Scaling Predictive Analysis of Concurrent Programs by Removing Trace Redundancy

JEFF HUANG, JINGUO ZHOU and CHARLES ZHANG, Hong Kong University of Science and Technology

Predictive trace analysis (PTA) of concurrent programs is powerful in finding concurrency bugs unseen in past program executions. Unfortunately, existing PTA solutions face considerable challenges in scaling to large traces. In this article, we identify that a large percentage of events in the trace are redundant for presenting useful analysis results to the end user. Removing them from the trace can significantly improve the scalability of PTA without affecting the quality of the results. We present a trace redundancy theorem that specifies a redundancy criterion and the soundness guarantee that the PTA results are preserved after removing the redundancy. Based on this criterion, we design and implement TraceFilter, an efficient algorithm that automatically removes redundant events from a trace for the PTA of general concurrency access anomalies. We evaluated TraceFilter on a set of popular concurrent benchmarks as well as real world large server programs. Our experimental results show that TraceFilter is able to significantly improve the scalability of PTA by orders of magnitude, without impairing the analysis result.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debuggin—*debugging aids; tracing; diagnostics*

General Terms: Algorithms, Performance, Theory

ACM Reference Format:
Huang, J., Zhou, J., and Zhang, C. 2013. Scaling predictive analysis of concurrent programs by removing trace redundancy. ACM Trans. Softw. Eng. Methodol. 22, 1, Article 8 (February 2013), 21 pages.
DOI = 10.1145/2430536.2430542 http://doi.acm.org/10.1145/2430536.2430542

1. INTRODUCTION

Predictive trace analysis (PTA), an effective technique to detect concurrency bugs, has drawn significant research attention in recent years [Sen and Agha 2005; Wang and Stoller 2006a; Chen et al. 2008; Farzan et al. 2009; Wang et al. 2009, 2010; Zhang et al. 2010; Vineet and Wang 2010]. Generally speaking, a PTA technique records a trace of execution events, statically (often exhaustively) generates other permutations of these events under certain scheduling constraints, and exposes concurrency bugs unseen in the recorded execution. PTA is powerful as, compared to dynamic analysis, it is capable of exposing bugs in unexercised executions and, compared to static analysis, it incurs much fewer false positives for the fact that its static analysis phase uses the concrete execution history. Unfortunately, the PTA-based solutions often experience scalability problems with large traces for the need of exhaustively checking all feasible permutations of the trace. The largest trace reported by the recent PTA techniques

Authors' address: The Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong, Correspondence e-mail: smhuang@cse.ust.hk.

DOI 10.1145/2430536.2430542 http://doi.acm.org/10.1145/2430536.2430542

```
      Thread T0          Thread T1,2,3

1:  for(i=1;i<=3;i++)    6:   lock l       11:  m()
2:  {                    7:   m()          12:  {
3:      write x;         8:   unlock l     13:    read x;
4:      fork Thread Ti;  9:   m()          14:  }
5:  }                    10:  m()
```

Fig. 1. Example code for illustrating the trace redundancy.

[Wang et al. 2009, 2010] contains less than 10K events[1] and one of the techniques [Wang et al. 2010] takes more than two minutes to analyze a trace with only 1K events. It is important for PTA techniques to scale because the trace of large complex concurrent programs can easily contain millions or even billions of events [Tallam et al. 2007].

We observe that the existing research that addresses the scalability of the PTA techniques targets at two causes of computational complexities. The first cause is the well-recognized exponential explosion of the schedule exploration space. An array of space reduction methods have been proposed such as the partial order reduction [Flanagan and Godefroid 2005], the maximal causal models [Şerbănuţă et al. 2010], and the staged analysis [Sinha and Wang 2010]. The second cause is the computational complexity inherent to the anomaly checking algorithms themselves. For instance, in any particular schedule, the number of event pairs to check for race conditions is $O(N^2)$ in the worst case, where N is the size of the trace. This complexity becomes $O(N^4)$ if the atomic-set serializability violations (ASV) [Vaziri et al. 2006] is to be checked [Lai et al. 2010]. Approaches such as the meta-analysis model [Farzan et al. 2009] and the work by Kahlon et al. [2005] can effectively reduce this type of complexity by limiting the analysis to programs that obey the nested locking discipline.

In this article, we identify that a third cause of the computational complexity comes from the fact that the trace often contains a large number of events that are mapped to the same lexical statements in the source code. While increasing the size of the trace significantly, these events do not reveal any additional information for fixing bugs when presented to the users of the PTA tools. Therefore, we can dramatically improve the scalability of PTA techniques if we can filter out this redundancy, that is, produce a smaller N, while preserving the quality of the results presented to the end user. On the surface, it seems to be a simple problem to solve by removing the operations that are lexically identical from the trace. Unfortunately, a treatment as such removes important dependency information and causes the PTA techniques to work incorrectly. Let us further illustrate this observation through an example.

The program in Figure 1 consists of a parent thread (T_0), executing line 1 to line 5, and three children threads ($T_{(1,2,3)}$), executing line 6 to line 14. Since T_0 generates 3 *writes* to variable x and $T_{(1,2,3)}$ generates 9 *reads* of x in total, a PTA technique for checking data races will need to examine 3*9=27 event pairs. It is apparent that these 27 pairs to the variable x eventually map to only two lines in the source code (line 3 and line 13). Therefore, only one racy pair of events is sufficient to tell the problem in this program. In modern-day concurrent programs, this type of redundancy is prevalent due to the single-process-multiple-data (SPMD) architectural design. A straightforward way to combat this redundancy is to only record one instance of each lexically distinctive statement. For instance, we can choose to only record the first *write* at the line 3 by T_0 as well as the first *reads* at line 13 by the other three threads. The obtained trace, albeit much smaller in size (4 x accesses instead of 12), is not that

[1]This corresponds to a 0.01sec execution of a Bank benchmark in Wang et al. [2009] with 135 lines of code.

```
T1          T2
                        5: m1():          9: m2():
                        6:   lock l;      10:  read x;
1: m1()     3: m2()     7:   write x;
2: m1()     4: m2()     8:   unlock l;
```

Fig. 2. Statements (10,7,10) form a real atomicity violation. However, the simple strategy of "dropping all re-references by the same thread to the same variable if there are no synchronization operations between them" would drop the second read of T_2 at line 10, which causes PTA to miss this atomicity violation.

useful in finding the race bug because it tells us that these *reads* are performed after the *write* as the result of the thread creation operation at line 4. Therefore, the data race cannot be detected. However, if we also record the second *write* by T_0, a PTA algorithm can correctly report the data race by only analyzing 5 accesses of x. Even better, by observing that the event sequences of the three threads $T_{(1,2,3)}$ are all identical, we can drop the events by any two of them, resulting in only 3 x accesses to be analyzed for the race detection.

Through this example, we want to point out that the identical lexical position of two recorded events is only a necessary but not sufficient condition for them to be redundant to each other, in terms of preserving the results of the PTA techniques. We propose the concept of *permutational redundancy*, in conjunction with the *lexical redundancy*, to serve as the criteria of the safe removal of events from traces before being analyzed by the PTA techniques. The *permutational redundancy criterion* states that two events by the same thread are redundant to each other (called *local redundancy*) if, first, their locksets contain no different locks and, second, their inter-thread happens-before relationships with all the other events generated by the other threads are equivalent. In addition, we extend this notion to characterize the redundant event sequences by different threads (called *global redundancy*). Two event sequences by different threads are redundant to each other if their corresponding events are lexically redundant. Going back to our example in Figure 1, the `fork Thread` statement states that the first *write* at line 3 by T_0 happens before the *read* of T_1 at line 13. However, this relationship is not true between this *read* operation and the second *write* of T_0. Therefore, the first and the second *writes* of T_0 are not permutationally redundant to each other and neither of them can be removed. By the same reasoning, the second and the third *writes* (*reads*) of T_0 ($T_{(1,2,3)}$) are in fact redundant to each other and only one of them is needed for further analysis. Moreover, because the event sequences of $T_{(1,2,3)}$ accessing x are lexically identical and their corresponding events are equivalent, they are globally redundant to each other and only one of them is needed for detecting the race problem.

