Predicting Serializability Violations: SMT-based Search vs. DPOR-based Search

Arnab Sinha[1], Sharad Malik[1], Chao Wang[2], and Aarti Gupta[3]

[1] Princeton University
[2] Virginia Polytechnic Institute
[3] NEC Laboratories America

Abstract. In our recent work, we addressed the problem of detecting serializability violations in a concurrent program using predictive analysis, where we used a graph-based method to derive a predictive model from a given test execution. The exploration of the predictive model to check alternate interleavings of events in the execution was performed explicitly, based on stateless model checking using dynamic partial order reduction (DPOR). Although this was effective on some benchmarks, the explicit enumeration was too expensive on other examples. This motivated us to examine alternatives based on symbolic exploration using SMT solvers. In this paper, we propose an SMT-based encoding for detecting serializability violations in our predictive model. SMT-based encodings for detecting simpler atomicity violations (with two threads and a single variable) have been used before, but to our knowledge, our work is the first to use them for serializability violations with any number of threads and variables. We also describe details of our DPOR-based explicit search and pruning, and present an experimental evaluation comparing the two search techniques. This provides some insight into the characteristics of the instances when one of these is superior to the other. These characteristics can then be used to predict the preferred technique for a given instance.

1 Introduction

The atomicity of a set of operations is a desired correctness condition for concurrent programs. There exist many different notions of atomicity, useful in various contexts [23, 12, 4]. In this paper, we address conflict-serializability [23], a widely used notion. 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. In general, atomicity violations can be detected by observing a particular trace, called the *monitoring problem* [5, 28], or by exploring alternate interleavings of events in a given trace, called the *predictive analysis problem* [6, 27, 29, 34, 38, 26, 7, 18].

The existing predictive analysis methods are broadly 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 [27, 38, 34]. Methods in the second category detect *may-violations*, i.e. the reported violation may be a real violation [26, 7, 18]. Due to higher precision, the methods in the first category are usually expensive. Therefore, effort is needed to improve performance and scale better on long traces.

In our recent prior work on predictive analysis [29], we proposed a method in the first category, where we used a graph-based technique to derive a predictive model called a TAS (Trace Atomicity Segment) that is based on read-write and synchronization events in the observed trace. The TAS is used to generate alternate interleavings, called *Almost View Preserving (AVP)* interleavings, that are guaranteed to be feasible executions of the concurrent program. Therefore, any serializability violations detected on AVP interleavings are also guaranteed to be real violations. The basic idea is to consider alternate interleavings where any thread is allowed to *break* its read-coupling, i.e. it reads from a different write event than what was observed in the given trace, but to *skip* all subsequent dependent events in this and other threads (since they can no longer be guaranteed to be feasible). Essentially, AVP interleavings preserve the view in each thread upto some prefix, and only the prefixes and broken read events are allowed to appear in a serializability violation.

1.1 Motivating Example

To motivate the issue of feasible executions, consider a program execution shown in Figure 1, with events numbered in the order of execution in the given trace. It has four concurrent threads ($T_1 \ldots T_4$) and four shared variables (x, y, z and t). The rectangular boxes (in threads T_2 and T_4) denote pre-specified atomic blocks (e.g. using `begin_atomic` and `end_atomic` labels in the trace). The inter-thread edges represent read-after-write (RaW) orders. Suppose that event e_9 in T_1 is conditional on the value of z in e_5, which gets its value from e_4. Note that the given trace is a serializable execution since the events can be ordered as (e_1, e_4, e_7),e_2, e_3, e_5, e_9,(e_6, e_8) without violating atomicity of the blocks. In Figure 2, we show two other possible interleavings of the same events, where the atomicity of the atomic block in thread T_2 is violated.

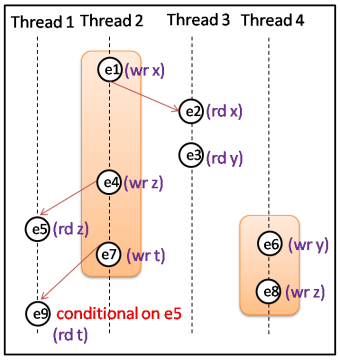

Fig. 1: This is a serializable program execution trace.

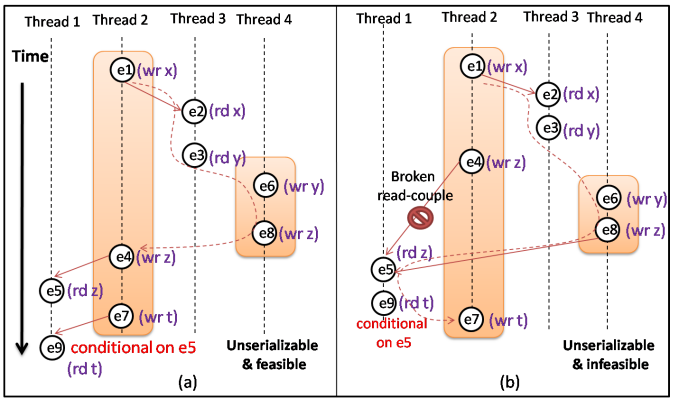

Fig. 2: The interleaving (a) is unserializable and feasible. However, the interleaving (b) is unserializable but not guaranteed to be feasible. (The broken arrow represents serializability violation path defined in Section 2.)

Although both interleavings shown in Figure 2 are unserializable, the violation in (a) is real, while the violation in (b) may be bogus. Specifically, in execution (b), event e_8 happens between e_4 and e_5, thereby assigning a different value to the shared variable z in event e_5 than in the original trace. Therefore, the value read by event e_5 may not be the same (we call this a *broken-read*), which may affect the occurrence of event e_9 down the line.

1.2 Overview and Contributions

In our prior work [29], we used an explicit state traversal technique based on Dynamic Partial Order Reduction (DPOR) [9] to generate AVP interleavings from our predictive model, and each interleaving was checked by looking for a cycle in the corresponding D-serializable (DSR) graph [23, 28]. Although use of dynamic partial order reduction ensures that no redundant interleavings (with respect to the conflict-based partial order)

are generated, the number of interleavings was still quite high for many benchmarks in our experiments. This motivated us to examine symbolic search techniques, that avoid an explicit enumeration of the AVP interleavings.

Note that symbolic encodings of interleavings in a concurrent program have been studied earlier [33, 34, 30, 31]. However, their focus is on read-write consistency constraints to ensure feasible interleavings, and suitable interference abstractions and refinement to weaken/strengthen these constraints for modular analysis. These encodings (capturing a concurrent trace) can be utilized with a suitable encoding of any target property for verification. In this paper, we use a similar encoding of read-write consistency (adapted to our predictive model), and combine it with an encoding for checking serializability violations (for any number of variables, for any number of threads) based on cycle detection. This general serializability property has not been addressed in any of the earlier symbolic efforts, which handled only data races, deadlocks, and simple atomicity/serializability violations (involving a single variable and two threads). Indeed, our symbolic encoding of serializability violations based on cycle detection can be potentially combined with other SMT-based encodings of interleavings in a concurrent program.

In recent years, the advancements in SAT and SMT solvers [20, 22] have been leveraged in many areas of automated software verification, including symbolic exploration for concurrent programs, e.g. symbolic partial order reduction [36], symbolic predictive analysis [35, 34], datarace detection [25]. Therefore, it is natural to examine whether SMT-based search can be useful in our method for predictive analysis. Although previous methods have used SMT-based encodings for detecting simple serializability violations involving two threads and a single variable [34], to the best of our knowledge, our method proposed here is the first SMT-based technique for detecting general serializability violations involving any number of threads and variables. Our SMT-based encoding can be potentially useful in other settings as well, e.g. bounded model checking [1, 10].

In this paper, we describe the DPOR-based search with additional pruning and heuristics optimized for our predictive analysis setting (these details were not described in our prior work [29], and also provide the context for comparison with SMT-based search). We then describe our SMT-based search in detail. Our SMT-based encoding performs both tasks: (a) symbolic exploration of AVP interleavings, and (b) detection of cycles in the associated DSR graph.

We have implemented our SMT-based technique and compare it with our DPOR-based technique on a suite of C/C++ and Java benchmark programs. The comparison reflects the classic tradeoff between time and space, and our results show that these techniques are complementary in the sense that neither outperforms the other on all benchmarks. We then consider some features of our predictive model that indicate which technique is likely to perform better (based on our current experiments). Specifically, we consider the relative TAS size (how big is the TAS model, relative to the trace) and coupling between threads (number of inter-thread edges, relative to number of all events) that seem to be good predictors.

