS, SP, $CS_{max} = \infty$'. This is due to an effective combination of TAS with DPOR. Specifically, we observe that TAS size of traces with mostly decoupled threads (i.e. insignificant inter-thread interaction) are bigger. However, fortunately, in such cases, the DPOR-based search explores only a very few interleavings e.g. in almost all the `Thrift` traces, although the absolute TAS sizes are fairly large, less than 20 interleavings were explored. Similar observation can be made regarding `Elevator` traces. *Thus, combining TAS with DPOR works well for both tightly as well as loosely coupled threads on this set of benchmarks.*

V. Related Work

We have already discussed the broadly related efforts on predictive analysis. More specifically, the maximal causal model proposed by Serbănută et al. [12] captures a related but incomparable set of interleavings. They record the actual values read and written in the trace, and allow only a single mismatch among the values read. In contrast, our predictive model does not interpret the values (and could handle nondeterministic values, e.g. in the initial state), and we allow multiple mismatches (in different threads) among the AVP interleavings explored. Farzan and Madhusudan use the notion of *causal atomicity* [6], which also relies on conflict-equivalence and hence is closely related to conflict-serializability. (A causal atomicity violation is a violation of conflict-serializability; but the reverse is not always true.) Besides this difference, as mentioned earlier, their predictive model does not guarantee feasible traces, i.e. may have false positives [16]. Their predictive analysis has been recently used to guide runtime testing [28].

Among other related techniques, Lu *et al.* [2] used access interleaving invariants to capture patterns of test runs and then monitor production runs for detecting three-access atomicity violations. Xu *et al.* [20] used a variant of the two-phase locking algorithm to monitor and detect serializability violations. Both methods were aimed at detecting, not predicting, violations in the given trace. Wang and Stoller [29] also studied the prediction of serializability violations under the assumptions of deadlock-freedom and nested locking; their algorithms are precise for checking violations involving one or two transactions but incomplete for checking arbitrary runs.

VI. Conclusions

We have proposed a graph-based predictive analysis method for detecting serializability violations in concurrent programs. Our method is precise in that when it reports a serializability violation, the reported interleaving is guaranteed to be feasible in the actual program execution (hence no false positives). We directly address the performance issues inherent in exploring a large number of interleavings by first deriving a smaller segment (TAS) of the given trace for checking a given atomic block. We prove that it is sufficient for exploring all interleavings in our target set (AVP interleavings). We also use the TAS to provide a quick static check for proving the absence of violations, which very often succeeds in practice. For the systematic search over all interleavings, we employ dynamic partial order reduction and other pruning techniques to reduce the number of interleavings to be checked. Our experimental results demonstrate the effectiveness of our TAS-based approach on several C and Java benchmark programs.

References

[1] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics," in *ASPLOS*. ACM, 2008, pp. 329–339.

[2] S. Lu, J. Tucek, F. Qin, and Y. Zhou, "AVIO: detecting atomicity violations via access interleaving invariants," in *ASPLOS*, 2006, pp. 37–48.

[3] F. Chen, T. Serbănută, and G. Rosu, "jPredictor: a predictive runtime analysis tool for java," in *ICSE*, 2008, pp. 221–230.

[4] A. Farzan and P. Madhusudan, "The complexity of predicting atomicity violations," in *TACAS*, 2009, pp. 155–169.

[5] A. Sinha and S. Malik, "Runtime checking of serializability in software transactional memory," in *IPDPS*, 2010, pp. 1–12.

[6] A. Farzan and P. Madhusudan, "Causal atomicity," in *CAV*, 2006, pp. 315–328.

[7] C. H. Papadimitriou, "The serializability of concurrent database updates," *J. ACM*, vol. 26, no. 4, pp. 631–653, 1979.

[8] A. Farzan and P. Madhusudan, "Monitoring atomicity in concurrent programs," in *CAV*, 2008, pp. 52–65.

[9] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[10] K. Sen, G. Rosu, and G. Agha, "Runtime safety analysis of multithreaded programs," in *Foundations of Software Engineering (FSE'03)*. ACM, 2003, pp. 337–346.