To remove this trace redundancy, another simpler strategy is to drop all re-references to the same variable at the same program location by the same thread if there are no synchronization operations between them. For instance, the third *reads* of $T_{(1,2,3)}$ in our example are removed from the trace. However, this simple strategy is less preferable in two aspects. First, it is limited in removing the redundant events within the same synchronization region. Redundant thread accesses across synchronization boundaries cannot be detected using this approach. More importantly, this approach is unsound in addressing the trace redundancy in the general PTA treatment of access anomalies. It may incorrectly drop useful events that manifest access anomalies other than data races. As illustrated in Figure 2, this simple strategy removes the second *read* of T_2, which results in missing a real atomicity violation formed by the statements (10,7,10).

Based on the previous observation, we present TraceFilter, a technique that efficiently removes redundant events from a trace and, at the same time, guarantees to preserve the results of the PTA techniques. We first propose a generalized model of the PTA algorithms for analyzing the access anomaly bugs in concurrent programs. Using this

model, we associate each event in the trace with a new attribute, called *concurrency context*, in addition to its lexical location. The concurrency context primarily contains both the lock acquire/release and the message send/receive histories of the thread, at the time when the event is triggered in the trace. We show that our technique is sound that it does not misclassify any useful event to be redundant, as the concurrency context strictly preserves the permutability conditions of events. Moreover, the prefix-sharing property of the concurrency context enables us to use a compact Trie data structure to detect redundancy in a memory-friendly way and to efficiently filter out redundant events.

To evaluate our technique, we have implemented a prototype tool for analyzing the trace of concurrent Java programs and evaluated the tool on a set of popular concurrent benchmarks and real-world large server programs. We considered the PTA of all the three common concurrency access anomaly bugs including data races [Savage et al. 1997], atomicity violations [Farzan and Madhusudan 2006], and ASVs [Vaziri et al. 2006]. Our experimental results show that (1) the trace redundancy pervasively exists in concurrent programs and our technique is very effective for detecting it. The overall percentage of redundant events detected by our technique ranges from 7.9% to 99.9% in the trace, while for the real server programs, the percentage of redundancy ranges from 34.7% to 85.5%, (2) our technique is able to significantly improve the scalability of PTA. For a trace with more than 2M events (2,236,960) in Derby, the PTA with our technique was able to finish in 177.5 seconds, whereas without our technique, the same PTA does not finish in 2 hours, (3) our technique does not impair the analysis result for the PTA of concurrency access anomaly bugs. By comparing the trace analysis results for all the evaluated benchmarks, we empirically confirm that the analysis results reported by the trace analysis algorithms with our technique are the same as without our technique.

In summary, the contributions of this article are as follows.

(1) We define the concept of trace redundancy in the context of PTA for the general access anomalies and show that the redundancy pervasively exists in concurrency software systems.
(2) We present a technique that automatically removes the redundant events in the trace for improving the scalability of PTA. The soundness of our technique is guaranteed by a theorem showing that our technique does not impair the trace analysis result.
(3) We evaluate our technique on a set of concurrency benchmarks as well as several large multithreaded applications. The results show that our technique is very effective and efficient, and can significantly improve the scalability of PTA.

The remainder of this article is organized as follows. Section 2 provides preliminaries of PTA; Section 3 presents a general PTA methodology for detecting general concurrency access anomalies; Section 4 presents our technique in detail; Section 5 describes our implementation; Section 6 presents our empirical evaluation; Section 7 discusses related work; and Section 8 concludes this article.

2. PRELIMINARIES

The essential idea behind PTA is that the events in the trace can be rearranged with respect to the program constraints, such as the program order and the synchronization operations, to expose bugs that are not directly witnessed in the original trace. To help with the understanding of our technique, we introduce the common program modeling used by PTA techniques and illustrate how PTA detects the race presented earlier in Figure 1.

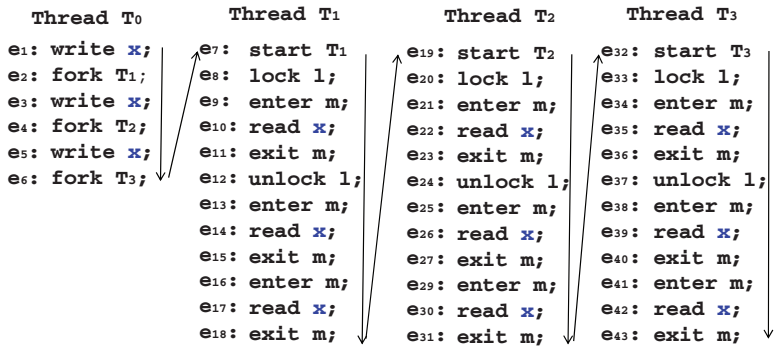

Fig. 3. A trace corresponding to a serial execution of the example program in Figure 1.

To detect the race, PTA first observes a run of the program. With no loss of generality, suppose we run the example program once and all the threads execute serially without interleaving ($T_0 \rightarrow T_1 \rightarrow T_2 \rightarrow T_3$), PTA will first produce an event trace (shown in Figure 3) that contains all the synchronization events as well as the read and write events on the shared variables, in their execution order.

Trace. A trace captures a multithreaded program execution as a sequence of events $\delta = \langle e_i \rangle$. An event e can be of the following forms [Sen 2008].

—MEM(σ, v, a, t, L) denotes that thread t performed an access $a \in \{\text{WRITE}, \text{READ}\}$ to a shared variable v while holding the set of locks L and executing the statement σ.
—ENT(m, t) denotes the entering of a method m by thread t.
—EXT(m, t) denotes the exiting of a method m by thread t.
—ACQ(l, t) denotes the acquisition of a lock l by thread t.
—REL(l, t) denotes the releasing of a lock l by thread t.
—SND(g, t) denotes the sending of a message with unique ID g by thread t.
—RCV(g, t) denotes the reception of a message with unique ID g by thread t.

In Java, the events SND(g, t) and RCV(g, t) are of the following types. If thread $t1$ starts a thread $t2$, then events SND(g, $t1$) and RCV(g, $t2$) are generated, where g is a unique message ID. If thread $t1$ calls $t2$.*join()* and $t2$ terminates, then events SND(g, $t2$) and RCV(g, $t1$) are generated, where g is a unique message ID. If a *o.notify()* on thread $t1$ signals a *o.wait()* on thread $t2$, then events SND(g, $t1$) and RCV(g, $t2$) are generated, where g is a unique message ID.

Happens-before relation. An important relation that is used by the PTA algorithms is the happens-before relation on the events exhibited by a concurrent execution. Given a trace δ, the happens-before relation $\prec$ is the smallest relation satisfying the following conditions.

—If e_i and e_j are events from the same thread and e_i comes before e_j in the trace, then $e_i \prec e_j$.
—If e_i is the sending (SND) and e_j is the receiving (RCV) of the message g, respectively, $e_i \prec e_j$.
—$\prec$ is transitively closed.

The computation of the relation $\prec$ is often done by maintaining a vector clock with every thread [Mattern 1988]. Note that, slightly different from the classical happens-before in the Java memory model [Manson et al. 2005], the lock acquisition (ACQ) and release (REL) events are not included in this happens-before model. Instead, they are modeled separately as the lockset condition (see Section 3.2).

	Examples		
	data race	atomicity violation	ASV
E	e1 – e2	e1 – e2 – e3	e1 – e2 – e3 – e4
T	t1 – t2	t1 – t2 – t1	t1 – t2 – t2 – t1
SV	s1 – s1	s1 – s1 – s1	s1 – s1 – s2 – s2
AR	u1 – u2	u1 – u2 – u1	u1 – u2 – u2 – u1
AT	r – w	r – w – r	w – r – r – w

Fig. 4. Examples of access anomaly patterns.

