PAPER

Biped: Bidirectional Prediction of Order Violations

Xi CHANG[†a)], **Zhuo ZHANG**[††b)], ***Nonmembers***, **Yan LEI**[††], ***Student Member***, ***and*** **Jianjun ZHAO**[†], ***Nonmember***

SUMMARY Concurrency bugs do significantly affect system reliability. Although many efforts have been made to address this problem, there are still many bugs that cannot be detected because of the complexity of concurrent programs. Compared with atomicity violations, order violations are always neglected. Efficient and effective approaches to detecting order violations are therefore in urgent need. This paper presents a bidirectional predictive trace analysis approach, BIPED, which can detect order violations in parallel based on a recorded program execution. BIPED collects an expected-order execution trace into a layered bidirectional prediction model, which intensively represents two types of expected-order data flows in the bottom layer and combines the lock sets and the bidirectionally order constraints in the upper layer. BIPED then recognizes two types of candidate violation intervals driven by the bottom-layer model and then checks these recognized intervals bidirectionally based on the upper-layer constraint model. Consequently, concrete schedules can be generated to expose order violation bugs. Our experimental results show that BIPED can effectively detect real order violation bugs and the analysis speed is 2.3x-10.9x and 1.24x-1.8x relative to the state-of-the-art predictive dynamic analysis approaches and hybrid model based static prediction analysis approaches in terms of order violation bugs.

key words: *concurrency bug, trace model, predictive trace analysis, order violation*

1. Introduction

Concurrency has become one of the major techniques to improve software performance. It, however, raises concurrency bugs, characterized by various criteria such as data races [1], dead lock [2]–[4], atomicity violation [5] and order violation [6]. Combating concurrency bugs effectively has been becoming more and more important.

Many techniques have been proposed to deal with data races [7]–[12]. A recent work [13] found that only 10% of the true data races are harmful and the remaining are benign data races which do not compromise program's correctness. Thus, atomicity violations and order violations are referred as more important criteria [6].

A real-world concurrency bug characteristic study [6] shows that order violations account for 30% of all non-deadlock concurrency bugs. Compared with atomicity violation bugs [14]–[20], many order violation bugs always change the expected inter-thread data flows, instead of thread local logic [21]. A few works [20], [21] address order violation bugs with dynamical detection and obtain considerable successes, however, there still remains much processes to make for exposing order violation bugs efficiently.

Focusing on a read owned by a thread, we observe that there exist two kinds of expected inter-thread data flows:

- *Use-Predef*: A shared variable should be used for a read event after its value has already been predefined by a prior write event, we refer to such a data flow as an *Use-Predef* relation. However, the order of such a data flow may be violated if there is not order synchronization within this cross-thread interval. As a result, this read event uses an overdue definition older than the expected one. We refer to such an order violation as an overdue usage (OU) and an execution interval allowing to violate the relative order of an *Use-Predef* relation as an overdue usage interval (OUI).

  Figure 1 (a) shows an overdue usage in OpenJMS-0.7.7. Normally, a thread *T2* reads the shared field `multiplexer` at line *S2* after it is initialized by another thread *T1* at line *S1*. However, there is not any order synchronization ensuring that *S2* happens after *S1* in the interval between *S2* and *S1*. Consequently, the thread *T2* will use an overdue definition (uninitialized), throwing a `ResourceException`.
- *Use-Redef*: A shared variable should be used for a read event before its value is redefined by a posterior write event, we refer to such a data flow as an *Use-Redef* relation. However, the order of such a data flow may be violated, if there is not order synchronization within this cross-thread interval. Consequently, this read event may use a premature definition newer than the expected one. We refer to such an order violation as a premature usage (PU) and an execution interval allowing to violate the relative order of an Use-Redef relation as a premature usage interval (PUI).

  Figure 1 (b) shows a premature usage in Jigsaw-2.2. Normally, a thread reads the shared field `queue.size` at line *S1* before clearing it from the queue at line *S2*. However, there is not any order synchronization ensuring that *S1* happens before *S2* in the interval between *S2* and *S1*, it leads to that thread *T2* will use a premature definition from *S2*, throwing an `ArrayIndexOutOfBoundsException`.