[11] F. Chen and G. Rosu, "Parametric and sliced causality," in *CAV*. Springer, 2007, pp. 240–253, LNCS 4590.

[12] T. F. Serbănută, F. Chen, and G. Rosu, "Maximal causal models for multithreaded systems," Tech. Rep.

[13] J. Yi, C. Sadowski, and C. Flanagan, "Sidetrack: generalizing dynamic atomicity analysis," in *PADTAD*, 2009, pp. 1–10.

[14] C. Wang, R. Limaye, M. Ganai, and A. Gupta, "Trace-based symbolic analysis for atomicity violations," in *TACAS*, 2010, pp. 328–342.

[15] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A dynamic data race detector for multithreaded programs," *ACM Trans. Comput. Syst.*, vol. 15, no. 4, pp. 391–411, 1997.

[16] A. Farzan and P. Madhusudan, "Meta-analysis for atomicity violations under nested locking," in *CAV*, 2009, pp. 248–262.

[17] V. Kahlon and C. Wang, "Universal Causality Graphs: A precise happens-before model for detecting bugs in concurrent programs," in *CAV*. Springer, 2010, pp. 434–449.

[18] C. Flanagan and P. Godefroid, "Dynamic partial-order reduction for model checking software," *SIGPLAN Not.*, vol. 40, no. 1, pp. 110–121, 2005.

[19] Y. Yang, X. Chen, and G. Gopalakrishnan, "Inspect: A

1. Traces	2. AB	3. Avg. TAS size (% of all events)	4. Scenarios	5. SP (% of all) AB's)	6. NV	7. V	8. TO	9. Int.	10. Time
conpool thrds: 4, evs: 97 l-evs: 16, l-vars: 1 rw-evs: 53, rw-vars: 5 wn-evs: 3	4	34.5 (35.56)	no TAS	-	4	0	0	56	0.2s
			TAS, no SP	-	4	0	0	37	0.017s
			TAS, SP, CS_{max}=5	2 (50)	2	0	0	33	0.02s
			TAS, SP, $CS_{max} = \infty$		4	0	0	33	0.02s
Barrier1 thrds: 10, evs: 653 l-evs: 108, l-vars: 2 rw-evs: 262, rw-vars: 12 wn-evs: 7	11	262.6 (40.2)	no TAS	-	2	2	7	5.4M	1h17m
			TAS, no SP	-	3	2	6	1.9M	1h4s
			TAS, SP, CS_{max}=5	0 (0)	0	2	2	571K	23m44s
			TAS, SP, $CS_{max} = \infty$		3	2	6	1.9M	1h4s
account1 thrds: 11, evs: 902 l-evs: 146, l-vars: 21 rw-evs: 430, rw-vars: 42 wn-evs: 10	61	307.6 (34.1)	no TAS	-	12	0	49	16.8M	8h1m
			TAS, no SP	-	47	0	14	1M	2h20m
			TAS, SP, CS_{max}=5	51 (83.6)	51	0	10	396K	1h40m
			TAS, SP, $CS_{max} = \infty$		51	0	10	839K	1h40m
philo thrds: 6, evs: 1141 l-evs: 126, l-vars: 6 rw-evs: 857, rw-vars: 23 wn-evs: 22	28	273.5 (23.9)	no TAS	-	21	0	7	9.2M	1h10m
			TAS, no SP	-	27	1	0	123K	1m31s
			TAS, SP, CS_{max}=5	6 (21.43)	6	0	0	20K	18s
			TAS, SP, $CS_{max} = \infty$		27	1	0	123K	1m30s
SynchBench thrds: 16, evs: 1510 l-evs: 306, l-vars: 2 rw-evs: 533, rw-vars: 15 wn-evs: 0	124	64.68 (4.2)	no TAS	-	97	0	27	33.6M	4h31m
			TAS, no SP	-	122	0	2	1.8M	20m14s
			TAS, SP, CS_{max}=5	124 (100)	124	0	0	0	0s
			TAS, SP, $CS_{max} = \infty$		124	0	0	0	0s
DaisyTest1 thrds: 3, evs: 2998 l-evs: 422, l-vars: 10 rw-evs: 2003, rw-vars: 45 wn-evs: 15	142	88.5 (2.9)	no TAS	-	141	1	0	298	0.3s
			TAS, no SP	-	141	1	0	241	0.17s
			TAS, SP, CS_{max}=5	140 (98.6)	140	1	0	5	0.1s
			TAS, SP, $CS_{max} = \infty$		141	1	0	5	0.1s
Elevator2 thrds: 4, evs: 5001 l-evs: 610, l-vars: 11 rw-evs: 3668, rw-vars: 117 wn-evs: 0	231	1875.4 (37.5)	no TAS	-	161	0	70	1.8M	11h42m
			TAS, no SP	-	231	0	0	517	2m5s
			TAS, SP, CS_{max}=5	225 (97.4)	225	0	0	124	1m38s
			TAS, SP, $CS_{max} = \infty$		231	0	0	125	1m35s
ThriftTrace6 thrds: 4, evs: 11357 l-evs: 1384, l-vars: 48 rw-evs: 3184, rw-vars: 171 wn-evs: 324	363	1191.36 (10.5)	no TAS	-	358	1	4	2.8M	40m17s
			TAS, no SP	-	362	1	0	19K	29s
			TAS, SP, CS_{max}=5	351 (96.7)	351	1	0	13	0.2s
			TAS, SP, $CS_{max} = \infty$		362	1	0	13	0.2s
ThriftTrace11 thrds: 6, evs: 25237 l-evs: 2522, l-vars: 158 rw-evs: 9218, rw-vars: 519 wn-evs: 549	703	3272.5 (12.9)	no TAS	-	393	0	310	126.8M	51h40m
			TAS, no SP	-	703	0	0	1001	2.9s
			TAS, SP, CS_{max}=5	702 (99.9)	702	0	0	1	0.9s
			TAS, SP, $CS_{max} = \infty$		703	0	0	1	0.9s
Tsp thrds: 4, evs: 45653 l-evs: 20, l-vars: 5 rw-evs: 25366, rw-vars: 42 wn-evs: 3	5	97 (0.2)	no TAS	-	5	0	0	76	0.2s
			TAS, no SP	-	5	0	0	5	0.012
			TAS, SP, CS_{max}=5	5 (100)	5	0	0	0	0s
			TAS, SP, $CS_{max} = \infty$		5	0	0	0	0s