Now let us consider the trace in Figure 3. There are in total 43 events (e_1-e_{43}) in the trace. The events e_1-e_6 are performed by T_0, e_7-e_{18} by T_1, e_{19}-e_{31} by T_2, and e_{32}-e_{43} by T_3. There are in total three write events, $e_{(1,3,5)}$, all by thread T_0, and nine read events, among which $e_{(10,14,17)}$ are by T_1, $e_{(22,26,30)}$ by T_2, and $e_{(35,39,42)}$ by T_3. The locksets of the read events $e_{(10,22,35)}$ contain a single lock l. The locksets of the other read/write events are empty. Since the `fork Thread` t_i event must be executed before the start of thread t_i, the happens-before relation between the events are $e_1 \prec e_2 \prec e_7 \prec \cdots \prec e_{18}$, $e_2 \prec e_3 \prec e_4 \prec e_{19} \prec \cdots \prec e_{31}$, and $e_4 \prec e_5 \prec e_6 \prec e_{32} \prec \cdots \prec e_{43}$.

Given the preceding trace, PTA will first list all the read/write events on the shared variable `x` by each thread. As $e_{(1,3,5)}$ are three write events by T_0 and $e_{(10,14,17,22,26,30,35,39,42)}$ are nine read events by $T_{1,2,3}$, PTA will then check the 3*9=27 pairs of candidate races. By evaluating the lockset and happens-before relation between the two events in each candidate race, PTA will get nine real race pairs $[(e_{(3)}, e_{(10,14,17)}),(e_{(5)}, e_{(10,14,17,22,26,30)})]$. For the real race pairs, PTA then reports the lexical statement pair contained in each of them, which finally produces the race at lines (3,13) to the user.

3. A GENERAL PTA MODEL

In this section, we present a generalized detection model for the PTA of concurrency access anomalies. The model consists of a generalized pattern specification of access anomalies and the corresponding PTA algorithm. This generalization is crucial for us to reason about event redundancy with respect to the nature of PTA algorithms and a foundation for the wide applicability of our redundancy criteria.

3.1. Concurrency Access Anomalies

The most commonly known concurrency access anomalies are data races, atomicity violations, and atomic-set serializability violations (ASV), which basically contain a sequence of two to four events by two different threads on a single or two shared variables. We generalize the concept of access anomalies to allow arbitrary number of events, threads, and shared variables. And we describe each type of access anomaly as an event sequence pattern. The event sequence pattern `p` is defined as a group of equal-length sequences `[E,T,SV,AR,AT]`. The meaning of each symbol is described as follows. `E` is the characteristic event sequence defined by the specific anomaly. `T` is the thread IDs of the events in `E`. `SV` is the accessed shared variable sequence corresponding to `E`. `AR` is the atomic region sequence corresponding to `E`. `AT` is the access type sequence corresponding to `E`.

Figure 4 shows examples of data race, atomicity violation, and ASV described using the specification. For example, a data race can be described as: `E`=e1-e2, `T`=t1-t2,

SV=s1-s1, AR=u1-u2, and AT=r-w, meaning that the first thread reads a shared variable and immediately the second thread writes to it. As another example, an atomicity violation in which a shared variable is accessed by three consecutive events e_i, e_k, and e_j in the written order with the access types *write-read-write*, and e_i, e_j belong to the same atomic region, e_k belong to another, can be written as E=e1-e2-e3, T=t1-t2-t1, SV=s1-s1-s1, AR=u1-u2-u1, and AT=w-r-w. Our generalized access anomaly patterns can describe all the three commonly known access anomalies. Moreover, it allows the user to define his/her own access anomalies that may contain much more complex thread interleavings.

3.2. General PTA Algorithm

The essential idea of PTA for detecting concurrency access anomalies [Chen et al. 2008; Lai et al. 2010] is based on a permutability property between events that combines both the lockset condition and happens-before condition. We describe a generalized PTA algorithm as follows.

A PTA algorithm A, given a trace δ and a pattern p, first decides, in the pattern, the number of different threads, different shared variables, and different events on each shared variable by each thread in the same atomic region. Then it uses this information to search in the trace to obtain a set of candidate access anomalies. In order for the search to be efficient, often the trace is preprocessed to build the index based on the thread ID, the shared variable, the access type, and the atomic region. Each candidate access anomaly contains a sequence of events usually satisfying one of the patterns in Figure 4 with respect to SV, AR, and AT, but not T, the thread scheduling order. In this case, it continues to check whether there exists certain allowed permutations of events that match T under the lockset and the happens-before constraints.

For a pair of events $e_i = \text{MEM}(\sigma_i, m_i, w, t_i, L_i)$ and $e_j = \text{MEM}(\sigma_j, m_j, r, t_j, L_j)$ in the candidate access anomaly, PTA checks the following two conditions.

I. *Lockset* condition: $L_i \cap L_j = \emptyset$.
II. *Happens-before* condition: $\neg(e_i \prec e_j) \wedge \neg(e_j \prec e_i)$.

The preceding conditions mean that two events in the candidate access anomaly are *permutable*, that is, concurrent to each other (i.e., one access does not happen-before the other). Consequently, the PTA algorithm can conclude that multiple thread scheduling orders are possible for this pair of events and report this pair as a data race bug.

Finally, for each access anomaly, PTA extracts the information contained in the events and presents it to the programmer for debugging. There is no uniform rule on what information is extracted from each event, as the users of PTA may require different level of details for understanding the access anomaly bug. However, the basic information of access anomalies should contain the lexical statements in the program on which the events are triggered.

4. REMOVING TRACE REDUNDANCY

This section presents our methodology for removing the trace redundancy. We start by giving a formal modeling of the trace redundancy. We then present our algorithm on detecting the redundant events.

4.1. Modeling Trace Redundancy

Consider a PTA algorithm A that takes a trace δ as the input and produces a set of access anomaly bugs as the output. We define the concept of redundancy as follows:

Definition 4.1. Given an algorithm A and an arbitrary input δ, a subsequence X of δ is *redundant* iff $A(I) = A(I \backslash X)$.

As described in Section 3.1, an access anomaly is a sequence of events that can be specified by a meta pattern (recall Section 3.1) that defines both the attribute values of these events and the order relation between them. To facilitate our discussion, we first define a concept called *candidate access anomaly* (CAA) that will be used in our modeling of trace redundancy.

Definition 4.2. A candidate access anomaly (CAA) corresponding to a pattern p is an *event sequence,* of which the event attribute values satisfy the condition defined in p, but the order relation between them might not satisfy the condition defined in p.

Note that a CAA should correspond to a certain pattern that is provided by the user of the PTA algorithm. For different patterns, a CAA may contain different numbers of distinctive events. For example, for a data race pattern, a CAA contains two events, while for atomicity violation patterns, it contains three events. We refer to this property as the *feasibility* property of CAA, used later in proving the trace redundancy theorem.

4.1.1. A Theory of Trace Redundancy. Given a pattern and a trace, as described in Section 3.2, a PTA algorithm proceeds in two steps. First, it analyzes the trace to find the sequences of events (i.e., access anomalies) that satisfy the conditions specified in the pattern. Second, for each sequence of events, it extracts the information contained in the events and reports it to the programmer for debugging. We can decompose such PTA algorithms into two components: a rule R and a function f. The rule R evaluates on a CAA, say s, and simply reports true or false. If R reports true, it means s is a real access anomaly, and f will be applied on each event in s to generate an output.

Let us assume the generated information of each event by f is its lexical location in the program source, σ. Let $s(e \rightarrow e')$ denote the result by replacing an event, e, in a CAA, s, with another event, e'. Let $R(s)$ denote that R reports true on s. And let $\sqsubset$ denote a relation where $X \sqsubset Y$ means all events of the sequence X are also in Y. Based on the preceding assumption, we have the following theorem for detecting redundancy in δ.

