A Method for Improving the Precision and Coverage of Atomicity Violation Predictions

Reng Zeng, Zhuo Sun, Su Liu, and Xudong He

School of Computing and Information Sciences
Florida International University
Miami, Florida 33199, USA
{rzeng001,zsun003,sliu002,hex}@cis.fiu.edu

Abstract. Atomicity violations are the most common non-deadlock concurrency bugs, which have been extensively studied in recent years. Since detecting the actual occurrences of atomicity violations is extremely hard and exhaustive testing of a multi-threaded program is in general impossible, many predictive methods have been proposed, which make error predictions based on a small number of instrumented interleaved executions. Predictive methods often make tradeoffs between precision and coverage. An over-approximate predictive method ensures coverage but lacks precision and thus may report a large number of false bugs. An under-approximate predictive method ensures precision but lacks coverage and thus can miss significant real bugs. This paper presents a post-prediction analysis method for improving the precision of the prediction results obtained through over-approximation while achieving better coverage than that obtained through under-approximation. Our method analyzes and filters the prediction results of over-approximation by evaluating a subset of read-after-write relationships without enforcing all of them as in existing under-approximation methods. Our post-prediction method is a static analysis method on the predicted traces from dynamic instrumentation of C/C++ executable, and is faster than dynamic replaying methods for ensuring precision.

1 Introduction

Multi-threaded programs are prone to bugs due to concurrency. Concurrency bugs are hard to find and reproduce due to the large number of interleavings. Most non-deadlock concurrency bugs are atomicity violation bugs due to the unprotected accesses of shared variables by multiple threads. Existing approaches for detecting atomicity violation can be static or dynamic. Static approaches [6] usually suffer from a large number of false positives due to concurrency and pointer aliasing. Dynamic approaches include monitor based methods that require atomicity violations to manifest during monitored runs [11][5][19], and predictive methods that explore atomicity violations in alternative interleavings extracted from some sample instrumented runs [17][18][4][1].

Predictive methods use either (1) under-approximate models ([15][16][1][14]): the set of extracted interleavings with the exact same read-after-write relationships as in the instrumented runs, which are a subset of all feasible interleavings;

or (2) over-approximate models ([21][17][8][7][20]): the set of all possible interleavings extracted from the instrumented runs, which may not be feasible in the original program due to data constraints and ad-hoc synchronization. Hence predictive methods based on under-approximate models have inadequate coverage and predictive methods based on over-approximate models lack precision. Many predictive methods mentioned above made tradeoffs between precision and coverage. Figure 1 shows the relationships between various predictive methods in terms of precision and coverage. Although precise coverage captures the exact bugs in a multi-threaded program and thus is ideal, it cannot be achieved practically. Our method AVFilter and replaying methods are not independent predictive methods, but post-prediction analysis methods. A replaying method can eliminate false bugs, but incurs heavy runtime overhead and may not be able to produce the exact same execution sequence as an instrumented run due to nondeterminism [2][9][10].

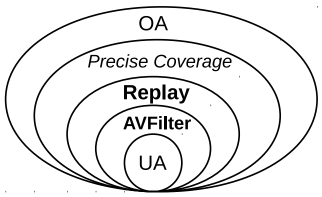


Fig. 1. Relationships of predictive methods on coverage and precision, in which a larger circle shows more coverage and the circles within the precise coverage do not contain false predictions.
UA - Under-approximate methods [15][16][1][14].
OA - Over-approximate methods [21][17][8][7][20], e.g. Figures 2, and 6.
AVFilter - Post-prediction analysis method in this paper.
Replay - Methods in [17] of rescheduling violation traces predicted by OA methods
Precise Coverage - Precise coverage [18] captures the exact bugs in a multi-threaded program.

This paper presents a post-prediction analysis method AVFilter for improving the precision of the prediction results obtained through over-approximation while achieving better coverage than that obtained from under-approximation methods. The method checks and filters the results of over-approximation by evaluating a subset of critical read-after-write relationships without enforcing all of them as in under-approximation methods.

2 Preliminaries

A multi-threaded program P has a set of threads and a set of shared variables. An instrumented execution $\sigma = s_1, ..., s_n$ of P is a sequence of executed

Table 1. Limited coverage of prediction using under-approximate (UA) models for two threads (Superscript denotes the thread number T1 or T2)