The above observation indicates that it is likely to expose these order violation bugs if we can find such OUIs and

Manuscript received October 17, 2014.
Manuscript publicized October 29, 2014.

[†]The authors are with the Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai, China.
[††]The authors are with College of Computer, National University of Defense Technology, Changsha, China.
a) E-mail: cici_cx@sjtu.edu.cn
b) E-mail: zz8477@126.com

DOI: 10.1587/transinf.2014EDP7347

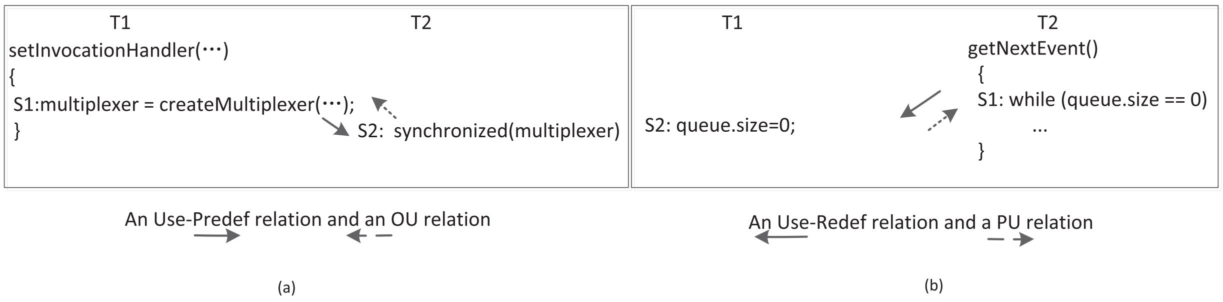

T1
T2
setInvocationHandler(⋯)
{
S1:multiplexer = createMultiplexer(⋯);
}
S2: synchronized(multiplexer)
An Use-Predef relation and an OU relation
(a)
T1
T2
getNextEvent()
{
S1: while (queue.size == 0)
...
}
S2: queue.size=0;
An Use-Redef relation and a PU relation
(b)

Fig. 1 Two kinds of order violations.

PUIs in an execution. It is, unfortunately, hard to find such OUIs and PUIs effectively by a dynamic analysis. Specifically, during one execution, an advanced dynamic analysis tool can only recognize one unexpected definition before a read event of a shared variable, but it cannot find other unexpected ones. For example in Fig. 1 (b), dynamic analysis tools would capture the definition from statement S2 which can reach statement S1 through postponing this statement S1, but it cannot find S2 as a premature definition to S1 in an execution.

In this paper, we extend the promising static technique, predictive trace analysis (PTA) [22]–[27] to the *bidirectional* predictive trace analysis, BIPED, for exposing order violations. In general, a PTA technique records a trace of execution events, statically (often exhaustively) generates other permutations of these events under certain scheduling constraints, and then exposes concurrency bugs that are unseen in the recorded execution. Based on the general PTA technique, on one hand, BIPED finds the OUIs from the recorded trace, each of which represents a possible implement that allows to change the order of an *Use-Predef* relation, and then generates concrete schedules to re-execute the changed trace for exposing OUs. On the other hand, BIPED finds the PUIs from the recorded trace, each of which represents a possible implement that allows to change the order of an *Use-Redef* relation, and then generates concrete schedules to to re-execute the changed trace for exposing PUs. Since the full history and context information of an execution can be used adequately and all OUs and PUs reported are real, we believe BIPED can effectively expose order violations based on this execution.

The key technical challenge is how to find the *possible* OUIs and PUIs from the original trace to expose the corresponding OUs and PUs. Considering that possible OUIs and PUIs cannot directly be found in the original trace, we first present a simple and safe tactics to use for recognizing candidate OUIs and PUIs. A pair of check algorithms is subsequently presented to find possible OUIs and PUIs from the candidate ones, respectively. Moreover, to concentrate on the *Use-Predef* and *Use-Redef* relations for recognizing and checking the candidate OUIs and PUIs, we use a layered prediction model to represent the inter-thread data flow information in the bottom layer and the thread-scheduling constraints in the upper layer. With encoding of the happens-before relationship between the events bidirectionally, the upper layer model supports efficient check of the candidate OUIs and PUIs in parallel, driving BIPED to predict long trace with different directions. Figure 2 shows the overview of BIPED's basic operations.