THEOREM 4.3. *Given an input trace δ, a pattern p and an algorithm A with rule R and function f. An event e is redundant if there exists another event e' in δ such that $\forall$ CAA $s \sqsubset \delta \wedge e \in s$, the following three conditions hold:*

—Lexical equivalence condition: $f(s) = f(s(e \rightarrow e'))$;
—Permutational equivalence condition: $R(s) \Rightarrow R(s(e \rightarrow e'))$;
—CAA feasibility condition: $s(e \rightarrow e')$ is a CAA corresponding to p;

PROOF. According to our definition of trace redundancy in Definition 4.1, if these three conditions are all satisfied, we have $\forall e, \forall s \sqsubset \delta, e \in s \wedge R(s), \exists s' = s(e \rightarrow e') \sqsubset \delta \backslash \{e\}$ *s.t.* $R(s') \wedge f(s) = f(s')$. Thus, e is redundant. □

Theorem 4.3 says that, for an event e in δ, if the three conditions are all satisfied, then the trace δ with e or without e would always produce the same result for all the PTA algorithms in our assumption. Therefore, we can detect redundancy in the trace by checking the three conditions for each event e.

Let us first consider the lexical and the permutational equivalence conditions in Theorem 4.3. Since f generates the lexical statement σ according to our assumption, the lexical equivalence is easy to evaluate, that is, it is satisfied iff e' and e are triggered on same lexical statement. For the permutational equivalence, as we have shown in Section 3.2, that the essential determinant is the lockset and the happens-before relation between the events, which instantiate the rule R. More specifically, we define that the events e and e' satisfy the permutational equivalence condition if the following hold.

—*Lockset Equivalence*. Their locksets contain no different locks, that is, $L_e = L_{e'}$;

—*Inter-Thread Happens-Before Equivalence.* Their happens-before relationships with all the events by other threads are equivalent, that is, $\forall e''\ t_{e''} \neq t_e \vee t_{e''} \neq t_{e'}$, $\neg(e \prec e'') \wedge \neg(e'' \prec e) \Longleftrightarrow \neg(e' \prec e'') \wedge \neg(e'' \prec e')$.

Since the preceding two conditions are indeed conservative in satisfying the permutational equivalence condition, we have the following theorem.

THEOREM 4.4. *Events e and e' are permutationally equivalent to each other if the lockset condition and the inter-thread happens-before condition are both satisfied.*

In the following, we say the two events are *fully equivalent* to each other if they satisfy both the lexical equivalence and the permutational equivalence conditions. By Theorem 4.4, we can determine if e and e' are fully equivalent to each other by checking three conditions in total: the lexical equivalence, the lockset equivalence, and the happens-before equivalence. However, note that neither e nor e' is redundant even if they are fully equivalent. To ensure the redundancy, we have to also consider the CAA feasibility condition in Theorem 4.3.

The CAA feasibility condition requires that $s(e \rightarrow e')$ is a CAA corresponding to the pattern p. Recall in Section 4.1 that no two events in the CAA should be the same. Therefore, to satisfy this condition, e' must not be in s. More specifically, to determine whether or not an event e is redundant, we have to ensure that, for any CAA s, there always exists an event e' that is not in s, such that e and e' are fully equivalent to each other. However, this condition in general is impossible to satisfy without considering the pattern p that s corresponds to.

To elucidate this point, consider an atomicity violation pattern, which specifies three events with two of them, e_1 and e_2, from the same thread and the third one from another thread. Even if these two events are fully equivalent to each other, the pattern requires both of them to be present to form the bug condition. However, an event, e_3, is truly redundant if it is also fully equivalent to e_1 and e_2 because two events are sufficient according to the definition of the bug pattern.

Hence, to determine whether the CAA feasibility condition can be satisfied or not, we need to consider the specific pattern that the CAA sequence, s, corresponds to. This leads to our definition of *norm* with respect to each pattern as follows.

Definition 4.5. The *norm* of a pattern p, denoted as $\|p\|$, is the maximum number of lexically and permutationally equivalent events allowed in p.

Given this definition of pattern norm, we have the following theorem.

THEOREM 4.6. *An event e is redundant if the number of fully equivalent events to e in the trace is no less than the pattern norm $\|p\|$.*

PROOF. Suppose there are $\|p\|$ or more equivalent events to e in the trace. Let us put them into a set S. As there are at most $\|p\|$ fully equivalent events for any CAA that corresponds to the pattern p, no matter what events the CAA, s, contains, there always exists at least one event in S that is not in s but fully equivalent to e. Therefore, the CAA feasibility condition in Theorem 4.3 is satisfied. e is redundant, because both the lexical and the permutational equivalence conditions are also satisfied. □

Therefore, using Theorem 4.6, given a pattern and a trace, we can determine whether an event e is redundant or not in the trace by counting the number of fully equivalent events to e. If the number is no less than the pattern norm, we can classify that e is redundant and remove e from the trace.

4.1.2. Concurrency Context. According to Theorem 4.3, to detect redundant events, we need to check the lexical equivalence, the permutational equivalence, and the CAA

feasibility conditions between events. While the lexical equivalence and the CAA feasibility conditions are straightforward to compute, we have to properly model the lockset and happens-before relationships of each event, to deal with the permutational equivalence condition.

Recall in Section 2 that the lockset of an event is the set of locks the thread is holding when it triggers the event, and the happens-before relation is computed using vector clocks by considering the internal events in each thread and the SND and RCV events across different threads. To support the efficient checking of these two conditions for detecting redundant events, we introduce a new attribute, *concurrency context*, for each event to encode the lockset and the happens-before relation in a uniform way.

Definition 4.7. The *concurrency context of each event* includes both the lock acquire/release (ACQ/REL) and the message send/receive (SND/RCV) history of the thread at the time when the event is triggered.

By defining the concurrency context in this way, we have the following theorem.

THEOREM 4.8. *Two events from the same thread with the same concurrency context are permutationally equivalent to each other.*

PROOF. Since the concurrency context encodes the lock acquire/release history, the two events must have the same lockset, which satisfies the lockset equivalence condition. In addition, since the concurrency context encodes the message send/receive history, which determines the happens-before relation between events across different threads (recall Section 2, happens-before relation, the second condition), these two events from the same thread must also have the same happens-before relations with all the other events by the other thread, hence, satisfying the inter-thread happens-before equivalence condition. □

In addition, as programmers may require more details besides the lexical location of the access anomaly, we also include the runtime method call stack (ENT/EXT) of each event in its concurrency context, to give programmers the full calling context information for understanding the bug. Note that our definition of the concurrency context naturally supports the online computation that does not demand the global trace information. This is important since, as mentioned earlier, the large traces are even difficult to be loaded into the memory for the analysis.

4.1.3. Two Dimensions of Redundancy. This model describes a general way of determining redundancy in the context of a PTA algorithm for concurrency access anomaly detection. According to Theorem 4.6, we know that the number of fully equivalent events need to be no more than the pattern norm, and all the additional ones are considered to be redundant. Conceptually, we can decompose the redundancy into two dimensions: the redundant events from the same thread and those across different threads. According to Theorem 4.8, since the full equivalence between two lexically equivalent events by the same thread can be determined by comparing their associated concurrency contexts, an advantage of this decomposition is that it allows the separation of the local and the global reasoning of redundancy with respect to each individual thread. We next show the decomposition in detail.

Local Redundancy. The first dimension of redundancy is called the local redundancy, defined over the events of each individual thread. Consider the set of fully equivalent events. If we further divide it into subsets grouped by the thread ID, we are able to determine the redundancy locally to each thread, without checking against all the events in the trace. More specifically, if the size of some subset exceeds the pattern norm, the additional events in the subset are already redundant regardless of the events in the other subsets. We refer to these additional events as the *locally redundant* events

and they can be safely removed from the trace. As an example, consider detecting the data race on the trace in Figure 3. Since the second and third writes of x ($e_{(3,5)}$) by thread T_0 are equivalent to each other, and the norm of a data race pattern is one, we can safely remove either e_3 or e_5 from the trace.

Global Redundancy. The second dimension of redundancy is called the global redundancy, which is defined over the events across different threads. For general access anomaly patterns, however, it is difficult to determine the equivalence between events from different threads. The reason is that the permutational equivalence condition requires checking the happens-before relation between the two events against all the other events in the trace. For two events from different threads, their happens-before relationships with the events from the other thread would be different and, thus, the permutational equivalence condition may never be satisfied. For example, the events e_{30} and e_{42} by threads T_2 and T_3 (in Figure 3), respectively, are not equivalent to each other, as their happens-before relationships with all the other events in T_2 and T_3 are different. Therefore, to determine the redundancy across different threads, we need to examine the access anomaly patterns in more detail.