Contributions: To summarize, this work makes the following contributions.

- We describe details of a DPOR-based explicit technique for exploration of interleavings, with additional pruning and heuristics optimized for finding serializability violations.
- We propose an SMT-based technique for detecting serializability violations using predictive analysis, suitable for modern SMT-solvers.
- We provide experimental results for the SMT-based technique, using two state-of-the-art SMT-solvers – Yices [16] and Z3 [32].
- Finally, we present the comparative results between DPOR-based and SMT-based techniques in our predictive analysis, and identify some characteristics of instances that can be used to select between them.

The following section introduces the needed background on our predictive analysis method [29]. Next, we will discuss the DPOR-based explicit exploration technique (Section 3) and the SMT-based encoding for the symbolic technique (Section 4). We present our experimental results in Section 5 and conclude in Section 6.

2 Preliminaries

In this section, we summarize the needed background on our prior work. We omit the detailed formal discourse (available online in [29]), and describe the main aspects. We consider a concurrent program 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.

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 (read/write accesses to shared variables and synchronization operations are regarded as visible operations) during the execution of the concurrent program. 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 operations) or a synchronization object, $child$ is the child thread index (in thread create/join). The event type is one of $\{read, write, fork, join, acquire, release, wait, notify, notifyall, atomic\_begin, atomic\_end\}$.

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.

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). A directed edge in $E(G)$ (the set of edges) is either a program order (PO), or a read-after-write (RaW), or a synchronization (Sync.) edge. If there is a RaW edge from a to b in G, then the pair (a, b) is defined as a *read-couple*. In a different interleaving, if b reads from a different event c, we say that the read-couple for b in G is *broken*. We assign clock-vectors to each vertex in G, following the idea of Lamport's Logical Clock [19] in order to check the causality order between any two events. An example of a partial order graph with 3 threads is shown in Figure 3. The rectangular block in this figure represents an atomic block denoted by $\mathcal{A}$. $V(\mathcal{A})$ denotes the vertices in $\mathcal{A}$. 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.

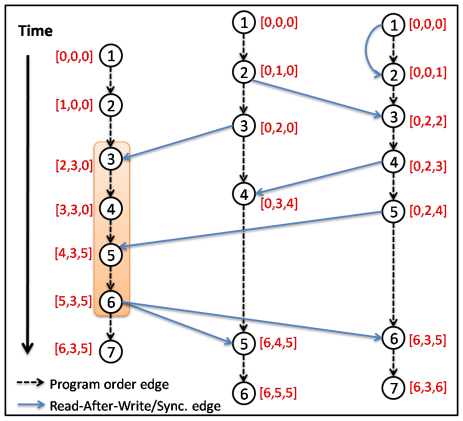

Fig. 3: The PO graph with vectors. Vertices represent events from the trace. The dashed and solid edges are PO and RaW/Sync. edges respectively. Note that RaW edges can both be inter- and intra-thread edges.

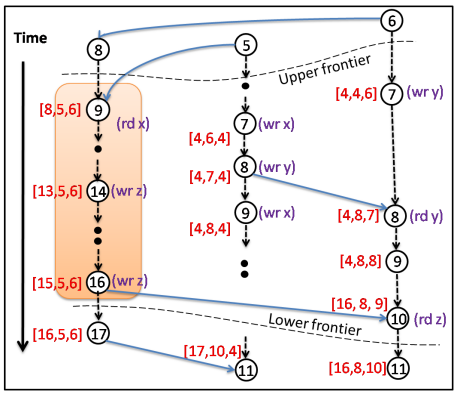

Fig. 4: The TAS in a partial order graph G, with respect to the atomic block (shaded rectangular region). The upper and lower frontiers are given by {8,5,6} and {17,11,11}, respectively.

Almost View Preserving (AVP) Interleavings: Let G' be a partial order graph derived from ρ similar to G, except that it has only the program order and sync. edges. Let $t \in T$, where T be 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, such that v must-happen-before w in G, are deleted from t resulting in t'. $AVP(\rho)$ is the set of all t' s.t. $t \in T$.

Serializability Violation Path: It is well-known that there is a conflict-serializability violation if there exists a cycle in the D-serializable (DSR) graph [23, 28]. In our setting, for any alternate interleaving we *conceptually* construct a conflict graph G_C (similar to a DSR graph), where the 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 in G_C.

Trace Atomicity Segment (TAS): For each atomic block $\mathcal{A}$, we identify a TAS as a subgraph $\mathcal{Z}_\mathcal{A} \subseteq G$ that is sufficient for the purpose of detecting all serializability violations among $AVP(\rho)$. Intuitively, the TAS captures all events that *may happen in parallel with events in* $\mathcal{A}$, until broken reads are encountered. Our overall technique derives a TAS for each atomic block in the given trace. In effect, this considers a sliding window over the trace, where each window looks at alternate interleavings among events that can happen in parallel with an atomic block.

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 (see Figure 4). Events that must happen before the first event within $\mathcal{A}$ are above the upper frontier. Analogously, events that must happen after the last event in $\mathcal{A}$ are below the lower frontier. The subgraph of G between the upper frontier and the lower frontier is called the TAS $\mathcal{Z}_\mathcal{A}$. The usefulness of frontiers is that no vertex above the upper frontier (below the lower frontier) may appear after (before) any vertex $v \in V_{RW}(\mathcal{A}) \subseteq V(\mathcal{Z}_\mathcal{A})$ in any interleaving in $AVP(\rho)$ (where $V_{RW}(\mathcal{A})$ denotes the read-write vertices in the given atomic block).

Example: Figure 4 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}. 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 4 as the frontiers demarcate the boundaries of the TAS. *We have shown that the TAS for a given atomic block is sufficient for detecting the existence of a serializability violation path among $AVP(\rho)$, i.e. there exists no violation path that includes vertices outside $\mathcal{Z}_\mathcal{A}$.* In effect, the TAS serves as our predictive model, and we search over all AVP interleavings over events in the TAS.

Note that 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 but within the TAS. Although such events may exist across a long trace, they may not occur within a relatively small TAS. This provides a static check: If such events do not exist in a TAS, then no violation is possible among $AVP(\rho)$. In practice, this static check is frequently successful.

3 Explicit Search within the TAS

If the TAS $\mathcal{Z}_\mathcal{A}$ fails the above simple static check, i.e. a serializability violation may be possible, we search among all the interleavings within $\mathcal{Z}_\mathcal{A}$ to find the unserializable one. In this section, we use a Dynamic Partial Order Reduction (DPOR [9, 37]) based explicit search algorithm. We also improve this DPOR search with several sound pruning techniques and search heuristics.

3.1 Overview of DPOR algorithm

An interleaving prefix $\pi = e_1, e_2, \ldots e_k$ is a sequence of a subset of the events in ρ; that is, $\forall i$, event e_i belongs to ρ, $|\pi| \leq |\rho|$, and events in π are not necessarily in the same order as in ρ. Two interleaving prefixes comprised of the same subset of events are *conflict-equivalent* iff the relative order of all pairs of conflicting events is same. The DPOR algorithm can be used to generate one representative interleaving from each conflict-equivalent class, by avoiding the other (redundant) interleavings.

In the DPOR algorithm, an interleaving prefix π is represented as a sequence $s_1, s_2, \ldots s_k$ of program states, where event e_i is executed during the transition from s_i to s_{i+1}, for all $i = 1, 2, \ldots k$. At each state s, we use $s.enabled$ to record the set of threads that are ready to execute. The next event in any thread $\tau \in s.enabled$ is referred to as the *ready-*

transition. We also use $s.backtrack \subseteq s.enabled$ to record a subset of the enabled threads, where each thread $\tau \in s.backtrack$ represents a possible scheduling choice at s in some future runs. Note that $\tau \notin s.backtrack$ means that, according to the partial order reduction theory, executing thread τ at state s would have led to a redundant interleaving [8].

Two mutually independent transitions (t_i,t_j) whose events are both ready for execution are referred to as *co-enabled* transitions. For instance, if a lock is acquired by one thread, it must be released before another thread can acquire it. Therefore, the transition that releases the lock and the transition that acquires it are mutually dependent, and hence are not co-enabled transitions.