	Observed Execution	Possible Alternative Execution	Description of Unserializability or Missed Reason
Covered	$R^1R^1W^2$	$R^1W^2R^1$	Two reading accesses read from different writes
	$R^1W^1W^2$	$R^1W^2W^1$	Forwarded writing access in T2 is overwritten
	$W^1W^1W^2$	$W^1W^2W^1$	Forwarded writing access in T2 is overwritten
	$R^2W^1W^2$	$W^1R^2W^2$	An intermediate value is read
	$W^2W^1W^1$	$W^1W^2W^1$	Forwarded writing access in T1 is overwritten
Missed	$W^1R^1W^2$	$W^1W^2R^1$	Intra-thread read-after-write in T1 prohibits interleaved writing in T2
	$W^1W^1R^2$	$W^1R^2W^1$	Inter-thread read-after-write prohibits forwarded reading in T2
	$W^2R^1R^1$	$R^1W^2R^1$	Inter-thread read-after-write prohibits forwarded reading in T1
	$W^2W^1R^1$	$W^1W^2R^1$	Intra-thread read-after-write in T1 prohibits interleaved writing in T2
	$W^2R^1W^1$	$R^1W^2W^1$	Inter-thread read-after-write prohibits forwarded reading in T1

statements. A trace is the projection of an execution to a sequence of annotated shared variable accesses and synchronization events. Formally, a trace, $\tau = e_1, ..., e_m$ is a sequence of events where each event $e_i (1 \leq i \leq m)$ is a tuple $\langle seq_i, tid_i, action_i, br_i \rangle$ in which seq_i is an increasing sequence number, tid_i is a thread handle, $action_i$ is either an atomic shared variable access or a synchronization event, and br_i is the number of branches between e_i and its immediate preceding event within the same thread. Given a trace $\tau = e_1, ..., e_m$, a partial order thread model $(E_\tau, \prec)$ can be defined, where E_τ is the set of events occurring in τ and $\prec$ is a causal relation on E_τ. The causal relation $\prec$ respects all constraints of synchronization primitives and program orders within individual threads. Thus $(E_\tau, \prec)$ captures a set of alternative interleaving traces derived from the original trace τ. A trace τ' in $(E_\tau, \prec)$ is feasible if and only if it is a projection of a feasible execution σ' of P. The strength of the causal relation $\prec$ affects the number of possible interleaved traces in $(E_\tau, \prec)$.

Definition 1. *(Under-approximate models) When the exact same read-after-write relation on all shared variables in τ is enforced in $\prec$, every trace $\tau' \in (E_\tau, \prec)$ is feasible. Such a partial order thread model $(E_\tau, \prec)$ is called under-approximate.*

Definition 2. *(Over-approximate models) When not the exact same read-after-write relation on all shared variables in τ is enforced in $\prec$, some trace $\tau' \in (E_\tau, \prec)$ may not be feasible. Such a partial order thread model $(E_\tau, \prec)$ is called over-approximate.*

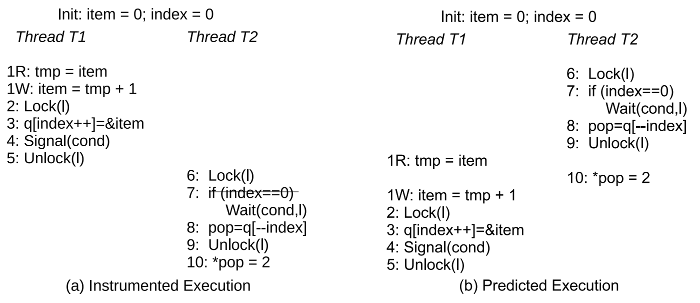
An atomicity violation bug is caused by a broken order of accesses to a shared variable x within one thread by another thread. Most atomicity violation bugs involve only three accesses to a shared variable within two threads based on the study in [11], in which 101 out of 105 bugs involved only two threads. Thus existing methods for atomicity violation detection and prediction work on every shared variable within every pair of threads incrementally. Furthermore, these methods assume sequential consistent memory such that the logic order of the program execution is respected in physical machine execution. Table 1 shows ten interleaving scenarios of three accesses to a shared variable between two threads that result in atomicity violations, among which only five can be predicted by methods using under-approximate models while other five are missed due to some broken read-after-write relationship within three accesses. Our method can predict each of scenarios above.

3 AVFilter: Performing Post-prediction Static Analysis

AVFilter works on over-approximate models from a given trace τ to remove false predictions so that the remaining atomicity violation predictions are all feasible. These remaining feasible atomicity violation predictions cover all the predictions obtained from predictive methods using under-approximate models of τ. This analysis method is general and is applicable to the prediction results from many existing predictive methods using over-approximate models. The only information needed is an instrumented trace τ and three memory accesses in τ that forms an atomicity violation pattern [21][18].

3.1 Data Constraints Causing False Predictions

Data constraints are data dependencies that can make a predicted atomicity violation trace infeasible. Typical data constraints include branch conditions dependent on shared variables and queue accesses dependent on shared indexing variables. In real-world applications, data dependency can be quite complicated and appear in various obscure ways. Figure 2 shows a reformatted code snippet from Apache web server, which gives an example of data constraints. Figure 2(a)



```
        Init: item = 0; index = 0
 Thread T1                Thread T2

1R: tmp = item
1W: item = tmp + 1
2: Lock(l)
3: q[index++]=&item
4: Signal(cond)
5: Unlock(l)
                          6:  Lock(l)
                          7:  if ~~(index==0)~~
                                Wait(cond,l)
                          8:  pop=q[--index]
                          9:  Unlock(l)
                          10: *pop = 2
```