Recall that an access anomaly pattern [E,T,SV,AR,AT] specifies a sequence of events by different threads. Consider the element T, which specifies the meta thread ID sequence in the pattern. Our observation is that, only a limited number (n_t) of different threads are required in the formation of an access anomaly pattern. If there are more than n_t threads in the trace that contain the lexically identical events with respective to the pattern, those additional threads are redundant and all their corresponding events, which are referred to as the globally redundant events, can be removed. For example, consider the threads $T_{(1,2,3)}$ in Figure 1, since their event sequences are lexically identical to each other (because they execute the same code), we only need to keep the events from two of them because the three common access anomalies all require to analyze only two threads. The reason is that any access anomaly contributed by the events from the redundant threads can be replaced by the events from the remaining threads in the trace, as the access anomaly pattern does not require concrete but rather meta thread IDs.

To generalize to any pattern that specifies n_t different threads, we determine the global redundancy by comparing the entire event sequences between different threads. For the set of threads that contain lexically identical event sequences, we only keep n_t of them (if the size of the set is larger than n_t) and discard the events from the rest of them. We detect the global redundancy after processing the local redundancy to reduce the computation effort.

4.2. Filtering Redundant Events

To efficiently encode and filter redundant events, we design two filters for dealing with both the locally and the globally redundant events. Our filters use the Trie data structure to represent the concurrency contexts. The reason for choosing Trie is that, for any particular thread, the stream of events exhibits strong temporal locality due to the stack-based computation model. Events generated at the top level of the function stack share all their preceding events generated by the entire stack. We leverage this phenomenon to make good use of the prefix sharing capability of Trie and to perform the online analysis of the events.

More specifically, each node in the Trie represents an element in the concurrency context, For example, a method entry or a lock acquisition operation, and each node is also associated with a bounded stack, of which the capacity is set to be the norm of the access anomaly pattern. During the filtering, the new coming event with the concurrency context represented by the corresponding node in the Trie is added to the

stack; when the stack is full, the event is discarded and automatically removed from the trace as it guarantees to be redundant with respect to the analysis result.

Algorithm 1 shows our TraceFilter algorithm for removing redundant events in the trace. It consists of two parts: an online algorithm (Algorithm 2 DetectLocalRedundancy) for detecting the local redundancy and an in-memory algorithm (Algorithm 3 DetectGlobalRedundancy) for detecting the global redundancy. The algorithm conducts a linear scan of the input trace and maintains a concurrency context for each thread during the analysis. The concurrency context is computed as follows. If the event is a method entry/lock acquisition (ENT/ACQ) event, the method/lock ID (m/l) will be added to the thread's concurrency context. If the event is a method exit/lock release (EXT/REL) event, the most recent method/lock ID (m/l) will be removed from the thread's concurrency context. If the event is a message send/receive (SND/RCV) event, the message ID (g) will be added to the thread's concurrency context. Otherwise, if the event is a shared variable read or write access (MEM), it will call the algorithm DetectLocalRedundancy for checking redundancy with the current concurrency context of the thread.

ALGORITHM 1: TraceFilter(δ)

```
1: Input: δ - a trace ⟨e_i⟩
2: cctx_t ← empty concurrency context for each thread t
3: for i = 1 to |δ| do
4:    switch e_i
5:       case: MEM(σ_i, v_i, a_i, t_i, L_i)
6:          DetectLocalRedundancy(t_i, σ_i, cctx_{t_i});
7:       case: ENT(m_i, t_i)
8:          add m_i to cctx_{t_i};
9:       case: EXT(m_i, t_i)
10:         remove m_i from cctx_{t_i};
11:      case: ACQ(l_i, t_i)
12:         add l_i to cctx_{t_i};
13:      case: REL(l_i, t_i)
14:         remove l_i from cctx_{t_i};
15:      case: SND/RCV(g_i, t_i)
16:         add g_i to cctx_{t_i};
17: end for
18: DetectGlobalRedundancy(δ);
```

ALGORITHM 2: DetectLocalRedundancy(e, t, σ, $cctx$)

```
1: Input: e - an event in the trace
2: Input: t - an thread ID
3: Input: σ - a program location
4: Input: cctx - a concurrency context
5: local_trie_map(t →(σ → trie)): a map that maps a given t and σ to a trie
6: trie ← local_trie_map(t, σ)
7: stack ← trie.get(cctx) //get the corresponding stack of cctx
8: if stack is full then
9:    discard e
10: else
11:    add e to stack
12: end if
```

Detecting Local Redundancy. In our algorithm DetectLocalRedundancy, our local redundancy filter checks each event that is associated with a shared variable access. We

ALGORITHM 3: DetectGlobalRedundancy(δ)

```
1: Input: δ - a trace ⟨e_i⟩
2: δ^t: the event sequence by thread t in δ
3: trie: the global trie
4: //iterate through the set of all threads
5: for all t ∈ T do
6:    trie ←UpdateGlobalTrie(δ^t)
7:    stack ← trie.get(t)//get the corresponding stack of t
8:    if stack is full then
9:       discard δ^t
10:   else
11:      add δ^t to stack
12:   end if
13: end for
```

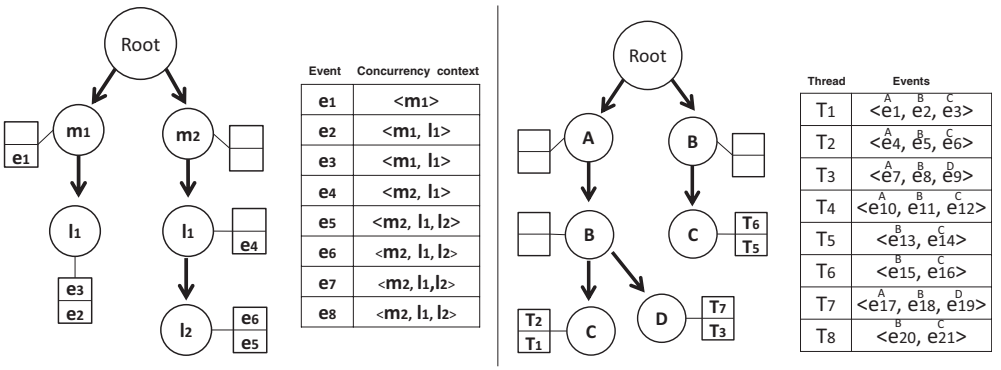

Fig. 5. Trie representation of local (left) and global (right) redundancy.

first find the node in the Trie, given the concurrency context of the event. If the box of the node is full, the event is discarded from the trace and the algorithm continues to process the next event. The algorithm terminates after the last event in the trace is analyzed. The worst case time complexity of this algorithm is linear to the trace size multiplied by the maximum length of the concurrency context, that is, the number of events in the concurrency context.

Figure 5 (left) shows an exemplary snapshot of the local filter, assuming the norm of the detected access anomaly pattern is 2. The table in Figure 5 (left) lists eight events and their associated concurrency contexts that consist of locks, l_1 and l_2, and methods, m_1 and m_2. In this Trie, each node contains a particular context element and is associated with a stack of size 2 for storing events. The events e_7 and e_8 are not stored because they hits the same node as e_5 and e_6 and the stack is full.

Detecting Global Redundancy. We invoke the algorithm, DetectGlobalRedundancy, to remove the global redundancy across different threads after removing the local redundancy in each thread. We first categorize the events according to their thread IDs. Instead of populating the Trie using the attributes of the concurrency context, we use the lexical location of the event as the key to populate the thread IDs of each event in the Trie. Our algorithm iterates through the set of all threads and updates the global trie according to the lexical locations of the events in the event sequence of each thread. If the corresponding lexical locations of two event sequences by two threads are identical, the two thread IDs will be placed in the stack associated with the same node. If a stack is full, all the events by the new coming thread are discarded.