Although DPOR is efficient in testing concurrent programs, it is not geared toward enumeration of interleavings in a predictive model such as ours. More specifically, it does not take TAS and the read-write coupling requirements into consideration. Therefore, we have customized the original DPOR for TAS, and our new algorithm is presented in Figures 5& 6. (Our modifications are lines marked with a '$\star$'.) In the pseudo-code, we use symbol S to denote the state stack ($s_0 s_1 \dots s_d \dots s_n$), and use $s.stack\_depth$ to denote the depth of state s in stack S. We start search with the first event in an atomic region $\mathcal{A}$ (say u_0) (lines 1-3, procedure **Init**). At each state s, we first find a set of preceding states (whose next events are mutually dependent with the enabled events at s) and update their *backtrack* sets (lines 2-12, procedure **Explore**). Here, in order to properly update the *backtrack* sets and avoid redundant interleavings, we need to track the conflicting pairs of transitions within the interleaving prefix. After finding and updating the backtrack sets of dependent preceding states, we need to pick an enabled thread at state s to execute. Rather than randomly picking (as in the original DPOR), we heuristically pick a thread, insert it in $s.backtrack$ (lines 14-15, procedure **Explore**), and continue. We continue exploring the enabled threads in $s.backtrack$ by calling **Explore** recursively, until all threads in this backtrack set are explored (lines 17-27). At this point, we insert the thread into the *done* set (line 18).

In this DPOR search, deciding whether any of the read-couples is broken in the current prefix is crucial. We use a map called *active_couple* to store the mapping from the 'written but not read' shared variables to the last coupled-reader in ρ (a written value can be read by multiple readers). Once the last reader reads the variable, the map is removed from *active_couple* (line 12, procedure **computeEnabledRaW**).

The set *enabled* is computed by procedure **computeEnabledRaW** if the current transition is a read/write access (Figure 6) and by the standard procedure **computeEnabled** otherwise (pseudo-code omitted for brevity). Procedure **computeEnabledRaW** is designed specifically to handle constraints from the RaW edges (or couples), while procedure **computeEnabled** deals with the standard synchronization primitives. It is worth pointing out that the original DPOR does not have or need procedure **computeEnabledRaW**. In our case, if a coupled read is mismatched, i.e. the read event is not reading the value it is supposed to read, the following events in the thread are skipped (line 6-7, procedure **computeEnabledRaW**).

Several helper procedures such as **preProcess** and **postProcess** (lines 1 and 16, respectively) are also called. We omit their pseudo code, but provide a brief description as follows. Procedure **preProcess** helps in initializing the various data-structures of a state (except for s_0, the current state inherits those data-structures from the previous state). Procedure **postProcess** helps in eliminating or inserting thread-ids into the *enabled* of the current state depending on certain conditions related to the ready-transitions and the data-structures of the current state.

Although we are performing a restricted search (by using RaW edges to reduce the number of interleavings), the explicit search overhead is still not practical in most of our initial experiments. These unrealistic run-times pushed us to look for smarter prunings and heuristics. In order to further reduce the number of interleavings, we propose several sound pruning techniques (Section 3.2) and search heuristics (Section 3.3).

3.2 Pruning TAS Search Space

Consider the scenario in Figure 7 (a), where $t_{lastInAB}$ is the last transition in the atomic region (from state s_d) and t is the current transition in the interleaving prefix. Assume that t_i, t_j and t_k are the predecessor transitions from states s_i, s_j and s_k respectively,

```
Init {
1:   if(u_0.type = read or write)
         computeEnabledRaW(s_0, s_0, u_0);
2:   else computeEnabled(s_0, s_0, u_0);
3:   S.push(s_0);
4:   Explore(S);
5: }

Explore(S) {
1:   let s = S.top_of_stack();
2:   for each thread h{
3:       let t_n be a transition such that t_n.tid ∈ s.enabled and t_n.tid = h
         and t_n is not a coupled read in the atomic block;
4:       for all transitions t_d in the current explored path dependent with
             t_n and it may be co-enabled with t_n{
5:           let s_d be the state in S from which t_d is executed;
⋆ 6:         if (s_d.cs ≥ CS_max or
               (bt_tag.isTrue ? (bt_tag.stack_depth< s_d.stack_depth):false))
⋆ 7:                 continue;
8:           let E = {q.tid ∈ s_d.enabled | (q.tid = h ∧ q is not a coupled
                 read from the atomic block) ∨ (t_d ≺ q ≺ t_n and q is
                 dependent with some t' such that, q ≺ t' ≺ t_n &
                 t'.tid = h)};
9:           if (E ≠ {}) then
                 choose any member of E and add to s_d.backtrack;
10:          else
                 add {q.tid | q.tid ∈ s_d.enabled and q is not a coupled
                 read from the atomic block} to s_d.backtrack;
11:      }
12:  }
13:  if (s.enabled is not empty) {
⋆ 14:    heuristically, pick t.tid from s.enabled;
15:      s.backtrack ← {t.tid};
16:      let done = {};
17:      while (∃t such that t.tid ∈ s.backtrack \done){
18:          add t to done;
19:          let, s' ← next(s, t);
⋆ 20:        computeContextSwitch(s, s', t); // compute cs for s'
⋆ 21:        if(t.type = read or write) computeEnabledRaW(s, s', t);
22:          else computeEnabled(s, s', t); // compute s'.enabled
23:          S.push(s');
24:          Explore(S);
25:          S.pop();
26:      }
27:  }
28: }
```

Fig. 5: Generating and checking the non-interfering runs

```
computeEnabledRaW (states: s_p, s_c, transition: t) {
    // s_p: previous state;  s_c: current state;
1:   preProcess (s_p, s_c);
2:   if (t.type = write and this write is coupled) {
3:     s_c.active_couple ← s_c.active_couple ∪ {(t.var, last reader of t)};
4:   } else if (t.type = read and this read is coupled) {
5:     if (t.var ∈ s_c.active_couple){
6:       if (read-couple is broken)
7:         s_c.next(t.tid) ← null; // thread virtually terminates
8:       if (t = s_c.active_couple[t.var]) //t is the last reader
9:         s_c.active_couple ← s_c.active_couple.erase(t.var);
10:    }
11:  }
12:  if(read-couple is broken and t is the broken read within the atomic region) {
13:      bt_tag.isTrue ← true;
14:      bt_tag.stack_depth ← s_p.stack_depth;
15:  }
16:  postProcess (s_p, s_c);
17: }
```

Fig. 6: Computing *enabled* using RaW edges.

conflicting with t. Our observation is that, if the atomicity property is not violated in the interleaving prefix until $t_{lastInAB}$, then updating the backtrack set of any successor state after s_d cannot generate any violation path. This claim can be justified as follows. First, updating the backtrack set of successors of s_d will not change the interleaving prefix until $t_{lastInAB}$. Second, there is no transition within the atomic block after $t_{lastInAB}$, and no violation exists in the prefix until $t_{lastInAB}$. Hence there does not exist a serializability violation path in the interleavings with fixed prefix until $t_{lastInAB}$. Therefore, in Fig-

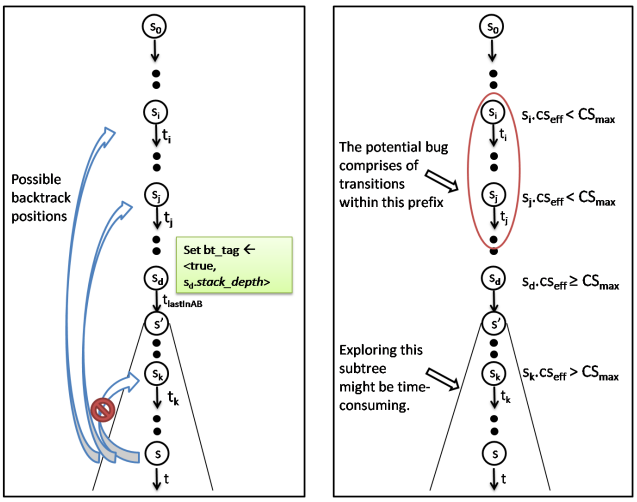

Fig. 7: (a) Setting a marker when the last transition ($t_{lastInAB}$) from the atomic region appears in the prefix helps in pruning the interleavings. (b) Tuning the parameter CS_{max} can help in carrying out localized search.

ure 7 (a), we update the backtrack sets of s_i and s_j only (and leave the backtrack set of s_k unchanged).