(a) Instrumented Execution

```
        Init: item = 0; index = 0
 Thread T1                Thread T2

                          6:  Lock(l)
                          7:  if (index==0)
                                Wait(cond,l)
                          8:  pop=q[--index]
                          9:  Unlock(l)
1R: tmp = item
                          10: *pop = 2
1W: item = tmp + 1
2: Lock(l)
3: q[index++]=&item
4: Signal(cond)
5: Unlock(l)
```

(b) Predicted Execution

Fig. 2. A false positive due to a data constraint (reformatted code snippet from Apache web server)

shows a trace of an instrumented execution, in which shared variable *index* is read in line 7 and line 8 after its writing in line 3, and hence there are data dependencies in two pairs of accesses to *index*: line 3 and line 7, line 3 and line 8. Figure 2(b) shows a trace with a predicted atomicity violation pattern in which line 10 has a writing access to the shared memory *item* in Thread T2 between the reading access (line 1R) and the writing access (line 1W) in Thread T1. However, both pairs of accesses to *index* above are broken, which make the memory access in line 10 in the predicted trace infeasible.

A solution to deal with data constraints requires a precise and complete partial order thread model extracted from an instrumented trace. The precision ensures the feasibility of any predicted atomicity violation in the partial order thread model, and the completeness requires any feasible atomicity violation be captured in the partial order thread model. Enforcing all the exact read-after-write relationships of the instrumented trace in the partial order thread model can ensure the precision of the partial order thread model. Several methods [16][14] introduced the exact read-after-write relationships as a simple solution to ensure the precision. However, the data constraints imposed by the exact read-after-write relationships are too strong, thus make the resulting partial order thread model over restrictive and under-approximate.

3.2 Problem Formulation

During post-prediction analysis, any predicted atomicity violation trace is an alternative interleaving respecting the same causal relations imposed by the synchronization events as the original instrumented trace. Thus we can view a trace as a sequence of atomic (reading or writing) accesses without synchronization events to simplify the discussion. Let $\tau = a_1^{t_1}, a_2^{t_2}, ..., a_n^{t_n}$ be a sequence of atomic accesses to share variables in an interleaved execution of two threads, in which a superscript indicates the thread an event belongs to, thus $t_i \in \{1, 2\}$ for $1 \leq i \leq n$; and a subscript indicates the occurrence position of an event in the interleaving trace.

Over-approximate predictive methods in [21][11][17] are based on three-access atomicity violation patterns. Table 1 gives all possible atomicity violation patterns after reordering the event in thread 2 to occur between the two events in thread 1. Although the above methods check only a pair of threads, they are applicable to many threads by checking every pair of threads on every shared variable one at a time.

A predicted atomicity violation trace is $\tau' = ..., a_{i'}^1, ..., a_{j'}^2, ..., a_{k'}^1, ...$ with atomicity violation pattern $a_{i'}^1, a_{j'}^2, a_{k'}^1$ which are three consecutive accesses to a shared variable x. τ' is the result of reordering some accesses in a given original instrumented trace τ where (1) $\tau = ..., a_i^1, ..., a_k^1, ..., a_j^2, ...$ or (2) $\tau = ..., a_j^2, ..., a_i^1, ..., a_k^1, ...$ Events $a_{i'}^1, a_{j'}^2, a_{k'}^1$ in τ' correspond to the exact same events a_i^1, a_j^2, a_k^1 in τ. Accesses other than $a_{i'}^1, a_{j'}^2, a_{k'}^1$ are not explicitly identified in τ' but may also be reordered due to the reordering of a_i^1, a_j^2, a_k^1 in τ. τ' may not be feasible due to the violation of some read-after-write relationship in

τ. τ' is feasible if its prefix up to $a^1_{k'}$ is feasible since anything happens after $a^1_{k'}$ does not affect the feasibility of τ'.

3.3 Our Method

Our method works on the predicted traces that already contained all synchronization information. The underlying idea of our method is to check whether any reordered event inside the violation pattern $a^1_{i'}, ..., a^2_{j'}, ..., a^1_{k'}$ in τ' can break some critical read-after-write relationship inside the subsequence $a^1_i, ..., a^1_k, ..., a^2_j$ in situation (1) or the subsequence $a^1_j, ..., a^1_i, ..., a^2_k$ in situation (2) in the instrumented trace τ. Before reordering, a^2_j may happen after a^1_k as in situation (1), or before a^1_i as in situation (2). Let $a \dashrightarrow b$ denote event a occurs before event b, We explain our checking idea for situation (1), i.e. $a^1_i \dashrightarrow a^1_k \dashrightarrow a^2_j$ (the checking idea for situation (2) is similar). In Figures 3, 4 and 5, w and r are used to describe a read-after-write relationship with regard to either a different shared variable or the same shared variable accessed in $a^1_{i'}, a^2_{j'}, a^1_{k'}$. In Figure 3, a reading event r^2 is moved forward due to the reordering of a^2_j, thus breaking the read-after-write relationship between w^1 and r^2.