Figure 5 (right) shows an exemplary snapshot of the global filter. The table shows the categorized events generated by eight threads. The lexical locations are shown on top of each event. The events from the threads T_1, T_2, and T_4 have the same lexical location sequence $< A, B, C >$, and the events from the threads T_5, T_6, and T_8 have the same lexical location sequence $< B, C >$. Following the sequence, the thread IDs are recorded by the filter. Suppose the access anomaly patterns in this example specify at most 2 different threads, the events from the threads T_4 and T_8 are all dropped because the corresponding stacks of T_4 and T_8 in the global trie are full. T_4 is mapped to the same node as T_1 and T_2, and T_8 is mapped to the same node as T_5 and T_6.

This operation uses the global information of the remaining event sequences of each thread. Therefore, the entire trace is required to be in the memory. For large raw trace, this requirement is hard to satisfy as the memory resource is often limited. Fortunately, after removing the local redundancy, the size of the raw trace is often greatly reduced, so that our technique is able to handle large traces despite the fact that removing the global redundancy is not memory-friendly.

5. IMPLEMENTATION

To evaluate our technique, we need a representative PTA implementation. However, none of the tools for the PTA techniques [Chen et al. 2008; Sorrentino et al. 2010; Wang et al. 2009, 2010; Sen and Agha 2005; Wang and Stoller 2006a; Farzan et al. 2009; Zhang et al. 2010] is publicly available. Therefore, we implemented our technique on top of PECAN [Huang and Zhang 2011], our PTA tool for detecting concurrency access anomalies for applications written in Java.[2] We faithfully implemented in PECAN the algorithms in Section 3.2 for detecting all the three commonly known access anomalies. Our implementation is publicly available at http://www.cse.ust.hk/prism/pecan/.

To obtain a trace, PECAN first takes the bytecode of an arbitrary Java program and outputs an instrumented version to collect interested events of the program execution.

For detecting concurrency bugs using PTA, PECAN collects the following types of events in a global order: read/write accesses to shared variables, method entry/exit, lock acquisition/release, fork/join, and wait/notify events. To support the recording of long running programs, PECAN does not hold the entire trace in the main memory but saves it to a database. To reduce the unnecessary recording of accesses on thread local variables, PECAN also performs a static thread escape analysis [Halpert et al. 2007] to identify all the possible shared variables in the program. Each event in the trace is associated with a set of attributes: the access type, the memory address, the thread ID, and the location in the program source. To reduce the runtime cost, the concurrency context information of each event used by our technique for detecting redundant events is not recorded during the trace collection. Instead, it is computed and maintained at the time when the events in the trace are processed by our technique.

After applying our technique for removing the redundant events in the trace, the PTA engine of PECAN takes the trace as input and reports detected access anomalies. Each of the reported access anomalies is a pure event sequence satisfying the specification of the access anomaly pattern. For two access anomalies with the same lexical information but contain different event sequences, PECAN is also configured to support reporting either both of them or only one of them, by checking the redundancy between them, for the evaluation purpose of our technique. Interested readers can refer to our earlier work [Huang and Zhang 2011] to see a detailed description of PECAN.

[2]Our technique is general to the execution traces of concurrent programs written in any programming language.

Table I. Experimental Results - RQ1: Effectiveness

Program	SLOC	Input/#Thread	Trace			TraceFilter	
			#Events	#SV	#Size	Local redundancy	Global redundancy
BuggyPro	348	33	10,075	5	424KB	5,876(**58.3%**)	147(1.5%)
Shop	220	100	15,560	3	654KB	6,684(**44.1%**)	462(2.9%)
Loader	139	100	34,788	2	1.5MB	12,094(**34.8%**)	97(0.3%)
ArrayList	5,979	451	40,558	696	1.7MB	3,208(**8.1%**)	0(0.0%)
LinkedList	5,866	451	53,173	2,266	2.2MB	4,020(**7.9%**)	0(0.0%)
RayTracer	1,924	SizeA/10	350,688	24	14.7MB	327,645(**93.4%**)	20(0.0%)
SpecJBB2005	17,245	8	484,841	113	20.4MB	281,338(**58.0%**)	2(0.0%)
Tsp	709	map4/4	1,048,433	260	44,1MB	1,042,293(**97.7%**)	1,248(0.1%)
Moldyn	1,352	SizeA/10	1,062,629	26	44.7MB	1,003,062(**94.4%**)	196(0.0%)
Sor	951	SizeA/4	5,545,200	6	233.2MB	5,544,954(**99.9%**)	0(0.0%)
OpenJMS	262,842	10	904,435	285	38.0M	773,764(**85.5%**)	0(0.0%)
Tomcat	339,405	100	1,296,338	401	54.5M	569,047(**43.9%**)	663(0.0%)
Jigsaw	381,348	10	479,105	407	20.1M	166,338(**34.7%**)	5(0.0%)
Derby	665,733	bug#2861/100	2,236,960	199	94.1M	1,449,550(**64.8%**)	4,502(0.2%)

6. EVALUATION

The goal of our technique is to improve the scalability of PTA of concurrency access anomalies while guarantee the soundness of the analysis. Accordingly, our evaluation aims at answering the following research questions.

RQ1. *Effectiveness*. How much local redundancy as well as global redundancy can our approach remove from the trace?
RQ2. *Efficiency*. How efficient is our approach for removing trace redundancy? And how much improvement on the scalability of PTA for concurrency access anomalies can our approach contribute?
RQ3. *Correctness*. Empirically does our approach indeed guarantee the soundness? that is, it should not remove any nonredundant events from the trace w.r.t the PTA.

All our experiments were conducted on a 8-core 3.00GHz Intel Xeon machines with 16GB memory and Linux version 2.6.22. The remainder of this section presents our experimental results on the three research questions.

6.1. Benchmarks

Our technique is quantified using a set of widely used third-party concurrency benchmarks. We configure the program inputs to generate traces of different sizes and complexity. To understand the performance of our technique on real applications in practice, we also include several large server systems in our benchmarks. The first column in Table I shows the benchmarks used in our experiments. The first ten benchmarks are multithreaded benchmarks. *BuggyPro*, *Shop* and *Loader* are from the IBM ConTest benchmark suite [Farchi et al. 2003], *ArrayList* and *LinkedList* are open libraries from Suns JDK 1.4.2, *RayTracer*, *Moldyn* and *Sor* are from the Java Grande Forum, and *Tsp* is a parallel branch and bound algorithm for the traveling salesman problem from ETH [von Praun and Gross 2001]; *SpecJBB2005* is a benchmark for evaluating the performance of server side Java from the Standard Performance Evaluation Corporation. The remaining four benchmarks are real server systems. *OpenJMS* is an enterprise message-oriented middleware server, *Tomcat* is a widely used JSP and Servlet Container, *Jigsaw* is W3C's leading-edge web server platform, and *Derby* is Apaches widely

used open source Java RDBMS. The sizes of our evaluation benchmarks range from a few hundred lines to over 600K lines of code.

6.2. RQ1: Effectiveness

The goal of our first research question is to investigate how much redundancy exists in the execution traces of real concurrent programs. To generate the data necessary for investigating this question, we proceed as follows. For each benchmark, we first run it multiple times with different inputs and the number of threads, and used PECAN to collect the corresponding trace of each run. For each trace, we then apply our technique to produce a filtered trace with the redundancy removed. As our technique deals with two dimensions of redundancy (the local redundancy and the global redundancy), we measured the percentage of redundant events with respect to the local and global redundancy, respectively.

Table I shows our experimental results. Column 3 (Input/#Thread) reports the input data (if available) and the number of threads configured in the recorded execution of the benchmark. Columns 4–6 (#Events, #SV, #Size) report the number of events in the trace, the number of real shared memory locations that contain both read and write accesses from different threads, and the size of the trace on the disk, respectively. As the table shows, the number of events in the trace ranges from more than 10K to 5M, with sizes from more than 400KB to 233MB on disk. Compared to the traces evaluated in the other PTA techniques [Wang et al. 2009, 2010, Vineet and Wang 2010; Chen et al. 2008], the traces in our experiments are orders of magnitude larger. Columns 7–8 (Local, Global) report the number of local and global redundant events, respectively, detected by our technique in the corresponding trace. In the small benchmarks, the percentage of local redundancy ranges from 7.9% to 99.9%, and the percentage of global redundancy ranges from 0.0% to 2.9%. For the real server programs, the percentage of local redundancy ranges from 34.7% to 85.5%, and the percentage of global redundancy ranges from 0.0% to 0.2%.