We implement this sound pruning technique in line 6 of procedure **Explore**, by introducing the global data-structure *bt_tag* (read as *backtrack-tag*). This data-structure has two fields: a Boolean variable $isTrue$ and an integer variable *stack_depth*. The data-structure is set in line 13-14 in procedure **computeEnabledRaW**, if current transition t is a mismatched read in $\mathcal{A}$ and the following events within $\mathcal{A}$ are skipped. The field *stack_depth* records the depth of s_d in the state-stack S such that, subsequent updates of the backtrack set of s can be ignored if $bt\_tag.isTrue$ is true and $bt\_tag.stack\_depth < s.stack\_depth$ (lines 6-7, Fig. 5).

3.3 Heuristics for TAS Search

We also use context bounding (i.e. limiting the number of context-switches) as a search heuristic to reduce cost. However, this is an unsound reduction technique because it may miss real bugs. We call a context switch *significant* when it is either inevitable (i.e. the previous transition is the last event of its thread) or needed to facilitate read-after-write or wait-notify couples. All other context-switches are referred to as *insignificant*. Our definition of *insignificant* context switches differs from the *preemptive* context switches in CHESS [21] since we also consider the RaW constraints.

We assign a counter $s.cs$ to record the number of insignificant context switches in the interleaving prefix up to state s. Let CS_{max} be the maximum number of insignificant context-switches permitted by the user. Observe that in line 6 of procedure **Explore**, for a state s_d, if $s_d.cs \geq CS_{max}$, we will skip the update of its backtrack set. The intuition behind this conditional update is as follows. Assume that there is a potential violation path comprising of fewer insignificant context-switches, then it will be detected by our algorithm with a small predetermined CS_{max}. In line 14, procedure **Explore**, the thread is heuristically picked from $s.enabled$ to efficiently utilize this budget. In practice, we can broaden the search space incrementally, by first exploring the interleavings with fewer context switches, and then gradually increasing the maximum number of insignificant context switches. In other words, the insignificant context-switch bounding enables a localized search within a fixed prefix length.

The complexity of this clock-vector based algorithm, as derived in [9], is $O(kdr)$, where k is number of threads, d is the maximum size of the search stack and r is the number of transitions explored.

4 Implicit Search within the TAS

As discussed earlier, an alternate framework for systematic exploration of events within the TAS is SMT-based implicit exploration. We now show how to encode the problem as an SMT instance using difference logic. This instance can then be analyzed by state of the art SMT solvers.

4.1 Encoding of the Violation Path Reachability

Consider the example in Figure 8. All the vertices shown in the figure (u, v, v' and w) write to variable x. The path $u \rightarrow v \rightarrow v' \rightarrow w$ is a violation path. We encode this violation path which may be present in some alternate interleaving $\rho' \in AVP(\rho)$. Although, an alternate interleaving ρ' may contain one or many broken read-couples (as any thread is allowed to break its read-coupling), for ease of understanding, we first consider the case which does not allow broken read-couples. This assumption will be subsequently relaxed to include the possibility of broken read couples.

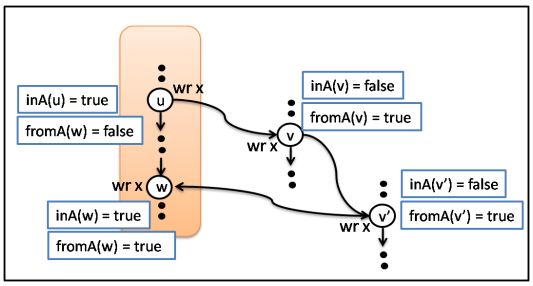

Fig. 8: The function $inA(u)$ denotes whether vertex u belongs to the atomic block, while function $fromA(u)$ denotes if u is reachable from some vertex within $\mathcal{A}$ via a violation path.

Atomic block membership condition: The violation path always begins in an atomic block. We define the following function $inA(u)$ for a vertex u.

$$inA(u) = \begin{cases} \mathsf{true} & \text{if } u \in V(\mathcal{A}) \\ \mathsf{false} & \text{otherwise} \end{cases} \tag{1}$$

Consider edge (u, v) such that v is a conflicting access in a different thread from u. In the interleaving ρ', u happens before v. Let $x(u)$ be a function that assigns an integer value to the vertex u. This is used to provide ordering constraints between vertices in the violation path. $x(u) < x(v)$ iff in ρ', u happens before v.

We define function $fromA(v)$ to be true if vertex v is reachable from $\mathcal{A}$ along a possible violation path. Let the function $edgeFromA(u, v)$ be true if v is reachable from $\mathcal{A}$ through u. There are two cases.

Case 1 ($u \in V(\mathcal{A})$): In ρ', $x(u) < x(v)$ and v should be outside the atomic block (i. e. $inA(v) = \mathsf{false}$) ensuring that the path leaves the atomic block. Then,

$$edgeFromA(u, v) = (\neg inA(v) \quad \wedge \quad (x(u) < x(v))) \quad \text{if } u \in V(\mathcal{A}) \tag{2}$$

In Fig. 8, $edgeFromA(u, v)$ is true.

Case 2 ($u \notin V(\mathcal{A})$): Vertex u happens before v and u is already on a violation path (i.e. $fromA(u) = \mathsf{true}$). Then,

$$edgeFromA(u, v) = (fromA(u) \quad \wedge \quad (x(u) < x(v))) \quad \text{if } u \notin V(\mathcal{A}) \tag{3}$$

In Fig. 8, $(x(v) < x(v'))$ and $fromA(v) = \mathsf{true}$ imply $edgeFromA(v, v') = \mathsf{true}$.

We refer to the set of eligible (u, v) pairs that need to be considered in the $edgeFromA$ computation defined above as EP_Set. There are two possibilities for edges in this set. Either, (i) the vertices u and v conflict or (ii) they belong to the same thread.

$$EP\_Set = \{(u, v) \mid ((u.type = write \vee v.type = write) \\ \wedge (u.var = v.var)) \vee (u.tid = v.tid), \text{where } u,v \in V_{RW}\}$$

Thus, combining Eq. 2 and Eq. 3 we can define function $fromA(v)$.

$$fromA(v) = \bigvee_{\forall(u,v)\in EP\_Set} edgeFromA(u,v) \tag{4}$$

4.2 Encoding of the Violation Path

Finally, there exists a violation path iff there exists at least one vertex u within $\mathcal{A}$ such that $fromA(u)$ is true implying that u is reachable from some vertex in $\mathcal{A}$ via a violation path that visits vertices outside $\mathcal{A}$. So the following condition (Φ) is satisfied if the interleaving contains a violation path.

$$\Phi_{VP} = \bigvee_{\forall u\in V(\mathcal{A})} fromA(u) \tag{5}$$

4.3 Encoding of the Program Order

For each edge $(u,v) \in E(G)$ except for the RaW edges (since the RaW edges denote read-couples which can be broken), the following constraint is introduced in the encoding.

$$HB(u,v) = (x(u) < x(v)) \tag{6}$$

Let,

$$\Phi_{PO\_Sync} = \bigwedge_{\forall(u,v)\in PO\_Sync\_Set} HB(u,v) \tag{7}$$

where,

$$PO\_Sync\_Set = \{(u,v) \mid (u,v) \in E(G) \text{ and } (u,v) \text{ is not a RaW edge.}\}$$

Moreover, for each conflicting pair (u,v) such that u may happen in parallel with v, denoted by $u \mid v$,

$$\phi_{Par}(u,v) = (x(u) < x(v)) \vee (x(v) < x(u)) \tag{8}$$

Let,

$$\Phi_{Par} = \bigwedge_{\forall(u,v)\in Par\_EP\_Set} \phi_{Par}(u,v) \tag{9}$$

where,

$$Par\_EP\_Set = \{(u,v) \mid (u,v) \in EP\_Set \text{ and } (u \mid v)\}$$

4.4 Encoding of Synchronizations

The synchronization events like lock-unlock and wait-notify also need to be encoded to get an alternate feasible interleaving. First, we encode the lock-unlocks. Let $lk_1 = (u_{lk1}, u_{unlk1})$ and $lk_2 = (u_{lk2}, u_{unlk2})$ are two pairs of vertices that operate on the same lock-variable. As the locked regions cannot overlap, there are two possibilities (1) $x(u_{unlk1}) < x(u_{lk2})$, or (2) $x(u_{unlk2}) < x(u_{lk1})$. Therefore, for each pair of lock-unlock events operating on the same lock,

$$\phi_{LK}(lk_1, lk_2) = (x(u_{unlk1}) < x(u_{lk2})) \vee (x(u_{unlk2}) < x(u_{lk1})) \tag{10}$$