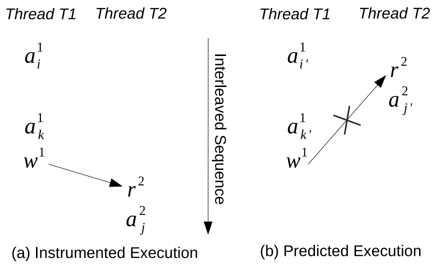


Fig. 3. Read-after-write relationship is broken, assuming $a^1_i \dashrightarrow a^1_k \dashrightarrow a^2_j$ and a moved forward reading event r^2 before $a^1_{k'}$, $r^2 \in \tau(a^1_k, a^2_j)$

In Figures 4 and 5, $Prev(a^2_j)$ denotes the immediate preceding access to the same shared variable as a^2_j. In Figure 4, due to the reordered $a^1_{i'}, a^2_{j'}, a^1_{k'}$, $Prev(a^2_j)$ is moved forward to happen before $a^1_{i'}$, thus r^2 is moved forward to happen before w^1, causing the breaking of the read-after-write relationship between w^1 and r^2.

In Figure 5, due to the reordered $a^1_{i'}, a^2_{j'}, a^1_{k'}$, $Prev(a^2_j)$ is moved forward to happen before $a^1_{i'}$, thus w^2 is moved forward to happen before r^1, causing the breaking of the read-after-write relationship between r^1 and its original defining writing access.

Lemmas 1 and 2 identify all the cases in which a reordered event may affect the feasibility of τ'. Let $\tau(a, b)$ be accesses in τ that occur after a and before b, $\tau[a, b)$ be accesses in $\tau(a, b)$ including a, and $\tau(a, b]$ be accesses in $\tau(a, b)$

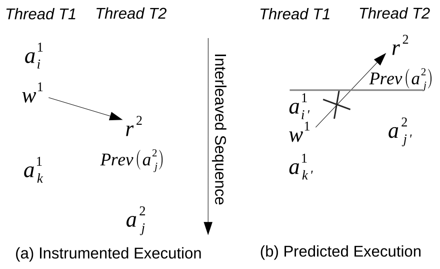


Fig. 4. Read-after-write relationship is broken, assuming $a_i^1 \dashrightarrow a_k^1 \dashrightarrow a_j^2$ and a moved forward reading event before $a_{i'}^1$,$r^2 \in \tau(a_i^1, Prev(a_j^2))]$

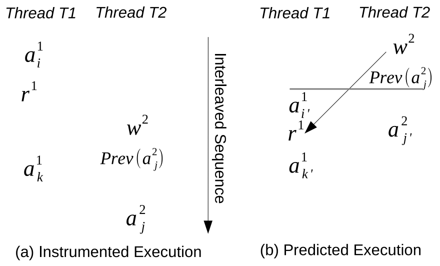


Fig. 5. Read-after-write relationship is broken, assuming $a_i^1 \dashrightarrow a_k^1 \dashrightarrow a_j^2$ and a moved forward writing event w^2, $w^2 \in \tau(a_i^1, Prev(a_j^2))]$

including b, $Prev(a^i)$ denote the immediate preceding atomic access to the same shared variable as a in thread i, and $Next(a^i)$ denote the immediate succeeding atomic access to the same shared variable as a in thread i.

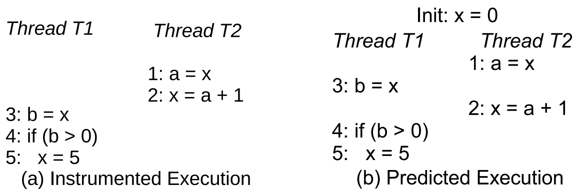
Lemma 1. *Given a predicted atomicity violation trace $\tau' = ..., a_{i'}^1, ..., a_{j'}^2, ...,$ $a_{k'}^1, ...$ with atomicity violation pattern $a_{i'}^1$,$a_{j'}^2, a_{k'}^1$ with regard to a shared variable x, and the original instrumented trace $\tau = ..., a_i^1, ..., a_k^1, ..., a_j^2, ...$ τ' can be infeasible due to a violated data constraint caused by only one of the following cases (1) a moved forward reading event in thread 2: $r^2 \in \tau(a_k^1, a_j^2)$ and $r^2 \dashrightarrow a_{k'}^1$; (2) a moved forward reading event in thread 2: $r^2 \in \tau(a_i^1, Prev(a_j^2))]$ and $r^2 \dashrightarrow a_{i'}^1$; or (3) a moved forward writing event in thread 2: $w^2 \in \tau(a_i^1, Prev(a_j^2))]$, $w^2 \dashrightarrow a_{i'}^1$ and the existence of a branch instruction between $\tau[a_i^1, a_k^1)$.*

The proof of the lemma is omitted due to limited space. Note a moved forward writing event in thread 2: $w^2 \in \tau[Prev(a_j^2), a_k^1)$ and $w^2 \dashrightarrow a_{k'}^1$ can break some read-after-write relationship after $a_{k'}^1$, but does not affect the feasibility of τ'.