Table I
EXPERIMENTAL DATA OF THE SERIALIZABILITY VIOLATION DETECTION. (AB=ATOMIC BLOCKS, SP=STATIC PROOFS, NV=NO VIOLATION POSSIBLE, V=VIOLATIONS FOUND, TO=TIMEOUTS, INT.=NO. OF INTERLEAVINGS GENERATED)

Runtime Model Checker for Multithreaded C Programs," University of Utah, Tech. Rep. UUCS-08-004, 2008.

[20] M. Xu, R. Bodík, and M. D. Hill, "A serializability violation detector for shared-memory server programs," in *PLDI*, 2005, pp. 1–14.

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

[22] E. Farchi, Y. Nir, and S. Ur, "Concurrent bug patterns and how to test them," in *IPDPS*, 2003, p. 286.

[23] K. Havelund, "Using runtime analysis to guide model checking of java programs," in *SPIN*, 2000, pp. 245–264.

[24] http://research.microsoft.com/qadeer/cav_issta.htm, "Joint cav/issta special event on specification, verification, and testing of concurrent software."

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

[26] C. von Praun and T. R. Gross, "Static detection of atomicity violations in object-oriented programs," *Object Technology*, vol. 3, no. 6, 2004.

[27] M. Musuvathi and S. Qadeer, "Iterative context bounding for systematic testing of multithreaded programs," *SIGPLAN Not.*, vol. 42, no. 6, pp. 446–455, 2007.

[28] F. Sorrentino, A. Farzan, and P. Madhusudan, "Penelope: Weaving threads to expose atomicity violations," in *FSE*. ACM, Nov, 2010.

[29] L. Wang and S. D. Stoller, "Runtime analysis of atomicity for multithreaded programs," *IEEE Trans. Software Eng.*, vol. 32, no. 2, pp. 93–110, 2006.