Let,

$$\Phi_{LK} = \bigwedge_{\forall(lk_1,lk_2)} \phi_{LK}(lk_1, lk_2) \tag{11}$$

The wait-notify events are encoded similarly. Let, (u_w, u_n) be a wait-notify couple and u'_n be another notify event operating on the same variable. Therefore, u'_n should not come in between u_w and u_n in any interleaving. Therefore,

$$\phi_{WN}(u_w, u_n, u'_n) = (x(u'_n) < x(u_w)) \vee (x(u_n) < x(u'_n)) \tag{12}$$

Let,

$$\Phi_{WN} = \bigwedge_{\forall(u_w,u_n,u'_n)} \phi_{WN}(u_w, u_n, u'_n) \tag{13}$$

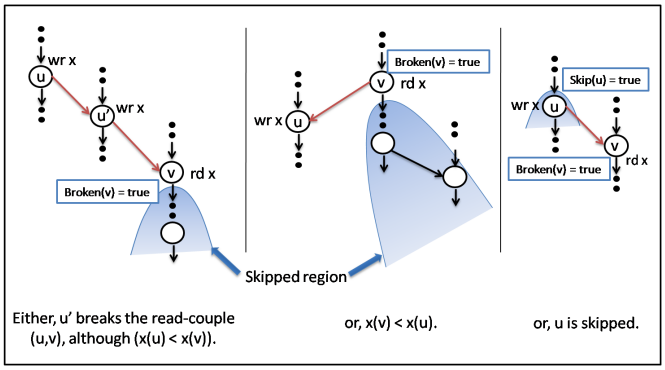

Fig. 9: Let (u, v) be a read-couple and u' writes the same variable. The read-couple can be broken in three ways in an alternate interleaving ρ'. (1) u' happens between u and v, or, (2) v happens before u, or, (3) u is skipped.

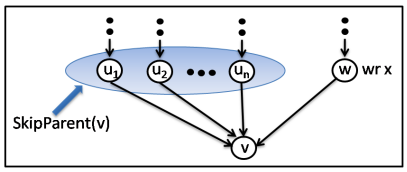

Fig. 10: Let $u_1, u_2, \ldots, u_n$ be the parents of v in G such that none of them writes. The set $\{u_1, u_2, \ldots, u_n\}$ is denoted as $SkipParent(v)$.

4.5 Encoding of Broken Read-couples

For a given trace ρ, the set $AVP(\rho)$ contains interleavings where one or many read-couples are broken. Next, we encode these broken read-couples. We introduce two new related functions - $Broken$ and $Skip$. Intuitively, whenever a read-couple is broken, $Broken(u)$ is true where u is the reader and $Skip(v)$ is true for all the following vertices v. These two functions are defined using mutual recursion. Consider the read-couple (u, v) in Figure 9. The read-couple can be broken under three circumstances.

1. Some other write event u' happens between u and v (i.e., $HB(u, u') \wedge HB(u', v)$), where $HB(u, v)$ stands for $(x(u) < x(v))$) and none of u, v and u' are skipped (i.e., $\neg(Skip(u) \vee Skip(v) \vee Skip(u'))$). We refer to u' as "challenger" as it challenges the read-couple (u, v). This condition is denoted as $\phi_{B1}(u, v)$.
2. Vertex v appears before u (i.e., $HB(v, u)$ is true) and none of u and v are skipped (i.e., $\neg(Skip(u) \vee Skip(v))$). This condition is denoted as $\phi_{B2}(u, v)$.
3. The writer u is skipped while reader v is not skipped (i.e., $Skip(u) \wedge \neg Skip(v)$). This condition is denoted as $\phi_{B3}(u, v)$.

Further, if some write u' happens before v, where v is initially not read-coupled and none of u' and v are skipped, then $Broken(v)$ is true.

Thus, we define $Broken(v)$ as follows.

$$Broken(v) = \begin{cases} \phi_{B1}(u,v) \vee \phi_{B2}(u,v) \vee \phi_{B3}(u,v) & (u,v) \text{ is a read-couple and } u' \text{ is challenger,} \\ \bigvee_{\forall u'} (HB(u',v) \wedge \neg(Skip(u') \vee Skip(v))) & v \text{ reads but not read-coupled within the TAS and } u' \text{ is a challenger,} \\ \text{false} & \text{otherwise.} \end{cases} \tag{14}$$

where,

$$\phi_{B1}(u,v) = \neg(Skip(u) \vee Skip(v)) \wedge \left(\bigvee_{\forall u'} (HB(u,u') \wedge HB(u',v) \wedge \neg Skip(u'))\right)$$
$$\phi_{B2}(u,v) = \neg(Skip(u) \vee Skip(v)) \wedge HB(v,u)$$
$$\phi_{B3}(u,v) = Skip(u) \wedge \neg Skip(v)$$

The intuitive idea behind the condition for a vertex v being skipped is as follows: (c.f. Figure 10): when any of the parents of v in G is skipped or broken, except when the parent is a conflicting write from a different thread (only possible in case of read-couples), v is also skipped. When a parent that writes in a different thread is skipped, v must be the coupled read which gets broken (but not skipped) and the events following v are skipped. The set $SkipParent(v)$ is defined as follows.

$$SkipParent(v) = \{u \mid (u,v) \in E(G) \text{ except when } u \text{ is a write from a different thread}\}$$

Next we define $Skip(v)$. We skip a vertex v when one of the members of $SkipParent(v)$ is skipped or broken.

$$Skip(v) = \begin{cases} \mathsf{false} & \text{if } SkipParent(v)=\{\} \\ \bigvee_{u \in SkipParent(v)} Skip(u) & \vee \ Broken(u) \\ & \text{otherwise.} \end{cases} \tag{15}$$

Note that although we have used mutual recursion in the definitions of $Skip$ and $Broken$, the definitions are not cyclic. The reason for this is as follows. The definition of $Skip(v)$ depends on the values of $Skip(u)$ and $Broken(u)$, where u is one of the parents of v. Therefore, the definition of $Skip$ is not cyclic. The function $Broken(v)$ is also acyclic since it refers to $Skip$ function and we have already argued that $Skip$ is acyclic.

4.6 Encoding Allowing the Broken Read-couples

We now state the modified constraints that allow the broken reads. The modifications in the constraints are underlined.

- **The atomic block membership constraints**: If a vertex within $\mathcal{A}$ is skipped, the violation path cannot start from the vertex. Therefore, we modify Eq. 1 to account for the broken read-couples in the following equation,

$$inA(u) = \begin{cases} \underline{\neg Skip(u)} & \text{if } u \in V(\mathcal{A}) \\ \underline{\mathsf{false}} & \text{otherwise} \end{cases}, \tag{16}$$

- **The encoding of the reachability of the violation path**: The function $edgeFromA$ is meaningless when either u or v is skipped. Thus, we add a new conjunctive clause $(\neg(Skip(u) \vee Skip(v)))$ to the original definition of $edgeFromA$.

$$edgeFromA(u,v) = \begin{cases} (\neg inA(v) \wedge (x(u) < x(v)) \\ \quad \wedge \underline{\neg(Skip(u) \vee Skip(v))}), \\ \qquad \underline{\text{if } u \in V(\mathcal{A})} \\ (fromA(u) \wedge (x(u) < x(v)) \\ \quad \wedge \underline{\neg(Skip(u) \vee Skip(v))}), \\ \qquad \underline{\text{otherwise}} \end{cases} \tag{17}$$

Finally, we combine Equations 5, 7, 9, 11 and 13 to get the complete encoding

$$\Phi = (\Phi_{VP} \wedge \Phi_{PO\_Sync} \wedge \Phi_{Par} \wedge \Phi_{LK} \wedge \Phi_{WN}) \tag{18}$$

where, the functions inA, $fromA$, $Broken$, $Skip$ are defined in Equations 16, 4, 14 and 15 respectively.

4.7 Complexity

The number of constraints is bounded by $O(N + mpq^2 + l_{ev}^2 l)$, where N, m, p, q, l_{ev} and l represent: number of events, number of variables, maximum number of reads per variable, maximum number of writes per variable, maximum number of events per lock variable and number of lock variables respectively.