Figure 2 shows an example of case (1) in Lemma 1, where the predicted atomicity violation trace τ' in (b) is infeasible. In (b) line 1R is $a_{i'}^1$, line 10 is $a_{j'}^2$, line 1W is $a_{k'}^1$, and line 7 is the moved forward reading r. The read-after-write relationship with line 3 is broken. As a result, the condition in line 7 is true and $Wait$ is executed that makes τ' infeasible.

Lemma 2. *Given a predicted atomicity violation trace* $\tau' = ..., a^1_{i'}, ..., a^2_{j'}, ..., a^1_{k'}, ...$ *with atomicity violation pattern* $a^1_{i'}$,$a^2_{j'}$, $a^1_{k'}$ *with regard to a shared variable x, and the original instrumented trace* $\tau = ..., a^2_j, ..., a^1_i, ..., a^1_k, ...$ τ' *can be infeasible due to a violated data constraint caused by only one of the following cases (1) a moved forward reading event in thread 1:* $r^1 \in \tau(a^2_j, a^1_i]$ *and* $r^1 \dashrightarrow a^2_{j'}$*; (2) a moved forward reading event in thread 1:* $r^1 \in \tau(Next(a^2_j), a^1_k)]$, $r^1 \dashrightarrow Next(a^2_{j'})$*, and the existence of some branch instruction between* $\tau[a^1_i, a^1_k)$*.*

The proof of this lemma is omitted due to limited space. Note any moved forward writing event in thread 1 does not affect the feasibility of τ'.



```
Thread T1       Thread T2
                1: a = x
                2: x = a + 1
3: b = x
4: if (b > 0)
5:   x = 5
(a) Instrumented Execution
```

```
          Init: x = 0
Thread T1       Thread T2
                1: a = x
3: b = x
                2: x = a + 1
4: if (b > 0)
5:   x = 5
(b) Predicted Execution
```

Fig. 6. A false positive due to local dependency

Figure 6 shows an example of case (1) in Lemma 2, where the predicted atomicity violation trace τ' in (b) is infeasible. In (b), line 3 is $a^1_{i'}$, line 2 is $a^2_{j'}$, line 5 is $a^1_{k'}$, and line 3 is the moved forward reading $r^1 \in \tau(a^2_j, a^1_i]$. The broken read-after-write relationship from line 2 now reads a new value 0 in line 3. As a result, $b^1_{k'}$ cannot be executed and thus τ' is infeasible.

Our method is realized in the following Algorithm 1. An instrumented trace contains a sequence of events, and each event is defined by a thread identifier *tid*, a memory access type (read or write) *rw*, a shared variable *var*, and the number *br* of branches between this event and its immediate preceding event within the same thread. Other fields in an instrumented trace are omitted here without affecting the post-prediction analysis. An atomicity violation prediction is based on an atomicity violation pattern $a^1_{i'}$, $a^2_{j'}$, $a^1_{k'}$ involving two threads 1 and 2. The algorithm analyzes the feasibility of a predicted violation according to Lemmas 1 and 2. The complexity is linear to the size of trace, and note the algorithm only needs to check the subsequence containing the three accesses a^1_i, a^2_j, a^1_k. Five true returns in the algorithm correspond to the five cases in Lemmas 1 and 2.

3.4 Comparison with Precise Coverage and the Coverage of Under-Approximate Methods

As shown in Figure 1, the coverage of our method AVFilter is a subset of the precise coverage and a superset of the coverage of under-approximate methods.

Algorithm 1. Algorithm of post-prediction analysis

Input: τ : $seq \rightarrow (tid_{seq}, rw_{seq}, var_{seq}, br_{seq})$, and three seq: $...a_i^1...$, $...a_j^2...$, $...a_k^1...$ that contain accesses relevant to a violation pattern $a_{i'}^1$, $a_{j'}^2$, $a_{k'}^1$ in τ'.
Output: Whether a predicted violation maybe infeasible.

```
1: if $a_j^2 > a_i^1$ then
2:     $prev \leftarrow max(seq)$ where $tid_{seq} = 2 \wedge var_{seq} = var_{a_j^2} \wedge seq < a_j^2$
3:     for $r \in (a_i^1, prev] \cup (a_k^1, a_j^2) \wedge rw_r = read \wedge tid_r = 2$ do
4:         $w = max(seq)$ where $rw_{seq} = write \wedge var_{seq} = var_r \wedge seq < r$
5:         if $r \in (a_i^1, prev] \wedge w > a_i^1 \wedge tid_w = 1$ then
6:             return $True$
7:         end if
8:         if $r \in (a_k^1, a_j^2) \wedge w > a_k^1 \wedge tid_w = 1$ then
9:             return $True$
10:        end if
11:    end for
12:    for $r \in [a_i^1, a_k^1) \wedge rw_r = read \wedge tid_r = 1$ do
13:        $w = min(seq)$ where $rw_{seq} = write \wedge var_{seq} = var_r \wedge seq > r \wedge tid_w = 2$
14:        if $w \leq prev \wedge \exists seq \centerdot (r < seq < a_k^1) \wedge (tid_{seq} = 1) \wedge br_{seq} > 0$ then
15:            return $True$
16:        end if
17:    end for
18: end if
19: if $a_j^2 < a_i^1$ then
20:    for $r \in (a_j^2, a_i^1] \wedge rw_r = read \wedge tid_r = 1$ do
21:        $w = max(seq)$ where $rw_{seq} = write \wedge var_{seq} = var_r \wedge seq < r$
22:        if $w \geq a_j^2 \wedge tid_w = 2$ then
23:            return $True$
24:        end if
25:    end for
26:    $next \leftarrow min(seq)$ where $tid_{seq} = 2 \wedge var_{seq} = var_{a_j^2} \wedge seq > a_j^2$
27:    for $r \in (a_i^1, a_k^1) \wedge rw_r = read \wedge tid_r = 1$ do
28:        $w = max(seq)$ where $rw_{seq} = write \wedge var_{seq} = var_r \wedge seq < r \wedge tid_w = 2$
29:        if $w > next \wedge \exists seq \centerdot (r < seq < a_k^1) \wedge (tid_{seq} = 1) \wedge br_{seq} > 0$ then
30:            return $True$
31:        end if
32:    end for
33: end if
34: return $False$
```