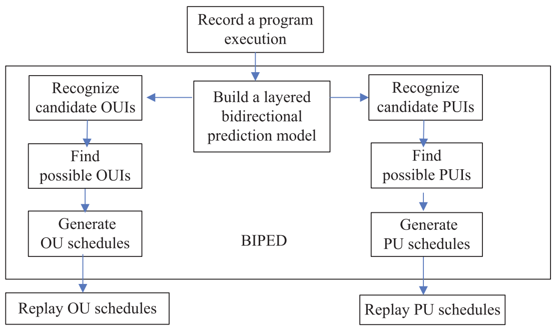


Fig. 2 Overview of BIPED's operation.

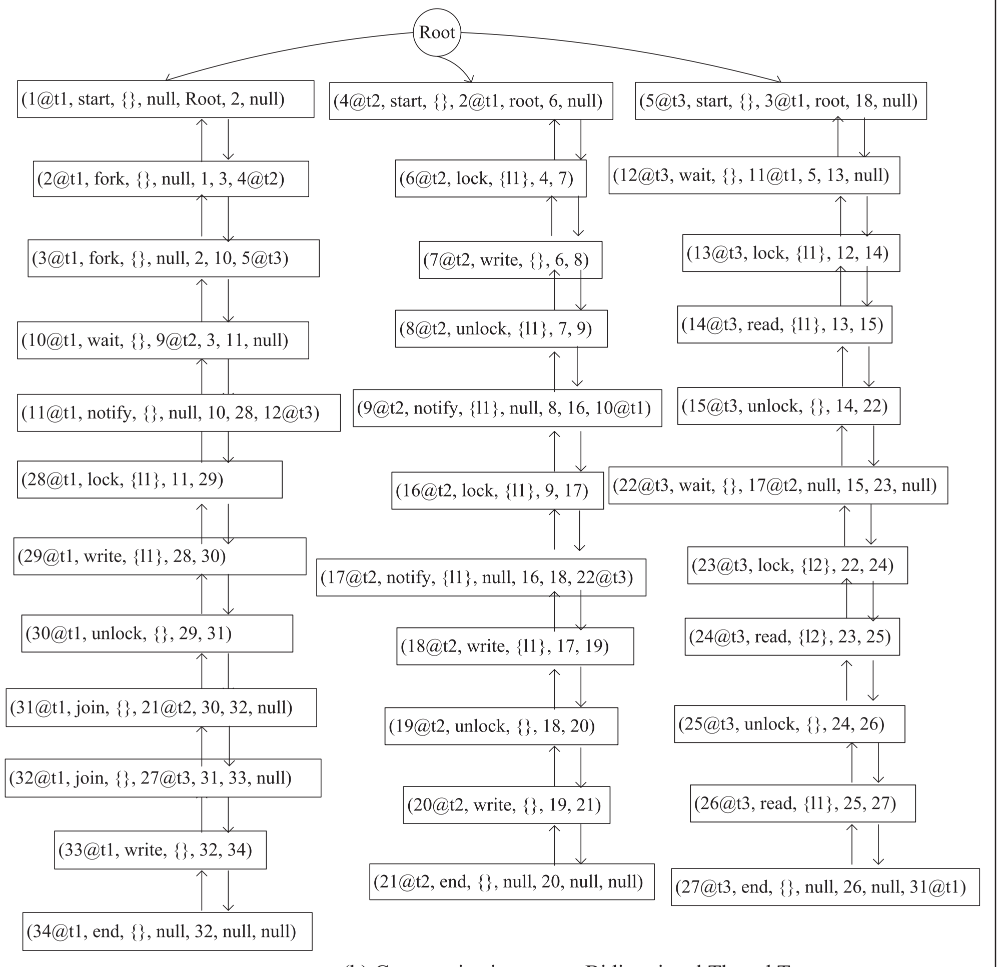
We have implemented BIPED for Java programs and conducted the experiments in IBM ConTest benchmark suite for evaluating it. Our evaluation results show that BIPED is able to effectively and efficiently predict the possible OUIs and PUIs to create concrete schedules for exposing order violation bugs in all the five evaluated subjects.