5 Results

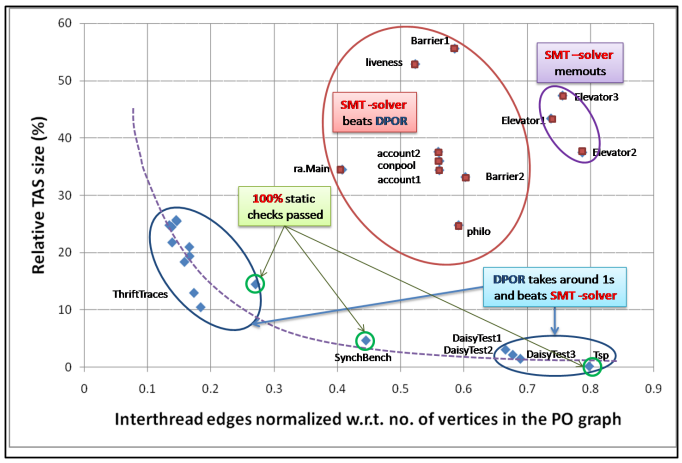

Fig. 11: It is our intuition, although not validated over large set of programs, that the winner among DPOR vs SMT can be predicted given the relative TAS size and the strength of the coupling (these can be determined statically) between the threads in a trace.

We have implemented our 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 program traces used are all available online [15]. The C++ benchmark is available online [13]. All the Java benchmarks are publicly available [3, 11, 14, 17, 24].

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 i7 machine (2.67 GHz, 3 GB memory) running Ubuntu 2.6.31-14-generic. Our experiments are designed to study how implicit interleaving enumeration using SMT compares against explicit enumeration using DPOR. As part of this, we consider two different SMT-solvers (Yices [16] & Z3 [32]). However, a fair comparison of Yices and Z3 is not possible as we can use the Yices API, but need to call Z3 using the SMT instance in a file as the Z3 API library is not available for Unix [2]. Further, we have also considered bit-blasting of the order variables (the x variables) in the SMT encoding rather than using difference logic. However, the results with bit-blasting are not significantly different from those without it.

The consolidated results of all the traces are presented in the appendix. We found that when the number of constraints generated in symbolic exploration (i.e. SMT) largely exceeds the interleavings explored in explicit exploration (i.e. DPOR), the DPOR-based strategy runs faster compared to the SMT-solvers. This observation has been validated for both the SMT solvers - Yices and Z3.

In Figure 11, we present an interesting observation made from our experimental results. We define *coupling strength* as the ratio of the inter-thread edges and the number of vertices in the graph. A low (high) number represents loosely (strongly)-coupled threads. (These indicators can be derived & generalized over all available traces.) We characterized the traces with respect to their relative TAS sizes (ratio of number of vertices within the TAS and in the entire trace) (Y-axis) and coupling strength (X-axis). We found that traces where DPOR beats SMT-solvers lie around the curve indicated in Figure 11. We classify the traces that lie further away from this curve into two sub-groups: (1) Those for which SMT-solvers beat DPOR, and (2) Those for which SMT-solvers run out of memory. Observe that, the traces belonging to sub-group 2 lie further away from the curve compared to those belonging to sub-group 1. We offer the following explanation for this observation. The curve contains traces for which one of the following is true.

1. *The strength of coupling is very low but relative TAS size is large* - due to low intensity of coupling, the number of conflicting accesses is probably very small, e.g. in `ThriftTraces` the coupling strength is between 0.1-0.3, and the relative TAS size is 10-26%. Hence, TAS+DPOR is effective for these traces.
2. *The strength of coupling is high but relative TAS size is small* - due to small relative TAS size, the possible number of interleavings is again very small, e.g. in `DaisyTest, Tsp` the strength of coupling is between 0.65-0.8 and the relative TAS size is 0-2%. Hence, once again, TAS+DPOR is effective for these traces.

However, in traces outside this curve, the coupling is such that the relative TAS size is still large, e.g. in `account, conpool` the strength of coupling is approximately 0.56 while the relative TAS size is 32-38%. In such cases, DPOR runs significantly slower than SMT-solvers. Finally, when both the coupling-strength and relative TAS size are both high the SMT-solvers run out of memory, e.g. in traces from `Elevator`, the strength of coupling is 0.7-0.8 and relative TAS size is 36-48% and SMT-solvers run out of memory. We would like to clarify that while this explanation seems to fit this limited data set, further experimentation is needed with a larger data set to draw general conclusions.

6 Conclusion

This paper builds on our previous work on predictive analysis using trace-atomicity-segments for almost-view-preserving interleavings [29]. It first provides details of an explicit search algorithm that explores possible interleavings using specialized heuristics in a DPOR based search. (This was not described in [29]). Next, it shows how this problem may be encoded as an SMT instance, thus leveraging modern SMT solvers. Finally, based on experimental evaluation, it provides some insight into the characteristics of the instances when one of these techniques is superior to the other. These characteristic can be used to predict the preferred technique for a given problem instance.

References

[1] Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: TACAS. pp. 193–207 (1999)

[2] http://research.microsoft.com/enus/um/redmond/projects/z3/ download.html: Z3: Linux binary

[3] Farchi, E., Nir, Y., Ur, S.: Concurrent bug patterns and how to test them. In: IPDPS. p. 286 (2003)

[4] Farzan, A., Madhusudan, P.: Causal atomicity. In: CAV. pp. 315–328 (2006)

[5] Farzan, A., Madhusudan, P.: Monitoring atomicity in concurrent programs. In: CAV. pp. 52–65. Springer-Verlag (2008)

[6] Farzan, A., Madhusudan, P.: The complexity of predicting atomicity violations. In: TACAS. pp. 155–169. Springer (2009)

[7] Farzan, A., Madhusudan, P.: Meta-analysis for atomicity violations under nested locking. In: CAV. pp. 248–262 (2009)

[8] Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: POPL. pp. 110–121 (2005)

[9] Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. SIGPLAN Not. 40(1), 110–121 (2005)

[10] Ganai, M.K., Gupta, A.: Accelerating high-level bounded model checking. In: ICCAD. pp. 794–801 (2006)

[11] Havelund, K.: Using runtime analysis to guide model checking of java programs. In: SPIN. pp. 245–264 (2000)
[12] Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. 12, 463–492 (July 1990)
[13] http://incubator.apache.org/thrift/:
[14] http://research.microsoft.com/qadeer/cav_issta.htm: Joint CAV/ISSTA special event on specification, verification, and testing of concurrent software
[15] http://www.princeton.edu/∼sinha/FMCAD11_Traces.zip:
[16] http://yices.csl.sri.com: Yices: An SMT solver.
[17] http://www2.epcc.ed.ac.uk/computing/research_activities/ java_grande/index_1.html: Java grande forum benchmark suite
[18] 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)
[19] Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM 21(7) (1978)
[20] Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: engineering an efficient SAT solver. In: DAC. pp. 530–535. New York, NY, USA (2001)
[21] Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P.A., Neamtiu, I.: Finding and reproducing heisenbugs in concurrent programs. In: OSDI. pp. 267–280 (2008)
[22] Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(t). J. ACM 53, 937–977 (November 2006)
[23] Papadimitriou, C.H.: The serializability of concurrent database updates. J. ACM 26(4), 631–653 (1979)
[24] von Praun, C., Gross, T.R.: Static detection of atomicity violations in object-oriented programs. Object Technology 3(6) (2004)
[25] Said, M., Wang, C., Sakalla, K., Yang, Z.: Generating data race witnesses by an SMT-based analysis. In: NFMS (2011)
[26] 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)
[27] Serbănuţă, T.F., Chen, F., Rosu, G.: Maximal causal models for multithreaded systems. Tech. Rep. UIUCDCS-R-2008-3017, UIUC
[28] Sinha, A., Malik, S.: Runtime checking of serializability in software transactional memory. In: IPDPS. pp. 1–12 (2010)
[29] Sinha, A., Malik, S., Wang, C., Gupta, A.: Predictive analysis for detecting serializability errors through trace segmentation. In: MEMOCODE (to appear) (2011), http://www.princeton.edu/∼sinha/SinhaMemocode11.pdf
[30] Sinha, N., Wang, C.: Staged concurrent program analysis. In: Foundations of Software Engineering (FSE) (2010)
[31] Sinha, N., Wang, C.: On interference abstractions. In: POPL'11. pp. 423–434 (2011)
[32] http://research.microsoft.com/en us/um/redmond/projects/z3/:
[33] Wang, C., Chaudhuri, S., Gupta, A., Yang, Y.: Symbolic pruning of concurrent program executions. In: Foundations of Software Engineering (FSE). pp. 23–32 (2009)
[34] Wang, C., Limaye, R., Ganai, M., Gupta, A.: Trace-based symbolic analysis for atomicity violations. In: TACAS. pp. 328–342. Springer (2010)
[35] Wang, C., Kundu, S., Ganai, M.K., Gupta, A.: Symbolic predictive analysis for concurrent programs. In: FM. pp. 256–272 (2009)
[36] Wang, C., Yang, Z., Kahlon, V., Gupta, A.: Peephole partial order reduction. In: TACAS. pp. 382–396 (2008)
[37] Yang, Y., Chen, X., Gopalakrishnan, G.: Inspect: A Runtime Model Checker for Multithreaded C Programs. Tech. Rep. UUCS-08-004, University of Utah (2008)
[38] Yi, J., Sadowski, C., Flanagan, C.: Sidetrack: generalizing dynamic atomicity analysis. In: PADTAD. pp. 1–10 (2009)

Appendix

This optional appendix provides the detailed experimental results that form the basis of the graphs in Section 5. In Table A1, we present the results from all the experiments on a specific trace in one horizontal slice. The columns contain the following information.