No False Positives with Better Coverage than Under-Approximate Methods. Lemmas 1 and 2 define the necessary conditions that a violated data constraint can cause a predicted atomicity violation trace infeasible. Thus Lemmas 1 and 2 have ensured that any surviving predicted atomicity violation trace is a feasible one, assuming a predictive method such as [21] preserved all control constraints. Under-approximate methods ensure precision by eliminating all traces breaking any read-after-write relationships. Our post-prediction analysis method ensures precision while eliminating only a subset of predicted atomicity violation traces breaking certain read-after-write relationships in the original instrumented trace.

Potential False Negatives. The false prediction shown in Figure 6 becomes a real one if the initial value of variable x is changed to 1, which is treated as infeasible by our method. The predicted traces often only contain the access information on shared variables while omitting the access information of local variables and the potential dependencies between shared and local variables. It is not possible to determine with certainty whether a trace with some broken read-after-write relationship is feasible or not without exploring the complex inter-variable dependencies in the actual program. Therefore, we treat those traces containing broken read-after-write relationship as infeasible to ensure the soundness of our method.

4 Experiments and Evaluation

We have implemented the proposed algorithm in a prototype tool based on the tool in [21] and conducted several experiments on a PC with dual core 2.33GHz CPU and 2GB memory. In the following subsections, we show the benefits of AVFilter in terms of improving precision, ensuring coverage, and achieving scalability.

Improving Precision

We evaluate our algorithm using as many benchmarks as available from the existing state of the art works [11][17][18]. [11] uses C based Apache web server and C++ based MySQL database, [17] uses small Java programs and Java based Apache ftp server, and [18] uses a few small C programs. We first run our tool [21] on three C/C++ benchmark programs to obtain predictive atomicity violations. Since our tool [21] implements an over-approximate method, the number of predicted atomicity violations should be representative in other over-approximate methods such as [11][17]. We then run AVFilter on the predicted atomicity violations obtained from [21] to eliminate potential false positives.

The experiment results using Apache web server, FFmpeg, and MySQL database are shown in Table 2, and the experiment results using the benchmarks in [18] are shown in Table 3. The experiment result of Apache ftp server of [17] is listed in the table for comparison purpose that shows our method is

Table 2. Experimental results using real world programs

	Program Size	Events in Trace	OA	AVFilter	AVFilter-time	Replay-time
Apache web server 2.0.48 (C)	1.5 MB	140532	155	1	12.1 sec	-
Apache ftp server (Java) [17]	53 KB	-	109	-	-	2.27 hrs
FFmpeg 2.0 (C)	41 MB	550352	29	0	11.6 sec	-
MySQL 4.0.12 (C++)	7 MB	3273281	5202	1	4322 sec	-

much faster than using replaying methods to achieve precision. Our algorithm has shown tremendous improvement on prediction precision.

In Table 2, both Apache web server and MySQL have a known atomicity violation bug but FFmpeg does not. The first column *Program Size* gives the size of the executable, the second column *Events in Trace* lists the number of events in the instrumented trace; the third column *OA* contains the number of predicted atomicity violations using the over-approximate method in [21]; the fourth column *AVFilter* is the number of predicted atomicity violations after the post-prediction analysis using AVFilter; the fifth column *AVFilter-time* is the time in seconds to perform post-prediction analysis; the last column shows the replaying time in hours to replay all predictions.