The percentage of global redundancy is often very small compared to that of the local redundancy. The reason is that our TraceFilter algorithm has in the first place removed most of events in the category of local redundancy. Hence, no matter how much the global redundancy there is, the number of the remaining events in the trace after removing the local redundancy is already much smaller compared to the size of the original trace. If the global redundancy is detected first, the reported percentage of global redundancy would be much higher. However, in that case, the entire trace should be loaded into the memory first, as detecting the global redundancy requires the global information of the trace. Nonetheless, the data in the table confirm our hypothesis that the redundancy pervasively exists in concurrent programs. Although the percentage of redundancy in the real large sever programs is not as high as that in the small benchmarks, it already accounts for more than one-third to a half of the entire trace.

6.3. RQ2: Efficiency

The goal of our second research question is to assess if our approach is efficient in detecting redundant events. Since our objective is to improve the overall scalability of PTA, the analysis time of our technique should not contribute significantly to the overall analysis time. Hence, we conduct experiments to evaluate the efficiency of our technique on various traces. To generate the data necessary for investigating this question, we proceeded as follows. For both of the original trace and the filtered trace, we use PECAN to analyze the three common access anomalies on them. During the analysis, we record the following three measurements: the amount of time needed by our technique to remove both the local and the global redundancy, the time taken for the bug detection of PTA using the filtered and the unfiltered trace. For large traces, it

Table II. Experimental Results - RQ2: Efficiency

Program	Trace	TraceFilter		PTA	
		Local	Global	N	Y
BuggyPro	10,075	105ms	9ms	3.50s	1.4s
Shop	15,560	599ms	2ms	45.1s	2.6s
Loader	34,788	1.06s	5ms	456.0s	71.7s
ArrayList	40,558	14.9s	5ms	131.5s	115.6s
LinkedList	53,173	26.4s	15ms	100.5s	128.9s
RayTracer	350,688	1.07s	3ms	>**2h**	9.2s
SpecJBB	484,841	2.2s	12ms	112.6s	25.5s
Tsp	1,048,433	22.6s	10ms	>**2h**	402.5s
Moldyn	1,062,629	3.3s	4ms	>**2h**	27.4s
Sor	5,545,200	4.8s	1ms	>**2h**	33.7s
OpenJMS	904,435	9.7s	2ms	220.0s	17.2s
Tomcat	1,296,338	12.0s	5ms	1440.1s	29.7s
Jigsaw	479,105	19.5s	22ms	695.6s	35.5s
Derby	2,236,960	42.3s	16ms	>**2h**	177.5s

is possible that a PTA tool, such as PECAN, is not able to load the trace into memory or finish processing the trace in a reasonable amount of time. In such cases, we set a 2-hour time bound for the analysis and we terminate it if it did not finish in 2 hours, and we report out of memory error (OOM) if the analysis crashed due to memory exhaustion.

Table II shows the experimental results for our research question RQ2. Columns 1–2 report the benchmark program and the size of the corresponding trace. Each trace is the same as the one for evaluating the effectiveness of our technique in Table I. Columns 3–4 report the time taking our technique to detect the local redundancy and the global redundancy, respectively, in the trace. The time for removing the local redundancy ranges from 105ms for small traces to 42.3s for large traces, while that of detecting the global redundancy is negligible (a few milliseconds), as the number of threads in the trace are relatively small (from 4 to 100). We observe that the analysis time of our technique really depends on the complexity of the trace, for example, the number of shared variables and the depth of the concurrency context of events in the trace. For instance, for the trace with more than 5M events in the *Sor* benchmark, our technique took only less than 5 seconds to process it, whereas it took 26.4s for processing the trace in the *LinkedList* benchmark containing only 53K events. However, overall, these results show that our technique is very efficient for removing the trace redundancy.

On the aspect of improving the PTA scalability, Columns 5–6 report the total amount of time for the PTA to process the trace, without and with our technique for removing the redundant events, respectively. The data show that, in most cases (except *LinkedList* and *ArrayList*), the time needed for PTA using our technique is significantly reduced compared to the runs without using our technique. For example, for the trace with more than 1M (1,296,338) events in the *Tomcat* benchmark, our technique reduced the original PTA time from 1440.1s to 29.7s. And for the trace with 2,236,960 events in the Derby benchmark, the trace analysis with our technique was able to finish in less than 177.5 seconds, whereas, for the unfiltered trace, the same analysis did not finish in 2 hours. The only two exceptions were the traces in the *LinkedList* benchmark and the *ArrayList* benchmark. We observe that the reason is that the percentages of redundancy in these two traces are relatively small (7.9% and 8.1% respectively). Since there is not many reduction opportunities, the amount of time for the PTA to analyze the traces with and without our technique are comparable. Nevertheless, as our technique is efficient, even for these two traces, the bug-detection time saved by our

Table III. Experimental Results - RQ3: Correctness

Program	Trace	Race		Atom		ASV	
		N	Y	N	Y	N	Y
BuggyPro	10,075	9	9	1	1	0	0
Shop	15,560	16	16	6	6	0	0
Loader	34,788	2	2	0	0	0	0
ArrayList	40,558	0	0	2	2	4	4
LinkedList	53,173	0	0	4	4	34	34
SpecJBB	484,841	24	24	1	1	0	0
OpenJMS	904,435	3	3	7	7	0	0
Tomcat	1,296,338	0	0	0	0	0	0
Jigsaw	479,105	121	121	209	209	443	443

technique for the PTA still almost offsets the cost incurred by the redundancy removal. In summary, the results demonstrate that our approach is very effective in removing the trace redundancy and therewith significantly improving the scalability of PTA for detecting concurrency access anomalies in real world large traces.

6.4. RQ3: Correctness

The validity of the effectiveness and the efficiency evaluation is based on the assumption that our technique does not affect the analysis results of PTA presented to the programmer. Although our redundancy model in Section 4.1 shows that our technique is able to guarantee the soundness, that is, it does not misclassify any non-redundant event to be redundant, we would also like to see whether the claim holds empirically in large traces in practice. It is important for us to confirm the correctness of our technique with experiments.

For large traces, to verify the correctness of PTA results is difficult because in many cases the bug detection does not finish in 2 hours. Therefore, we are unable to analyze benchmarks including *RayTracer*, *Moldyn*, *Tsp*, *Sor* and *Derby*. For the traces of other benchmarks, we first run them unaltered through PECAN and obtain the detected access anomalies that are potentially duplicated with respect to the source code locations. From these results, we remove the duplicated reports and compare the remaining results to the ones reported by PECAN using the filtered trace.

Table III shows the trace we selected and the number of distinct access anomalies for each type of analysis. Columns labeled N and Y indicate whether the analysis is on the unfiltered or the filtered trace. The results empirically support the correctness of our technique. For all these traces, we found that the PTA using the filtered trace produced the same result as that of the unfiltered trace. The reason that many cells in the table are zero is that PTA did not detect any bug from the recorded trace.

6.5. Threats to validity

We analyze the construct validity and internal validity for our experiments in this work.

Construct Validity. Because none of the PTA tools is publicly available, we implemented the PTA algorithms according to Section 3.2. The results might suffer from the subjectiveness incurred by performing the experiments on our own implementation of PTA algorithms. This threat is not significant for two reasons. First, PECAN is a serious implementation of PTA that has a good capability of predicting bugs. Second, we use PECAN as a black box to show the comparative performance differences. We put our implementation publicly available [Huang and Zhang 2011] and welcome external validations.

Representativeness of the Trace. We only ran each benchmark a number of times with a limited input for the trace collection, the trace might not be representative for real programs. However, compared to existing literature, the traces we use are already orders of magnitude larger with randomly selected inputs. This is sufficient to demonstrate the effectiveness of our technique.