- **Column 1:** This column presents the various statistics of the logged program execution traces: number of threads (thrds), number of events (evs), number of lock events (l-evs) and variables (l-vars), number of read/write events (rw-evs) and variables (rw-vars) and number of wait-notify events (wn-evs).
- **Column 2:** This indicates the number of atomic blocks.
- **Column 3:** The average size of the TAS (absolute and relative to the full trace (%)) is given here.
- **Column 4:** This column enumerates the various scenarios explored, namely 'DPOR', 'Yices, No BB', 'Yices, BB', 'Z3, No BB' and 'Z3, BB' where BB stands for bit-blasting.
- **Column 5:** (SP) gives the number (and %) of atomic blocks for which the static check passes, i.e. no bug is possible among AVP interleavings.

A timeout of 10 minutes per atomic block is chosen for the search phase. The remaining columns report the results for the search phase.

- **Column 6:** (NV) reports the number of atomic blocks for which no violations are possible in AVP interleavings.
- **Column 7:** (V) reports the number of atomic blocks for which violations were found.
- **Column 8:** (TO) gives the number of atomic blocks for which the search for violations did not terminate within the time-limit.
- **Column 9:** (MO) gives the number of atomic blocks for which the search for violations did not terminate as the SMT-solver ran out of memory.
- **Column 10:** (Encoding Time) is the time spent in generating the encoding.
- **Column 11:** (Solving Time) is the time spent in solving by the SMT-solver or the time spent in the DPOR-based search-phase.
- **Column 12:** (#Int.(DPOR)/Constraints(SMT)) is the number of interleavings (DPOR) or constraints generated (SMT).

1. Benchmark	2. AB	3. Avg. TAS size (% of all events)	4.	5. SP (% of all) AB's)	6. NV	7. V	8. TO	9. MO	10. Encoding Time	11. Solving Time	12. #Int. (DPOR)/ Constraints (SMT)
conpool			DPOR		2	0	0	-	-	0.02s	33
thrds: 4, evs: 97			Yices, No BB	2	2	0	0	0	0.04s	0s	
l-evs: 16, l-vars: 1	4	34.5	Yices, BB	(50)	2	0	0	0	0.06s	0s	708
rw-evs: 53, rw-vars: 5		(35.56)	Z3, No BB		2	0	0	0	0s	0s	
wn-evs: 3			Z3, BB		2	0	0	0	0.01s	0s	
liveness			DPOR		8	0	2	-	-	20m	1.2M
thrds: 7, evs: 283			Yices, No BB	5	8	2	0	0	1.05s	50.81s	
l-evs: 44, l-vars: 9	15	145.8	Yices, BB	(33.3)	8	2	0	0	1.04s	50.9s	14804
rw-evs: 163, rw-vars: 12		(51.5)	Z3, No BB		8	2	0	0	0.09s	3s	
wn-evs: 6			Z3, BB		8	2	0	0	0.07s	1m13s	
SynchBench			DPOR		0	0	0	-	-	0s	0
thrds: 16, evs: 1510			Yices, No BB	124	0	0	0	0	0s	0s	
l-evs: 306, l-vars: 2	124	64.68	Yices, BB	(100)	0	0	0	0	0s	0s	0
rw-evs: 533, rw-vars: 15		(4.2)	Z3, No BB		0	0	0	0	0s	0s	
wn-evs: 0			Z3, BB		0	0	0	0	0s	0s	
Barrier1			DPOR		2	3	6	-	-	1h4s	1.9M
thrds: 10, evs: 653			Yices, No BB	0	2	5	0	4	3.21s	6m44s	
l-evs: 108, l-vars: 2	11	262.6	Yices, BB	(0)	2	5	0	4	3.22s	6m44s	32623
rw-evs: 262, rw-vars: 12		(40.2)	Z3, No BB		2	9	0	0	0.19s	20s	
wn-evs: 7			Z3, BB		2	9	0	0	0.18s	12m34s	
Barrier2			DPOR		0	2	9	-	-	1h30m	2.2M
thrds: 13, evs: 805			Yices, No BB	0	2	4	0	5	5.11s	4m25s	
l-evs: 136, l-vars: 2	11	342.27	Yices, BB	(0)	2	4	0	5	5.02s	4m32s	52549
rw-evs: 340, rw-vars: 16		(42.5)	Z3, No BB		2	9	0	0	0.24s	2m27s	
wn-evs: 7			Z3, BB		1	6	4	0	0.26s	51m10s	
account1			DPOR		0	0	10	-	-	1h40m	839K
thrds: 11, evs: 902			Yices, No BB	51	7	0	0	3	2.47s	4m2s	
l-evs: 146, l-vars: 21	61	307.6	Yices, BB	(83.6)	7	0	0	3	2.46s	1m16s	25228
rw-evs: 430, rw-vars: 42		(34.1)	Z3, No BB		7	3	0	0	0.19s	12s	
wn-evs: 10			Z3, BB		7	3	0	0	0.18s	1m2s	
account2			DPOR		0	2	19	-	-	3h10m	400K
thrds: 21, evs: 1747			Yices, No BB	100	16	0	0	5	16.2s	2m6s	
l-evs: 282, l-vars: 41	121	652.7	Yices, BB	(82.6)	16	0	0	5	16.47s	2m7s	122896
rw-evs: 850, rw-vars: 82		(37.3)	Z3, No BB		16	5	0	0	0.8s	1m50s	
wn-evs: 20			Z3, BB		16	5	0	0	0.84s	10m26s	
DaisyTest1			DPOR		1	1	0	-	-	0.1s	5
thrds: 3, evs: 2998			Yices, No BB	140	1	0	0	1	16.64s	57.74s	
l-evs: 422, l-vars: 10	142	88.5	Yices, BB	(98.6)	1	0	0	1	16.46s	1m3s	45840
rw-evs: 2003, rw-vars: 45		(2.9)	Z3, No BB		1	0	0	1	0.42s	7m33s	
wn-evs: 15			Z3, BB		1	0	1	0	0.41s	11m2s	
DaisyTest2			DPOR		2	1	0	-	-	0.1s	6
thrds: 3, evs: 4998			Yices, No BB	200	2	0	0	1	31.4s	0s	
l-evs: 699, l-vars: 14	203	88.5	Yices, BB	(98.5)	2	0	0	1	31.51s	0s	286014
rw-evs: 3330, rw-vars: 64		(1.7)	Z3, No BB		2	0	0	1	2.85s	2m10s	
wn-evs: 25			Z3, BB		2	0	0	1	2.75s	3m7s	
DaisyTest3			DPOR		2	1	0	-	-	0.1s	6
thrds: 3, evs: 7999			Yices, No BB	288	2	0	0	1	42.92s	0s	
l-evs: 1096, l-vars: 18	291	115.3	Yices, BB	(98.96)	2	0	0	1	42.91s	0s	1.08M
rw-evs: 5341, rw-vars: 81		(1.4)	Z3, No BB		2	0	0	1	10.75s	11m1s	
wn-evs: 36			Z3, BB		2	0	0	1	11.51s	5m8s	
Elevator1			DPOR		1	2	0	-	-	1.9s	2
thrds: 4, evs: 3004			Yices, No BB	180	1	0	0	2	74.9s	1.69s	
l-evs: 370, l-vars: 11	183	1306.17	Yices, BB	(98.4)	1	0	0	2	75.8s	1.72s	82650
rw-evs: 1795, rw-vars: 70		(43.4)	Z3, No BB		0	0	0	3	2.79s	11m4s	
wn-evs: 0			Z3, BB		0	0	0	3	2.81s	5m36s	
Elevator2			DPOR		6	0	0	-	-	1m35s	125
thrds: 4, evs: 5001			Yices, No BB	225	1	0	0	5	182.31s	2.95s	
l-evs: 610, l-vars: 11	231	1875.4	Yices, BB	(97.4)	1	0	0	5	182.38s	2.97s	291534
rw-evs: 3668, rw-vars: 117		(37.5)	Z3, No BB		0	0	0	6	12.16s	14m43s	
wn-evs: 0			Z3, BB		0	0	0	6	12.30s	8m24s	
Elevator3			DPOR		7	2	0	-	-	12s	206
thrds: 4, evs: 8004			Yices, No BB	553	1	0	0	8	234.23s	1.98s	
l-evs: 1128, l-vars: 11	562	3795.6	Yices, BB	(98.4)	1	0	0	8	231.31s	1.97s	1.09M
rw-evs: 5601, rw-vars: 94		(47.4)	Z3, No BB		0	0	0	9	55.33s	27m47s	
wn-evs: 0			Z3, BB		0	0	0	9	55.87s	21m17s	