Ensuring Coverage

In Table 3, Programs *atom001* and *atom002* have atomicity violations that are extracted from a real bug [12]. Their modified versions without atomicity violations are *atom001a* and *atom002a*. Other programs are Linux/Pthreads/C implementation of the parameterized bank example [3], in which program *bank-av-8* has atomicity violations; program *bank-sav-8* adds a condition variable as a partial fix without avoiding all atomicity violations for any shared variable; and program *bank-nav-8* adds a transaction lock to remove all atomicity violations. The first three columns provide the statistics of programs, in which *svars-causing-av* is the number of shared variables causing predicted atomicity violations. The next two columns provide the statistics of our method, which uses the results of an over-approximate method in [21]. *OA-svars* is the number of shared variables causing predicted atomicity violations using over-approximate methods, *AVFilter-svars* is the number of shared variables causing predicted atomicity violations after post-prediction analysis using AVFilter. Note that a single shared variable may generate many possible atomicity violation traces, which can often be eliminated by a single fix. We count shared variables in *AVFilter-svars* that have at least one feasible predicted violation trace. The last column *UA-avs* is the number of predicted atomicity violation traces generated by under-approximate methods that enforce the exact same read-after-write relation on all shared variables in an instrumented trace.

One shared variable in *atom002* is missed due to read-after-write relationships on other shared variables. Our method cannot decide whether it is feasible because the value of a shared variable or a local variable depends on the value

Table 3. Experimental results on precision and coverage using the benchmark in [18]

Program			Our method		UA methods
name	threads	svars-causing-av	OA-svars	AVFilter-svars	UA-avs
atom001	3	1	1	1	0
atom001a	3	0	1	0	0
atom002	3	1	1	0	0
atom002a	3	0	1	0	0
bank-av-8	9	8	8	8	0
bank-sav-8	9	8	8	8	0
bank-nav-8	9	0	8	0	0

of another shared variable. [18] collects and encodes all program information in CTP and thus can detect it. From the experiment results in Table 3, It can be seen that under-approximate methods miss feasible atomicity violations.

Achieving Scalability

We compare the running time to CTP [18] in Figure 7, based on statistics provided in [18] as CTP is not publicly available.

The running times, under negligible hardware differences, in Figure 7 show that our method's scalability is promising compared to those of the symbolic method CTP [18]. When the size of programs grows, e.g. *bank-nav-8* containing more code than others, the formulas built in CTP also grow bigger and require more time to be solved. Our method stops as soon as a broken read-after-write relationship defined in Lemmas 1 or 2 is detected, and incurs insignificant time increase when the size of a program grows and thus can handle much larger programs.

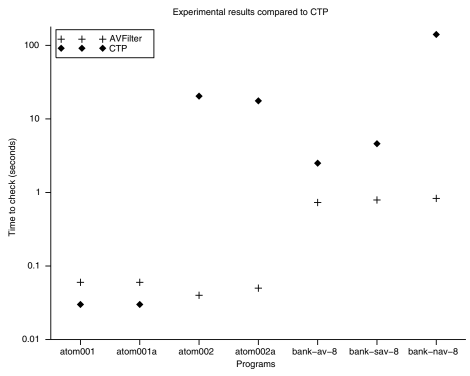


Fig. 7. Performance compared to CTP [18]

Our method is also evaluated using the complete Apache web server, MySQL database and FFmpeg audio/video codec library, as shown in Table 2, and is applicable to large scale programs.

5 Related Works

Predictive methods based on under-approximate models such as [16][1][14] admit only interleaving traces with the exact same read-after-write relations on all shared variables as in the instrumented executions to achieve precision; however, the constraints imposed by the read-after-write relations are too strong, which make the derived partial order thread models over restrictive and thus exclude many feasible alternative interleaving traces. Predictive methods based on over-approximate models such as [21][17][8][7][20] admit not only all feasible interleaving traces but also infeasible interleaving traces due to data constraints and ad-hoc synchronization, and thus can make imprecise false predictions. [15] allows broken read-after-write relations but prohibits the thread with such a read event to continue, hence can be considered as using under-approximate model.

CTP [18] is an analysis tool applicable to the predicted atomicity violation traces generated by over-approximate methods, thus is the most relevant work to ours. CTP achieves precision and complete coverage by using the values of shared variables and local variables in the predicted atomicity violation trace, which requires heavy instrumentation and the static analysis of the complete source code. Our method explores ways to improve precision and to ensure coverage while avoiding heavy instrumentation and the static analysis of source code.

Some tools use replaying methods to ensure precision. Penelope [17] instruments the scheduler to follow a predicted schedule, from which it gets a set of threads and the number of steps that each thread should take before the next context switch. Only after execution reaches the point of the violation pattern, the scheduler releases all threads to their normal execution. Before the execution reaches the violation point, it incurs the same overhead as an instrumented execution, in addition to the overhead of instrumenting scheduler. CHESS [13] is a systematic and deterministic testing tool for concurrent programs, which takes complete control over scheduling of threads; however, its scheduler is non-preemptive and therefore cannot model the behavior of a real scheduler that may preempt a thread at any point during its execution. Following the exact same schedule of a predicted atomicity violation trace still cannot guarantee perfect replaying since perfect replaying is impossible without capturing all sources of nondeterminism, as demonstrated in [2][9][10].