7. RELATED WORK

A large body of recent research has invested in the PTA of concurrent programs. Sen and Agha [2005] first proposed a generalized predictive analysis technique for detecting violations of safety properties. Wang and Stoller [2006b] proposed the reduction-based and block-based algorithms for checking atomicity on the execution trace. Chen et al. [2008] presented a framework for predictive analysis of concurrent Java programs. Lai et al. [2010] combined PTA with randomized active testing [Sen 2008] to detect atomic-set serializability violations in a run. Wang et al. [2010] also developed a symbolic analysis model for finding concurrency errors based on the execution trace. A common difficulty in these techniques is that they do not scale as the size of executions increases, because essentially the size of thread interleavings they need to explore from the trace is exponential.

To alleviate the scalability problem of PTA, Farzan et al. [2009] developed a meta-analysis model that produces an efficient algorithm for checking atomicity violations in programs that obey the nested locking discipline. The algorithm works in time linear in the length of the runs, and quadratic in the number of threads, and was also used in a recent work [Sorrentino et al. 2010] for testing and debugging atomicity violations.

A number of runtime monitoring tools have been developed that also incorporate the capability of PTA for detecting data races. These tools include Intel Thread Checker, [3] Sun Thread Analyzer [Sun Microsystems, Inc. 2007], and Thread Sanitizer Serebryany and Iskhodzhanov [2009]. To improve the performance of these tools, Sack et al. [2006] investigated several runtime filtering approaches to filter away references unlikely to be involved in data races. One such filter related to our technique is called duplicate filter that filters out repeated loads and stores to the same memory addresses. Besides the fact that the duplicate filter is used at runtime to filter out redundant events that causes significant runtime overhead, the main problem in the duplicate filter is that it does not consider the concurrency context in each events and, hence, might miss real bugs as illustrated in our Figure 1.

An alternative way to reduce the size of the trace for improving the scalability of PTA is to selectively recording events at runtime. Several efficient online sampling and tracking approaches [Marino et al. 2009; Bond et al. 2010; Yu et al. 2005] have been proposed along this line of research. For instance, both LiteRace [Marino et al. 2009] and Pacer [Bond et al. 2010] presented low-overhead sampling-based techniques for detecting data races, and RaceTrack [Yu et al. 2005] employed an adaptive approach that automatically directs more effort to areas that are more race-suspicious. While these approaches produce traces much smaller in size, they have a hard time in balancing the bug detection capability and the runtime overhead. Moreover, for long running programs, essentially these approaches are not applicable for producing small traces, as the size of logged trace always increases with the program execution.

8. CONCLUSION

We have presented a technique that automatically removes redundant events from the execution trace, which significantly improves the scalability of predictive analysis techniques for detecting concurrency access anomalies. The soundness of our technique is

[3]http://www.intel.com/support/performancetools/thread-checker.

guaranteed by a trace redundancy theorem showing that our technique does not impair the trace analysis result. Our experiments on a set of popular concurrent benchmarks as well as real world large server programs demonstrated the effectiveness, the efficiency, as well as the correctness of our technique.

REFERENCES

BOND, M. D., COONS, K. E., AND MCKINLEY, K. S. 2010. Pacer: Proportional detection of data races. In *Proceedings of PLDI*.

CHEN, F., SERBANUTA, T. F., AND ROSU, G. 2008. Jpredictor: A predictive runtime analysis tool for Java. In *Proceedings of ICSE*.

FARCHI, E., NIR, Y., AND UR, S. 2003. Concurrent bug patterns and how to test them. In *Proceedings of IPDPS*.

FARZAN, A. AND MADHUSUDAN, P. 2006. Causal atomicity. In *Proceedings of CAV*.

FARZAN, A., MADHUSUDAN, P., AND SORRENTINO, F. 2009. Meta-analysis for atomicity violations under nested locking. In *Proceedings of CAV*.

FLANAGAN, C. AND GODEFROID, P. 2005. Dynamic partial-order reduction for model checking software. In *Proceedings of POPL*.

HALPERT, R. L., PICKETT, C. J. F., AND VERBRUGGE, C. 2007. Component-based lock allocation. In *Proceedings of PACT*.

HUANG, J. AND ZHANG, C. 2011. PECAN: Persuasive Prediction of Concurrency Access Anomalies. In *Proceedings of ISSTA*.

KAHLON, V., IVANCIC, F., AND GUPTA, A. 2005. Reasoning about threads communicating via locks. In *Proceedings of CAV*.

LAI, Z., CHEUNG, S. C., AND CHAN, W. K. 2010. Detecting atomic-set serializability violations in multithreaded programs through active randomized testing. In *Proceedings of ICSE*.

MANSON, J., PUGH, W., AND ADVE, S. V. 2005. The Java memory model. In *Proceedings of POPL*.

MARINO, D., MUSUVATHI, M., AND NARAYANASAMY, S. 2009. Literace: Effective sampling for lightweight data-race detection. In *Proceedings of PLDI*.

MATTERN, F. 1988. Virtual time and global states of distributed systems. In *Proceedings of the Workshop on Parallel and Distributed Algorithms*.

SACK, P., BLISS, B. E., MA, Z., PETERSEN, P., AND TORRELLAS, J. 2006. Accurate and efficient filtering for the intel thread checker race detector. In *Proceedings of ASID*.

SAVAGE, S., BURROWS, M., NELSON, G., SOBALVARRO, P., AND ANDERSON, T. 1997. Eraser: A dynamic data race detector for multi-threaded programs. In *Proceedings of TOCS*.

SEN, K. 2008. Race directed random testing of concurrent programs. In *Proceedings of PLDI*.

SEN, K. AND AGHA, G. 2005. Detecting errors in multithreaded programs by generalized predictive analysis of executions. In *Proceedings of FMOODS*.

ŞERBĂNUŢĂ, T. F., CHEN, F., AND ROŞU, G. 2010. Maximal causal models for sequentially consistent multithreaded systems. Tech. rep., University of Illinois.

SEREBRYANY, K. AND ISKHODZHANOV, T. 2009. Threadsanitizer: Data race detection in practice. In *Proceedings of WBIA*.

SINHA, N. AND WANG, C. 2010. Staged concurrent program analysis. In *Proceedings of FSE*.

SORRENTINO, F., FARZAN, A., AND MADHUSUDAN, P. 2010. Penelope: Weaving threads to expose atomicity violations. In *Proceedings of FSE*.

SUN MICROSYSTEMS, INC. 2007. Sun studio 12: Thread analyzer user's guide. http://download.oracle.com/docs/cd/E19205-01/820-0619/820-0619.pdf.

TALLAM, S., TIAN, C., GUPTA, R., AND ZHANG, X. 2007. Enabling tracing of long-running multithreaded programs via dynamic execution reduction. In *Proceedings of ISSTA*.

VAZIRI, M., TIP, F., AND DOLBY, J. 2006. Associating synchronization constraints with data in an object-oriented language. In *Proceedings of POPL*.

VINEET, K. AND WANG, C. 2010. Universal causality graphs: A precise happens-before model for detecting bugs in concurrent programs. In *Proceedings of CAV*.

VON PRAUN, C. AND GROSS, T. R. 2001. Object race detection. In *Proceedings of OOPSLA*.

WANG, C., KUNDU, S., GANAI, M. K., AND GUPTA, A. 2009. Symbolic predictive analysis for concurrent programs. In *Proceedings of FM*.

WANG, C., LIMAYE, R., GANAI, M. K., AND GUPTA, A. 2010. Trace-based symbolic analysis for atomicity violations. In *Proceedings of TACAS*.

WANG, L. AND STOLLER, S. D. 2006a. Accurate and efficient runtime detection of atomicity errors in concurrent programs. In *Proceedings of PPoPP*.

WANG, L. AND STOLLER, S. D. 2006b. Runtime analysis of atomicity for multithreaded programs. In *Proceedings of TSE*.

YU, Y., RODEHEFFER, T., AND CHEN, W. 2005. Racetrack: Efficient detection of data race conditions via adaptive tracking. In *Proceedings of SOSP*.

ZHANG, W., SUN, C., AND LU, S. 2010. Conmem: Detecting severe concurrency bugs through an effect-oriented approach. In *Proceedings of ASPLOS*.

Received June 2011; revised October 2011; accepted November 2011