1. Benchmark	2. AB	3. Avg. TAS size (% of all events)	4.	5. SP (% of all) AB's)	6. NV	7. V	8. TO	9. MO	10. Encoding Time	11. Solving Time	12. #Int. (DPOR)/ Constraints (SMT)
philo			DPOR		21	1	0	-	-	1m30s	123K
thrds: 6, evs: 1141			Yices, No BB	6	20	0	0	2	18.39s	49.83s	
l-evs: 126, l-vars: 6	28	273.5	Yices, BB	(21.43)	20	0	0	2	17.63s	50.09s	93487
rw-evs: 857, rw-vars: 23		(23.9)	Z3, No BB		20	1	1	0	0.78s	18m41s	
wn-evs: 22			Z3, BB		20	0	2	0	0.71s	28m7s	
Tsp			DPOR		0	0	0	-	-	0s	0
thrds: 4, evs: 45653			Yices, No BB	5	0	0	0	0	0s	0s	
l-evs: 20, l-vars: 5	5	97	Yices, BB	(100.0)	0	0	0	0	0s	0s	0
rw-evs: 25366, rw-vars: 42		(0.2)	Z3, No BB		0	0	0	0	0s	0s	
wn-evs: 3			Z3, BB		0	0	0	0	0s	0s	
ThriftTrace1			DPOR		4	1	0	-	-	0.04s	6
thrds: 4, evs: 2406			Yices, No BB	48	4	0	0	1	15.21s	25.15s	
l-evs: 226, l-vars: 12	53	614.2	Yices, BB	(90.6)	4	0	0	1	15.04s	25.24s	39808
rw-evs: 869, rw-vars: 62		(25.5)	Z3, No BB		4	1	0	0	0.65s	2m14s	
wn-evs: 53			Z3, BB		4	1	0	0	0.66s	7m31s	
ThriftTrace2			DPOR		3	2	0	-	-	0.04s	6
thrds: 4, evs: 2452			Yices, No BB	50	4	0	0	1	15.56s	25.18s	
l-evs: 234, l-vars: 12	55	620.7	Yices, BB	(90.9)	4	0	0	1	15.59s	25.2s	40692
rw-evs: 873, rw-vars: 62		(25.3)	Z3, No BB		4	1	0	0	0.67s	2m47s	
wn-evs: 55			Z3, BB		4	1	0	0	0.66s	7m11s	
ThriftTrace3			DPOR		1	0	0	-	-	0.1s	1
thrds: 10, evs: 4844			Yices, No BB	95	1	0	0	0	2.26s	0.12s	
l-evs: 404, l-vars: 18	96	1044.5	Yices, BB	(99)	1	0	0	0	2.24s	0.12s	10925
rw-evs: 1739, rw-vars: 129		(21.5)	Z3, No BB		1	0	0	0	0.09s	5s	
wn-evs: 91			Z3, BB		1	0	0	0	0.13s	55s	
ThriftTrace4			DPOR		11	1	0	-	-	0.1s	13
thrds: 4, evs: 6761			Yices, No BB	150	2	0	0	10	219.09s	2m52s	
l-evs: 586, l-vars: 48	162	1304.3	Yices, BB	(92.6)	2	0	0	10	222.25s	2m52s	238370
rw-evs: 2785, rw-vars: 171		(19.3)	Z3, No BB		10	0	1	1	10.17s	21m46s	
wn-evs: 125			Z3, BB		1	0	1	10	10.38s	1h5m40s	
ThriftTrace5			DPOR		4	0	0	-	-	0.4s	4
thrds: 20, evs: 8950			Yices, No BB	162	4	0	0	0	30.62s	3.6s	
l-evs: 722, l-vars: 28	166	2200	Yices, BB	(97.5)	4	0	0	0	30.49s	3.6s	143395
rw-evs: 3320, rw-vars: 223		(24.5)	Z3, No BB		4	0	0	0	1.46s	44s	
wn-evs: 165			Z3, BB		2	0	0	2	1.37s	6m37s	
ThriftTrace6			DPOR		11	1	0	-	-	0.2s	13
thrds: 4, evs: 11357			Yices, No BB	351	1	0	0	11	209.06s	0s	
l-evs: 1384, l-vars: 48	363	1191.36	Yices, BB	(96.7)	1	0	0	11	208.86s	0s	1.01M
rw-evs: 3184, rw-vars: 171		(10.5)	Z3, No BB		10	0	0	2	22.57s	24m52s	
wn-evs: 324			Z3, BB		1	0	0	11	22.65s	40m22s	
ThriftTrace7			DPOR		5	0	0	-	-	0.5s	5
thrds: 20, evs: 13561			Yices, No BB	362	0	0	0	5	154.06s	0s	
l-evs: 1520, l-vars: 28	367	2495.6	Yices, BB	(98.6)	0	0	0	5	156.85s	0s	744368
rw-evs: 3727, rw-vars: 223		(18.3)	Z3, No BB		2	0	1	2	6.45s	29m4s	
wn-evs: 372			Z3, BB		0	0	0	5	6.58s	34m28s	
ThriftTrace8			DPOR		3	0	1	-	-	10m1s	6K
thrds: 40, evs: 15975			Yices, No BB	309	3	0	0	1	82.64s	1.4s	
l-evs: 1239, l-vars: 48	313	3874.9	Yices, BB	(98.7)	3	0	0	1	82.51s	1.37s	435897
rw-evs: 5485, rw-vars: 429		(24.2)	Z3, No BB		2	0	1	1	4.14s	20m26s	
wn-evs: 259			Z3, BB		0	0	0	4	4.39s	32m41s	
ThriftTrace9			DPOR		0	0	0	-	-	0s	0
thrds: 10, evs: 16040			Yices, No BB	80	0	0	0	0	0s	0s	
l-evs: 2981, l-vars: 18	80	2322.5	Yices, BB	(100)	0	0	0	0	0s	0s	0
rw-evs: 1345, rw-vars: 107		(14.4)	Z3, No BB		0	0	0	0	0s	0s	
wn-evs: 1408			Z3, BB		0	0	0	0	0s	0s	
ThriftTrace10			DPOR		1	0	0	-	-	0.8s	1
thrds: 6, evs: 20640			Yices, No BB	501	1	0	0	0	1.28s	0.07s	
l-evs: 1724, l-vars: 158	502	4321.8	Yices, BB	(99.8)	1	0	0	0	1.24s	0.06s	4979
rw-evs: 8818, rw-vars: 519		(20.9)	Z3, No BB		1	0	0	0	0s	2s	
wn-evs: 349			Z3, BB		1	0	0	0	0s	11s	
ThriftTrace11			DPOR		1	0	0	-	-	0.9s	1
thrds: 6, evs: 25237			Yices, No BB	702	0	0	0	1	19.66s	0s	
l-evs: 2522, l-vars: 158	703	3272.5	Yices, BB	(99.9)	0	0	0	1	19.65s	0s	66806
rw-evs: 9218, rw-vars: 519		(12.9)	Z3, No BB		0	0	0	1	0s	3m49s	
wn-evs: 549			Z3, BB		0	0	0	1	0s	9m21s	

Table A1: Experimental data of the serializability violation detection. (AB=atomic blocks, SP=Static proofs, NV=Diagnosed to contain no violation after TAS failed the static check, V=Violations found, TO=Timeouts, MO=Ran out of memory, BB=Bit-blasting). Note that, AB=SP+NV+V+TO+MO.