6 Conclusion

Predictive methods for atomicity violations need to consider the tradeoffs between precision and coverage. This paper presents a post-prediction analysis method AVFilter to improve the precision of predicted atomicity violation traces generated from over-approximate methods and to achieve better coverage than that obtained from under-approximate methods. AVFilter covers all ten scenarios in Table 1. AVFilter is general and is applicable to the prediction results from many existing predictive methods using over-approximate models. AVFilter does not rely on the instrumentation of local variables and the analysis of source code, and thus is scalable and applicable to large programs.

Acknowledgements. This work was partially supported by the NSF of U.S. under award HRD-0833093. Reng has also been supported by a Dissertation Year Fellowship of Florida International University.

References

1. Chen, F., Serbanuta, T.F., Rosu, G.: jPredictor: a predictive runtime analysis tool for java. In: Proceedings of the 30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, pp. 221–230 (2008)
2. Dunlap, G.W., King, S.T., Cinar, S., Basrai, M.A., Chen, P.M.: ReVirt: enabling intrusion analysis through virtual-machine logging and replay. In: Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI 2002), Boston, MA, USA, pp. 211–224 (2002)
3. Farchi, E., Nir, Y., Ur, S.: Concurrent bug patterns and how to test them. In: Proceedings of the 17th International Symposium on Parallel and Distributed Processing, IPDPS 2003 (2003)
4. Farzan, A., Madhusudan, P.: The complexity of predicting atomicity violations. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 155–169. Springer, Heidelberg (2009)
5. Flanagan, C., Freund, S.N., Yi, J.: Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. In: Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2008), Tucson, AZ, USA, pp. 293–303 (2008)
6. Flanagan, C., Qadeer, S.: A type and effect system for atomicity. In: Proceedings of the 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2003), San Diego, CA, USA, pp. 338–349 (2003)
7. Ganai, M.K.: Scalable and precise symbolic analysis for atomicity violations. In: Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, pp. 123–132 (2011)
8. Kahlon, V., Wang, C.: Universal causality graphs: A precise happens-before model for detecting bugs in concurrent programs. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 434–449. Springer, Heidelberg (2010)
9. Konuru, R., Srinivasan, H., Choi, J.D.: Deterministic replay of distributed java applications. In: Proceedings of 14th International Parallel and Distributed Processing Symposium (IPDPS 2000), Cancun, Mexico, pp. 219–227 (2000)
10. Liu, X., Lin, W., Pan, A., Zhang, Z.: WiDS checker: combating bugs in distributed systems. In: Proceedings of the 4th USENIX Conference on Networked Systems Design and Implementation (NSDI 2007), Cambridge, MA, USA, pp. 19–19 (2007)
11. Lu, S., Park, S., Zhou, Y.: Finding Atomicity-Violation bugs through unserializable interleaving testing. IEEE Transactions on Software Engineering 38(4), 844–860 (2011)

12. Lu, S., Tucek, J., Qin, F., Zhou, Y.: AVIO: detecting atomicity violations via access interleaving invariants. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2006), San Jose, CA, USA, pp. 37–48 (2006)
13. Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P.A., Neamtiu, I.: Finding and reproducing heisenbugs in concurrent programs. In: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI 2008), San Diego, CA, USA, pp. 267–280 (2008)
14. Sen, K., Roşu, G., Agha, G.: Detecting errors in multithreaded programs by generalized predictive analysis of executions. In: Steffen, M., Zavattaro, G. (eds.) FMOODS 2005. LNCS, vol. 3535, pp. 211–226. Springer, Heidelberg (2005)
15. Şerbănuţă, T.F., Chen, F., Roşu, G.: Maximal causal models for sequentially consistent systems. In: Qadeer, S., Tasiran, S. (eds.) RV 2012. LNCS, vol. 7687, pp. 136–150. Springer, Heidelberg (2013)
16. Sinha, A., Malik, S., Wang, C., Gupta, A.: Predictive analysis for detecting serializability violations through trace segmentation. In: Proceedings of the 9th International Conference on Formal Methods and Models for Codesign (MEMOCODE 2011), Cambridge, UK, pp. 99–108 (2011)
17. Sorrentino, F., Farzan, A., Madhusudan, P.: Penelope: weaving threads to expose atomicity violations. In: Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2010), Santa Fe, NM, USA, pp. 37–46 (2010)
18. Wang, C., Limaye, R., Ganai, M., Gupta, A.: Trace-based symbolic analysis for atomicity violations. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 328–342. Springer, Heidelberg (2010)
19. Wang, L., Stoller, S.D.: Runtime analysis of atomicity for multithreaded programs. IEEE Transactions on Software Engineering 32, 93–110 (2006)
20. Yi, J., Sadowski, C., Flanagan, C.: SideTrack: generalizing dynamic atomicity analysis. In: Proceedings of the 7th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD 2009), Chicago, IL, USA, pp. 8:1–8:10 (2009)
21. Zeng, R., Sun, Z., Liu, S., He, X.: McPatom: A predictive analysis tool for atomicity violation using model checking. In: Donaldson, A., Parker, D. (eds.) SPIN 2012. LNCS, vol. 7385, pp. 191–207. Springer, Heidelberg (2012)