In summary, the main contributions of this work can be summarized as follows:

1. We present a bidirectional predictive trace analysis technique, BIPED, for exposing order violations in concurrent Java programs in parallel.
2. We present a layered prediction model, which distinguishes two types of expected data flow information and encodes the happens-before relations with two different directions. This model can ease the recognition of candidate violation intervals and bidirectionally check these two types of intervals.
3. We present a pair of statically prediction algorithms that respectively check whether the two types of candidate violation intervals satisfy the corresponding mutation condition.
4. We implement a prototype tool BIPED and our experiments demonstrate that BIPED is able to efficiently predict OUIs and PUIs and generate the concrete schedules to expose real order violations.

K	T	ET	Var
1	t1	start	
2	t1	fork	t2
3	t1	fork	t3
4	t2	start	
5	t3	start	
6	t2	lock	l1
7	t2	write	x
8	t2	unlock	l1
9	t2	notify	g1
10	t1	wait	g1
11	t1	notify	g2
12	t3	wait	g2
13	t3	lock	l1
14	t3	read	x
15	t3	unlock	l1
16	t2	lock	l2
17	t2	notify	g3
18	t2	write	y
19	t2	unlock	l2
20	t2	write	z
21	t2	end	
22	t3	wait	g3
23	t3	lock	l2
24	t3	read	y
25	t3	unlock	l2
26	t3	read	z
27	t3	end	
28	t1	lock	l1
29	t1	write	x
30	t1	unlock	l1
31	t1	join	t2
32	t1	join	t3
33	t1	write	y
34	t1	end	

(a) Normal Execution


Root
(1@t1, start, {}, null, Root, 2, null)
(2@t1, fork, {}, null, 1, 3, 4@t2)
(3@t1, fork, {}, null, 2, 10, 5@t3)
(10@t1, wait, {}, 9@t2, 3, 11, null)
(11@t1, notify, {}, null, 10, 28, 12@t3)
(28@t1, lock, {l1}, 11, 29)
(29@t1, write, {l1}, 28, 30)
(30@t1, unlock, {}, 29, 31)
(31@t1, join, {}, 21@t2, 30, 32, null)
(32@t1, join, {}, 27@t3, 31, 33, null)
(33@t1, write, {}, 32, 34)
(34@t1, end, {}, null, 32, null, null)
(4@t2, start, {}, 2@t1, root, 6, null)
(6@t2, lock, {l1}, 4, 7)
(7@t2, write, {}, 6, 8)
(8@t2, unlock, {l1}, 7, 9)
(9@t2, notify, {l1}, null, 8, 16, 10@t1)
(16@t2, lock, {l1}, 9, 17)
(17@t2, notify, {l1}, null, 16, 18, 22@t3)
(18@t2, write, {l1}, 17, 19)
(19@t2, unlock, {}, 18, 20)
(20@t2, write, {}, 19, 21)
(21@t2, end, {}, null, 20, null, null)
(5@t3, start, {}, 3@t1, root, 18, null)
(12@t3, wait, {}, 11@t1, 5, 13, null)
(13@t3, lock, {l1}, 12, 14)
(14@t3, read, {l1}, 13, 15)
(15@t3, unlock, {}, 14, 22)
(22@t3, wait, {}, 17@t2, null, 15, 23, null)
(23@t3, lock, {l2}, 22, 24)
(24@t3, read, {l2}, 23, 25)
(25@t3, unlock, {}, 24, 26)
(26@t3, read, {l1}, 25, 27)
(27@t3, end, {}, null, 26, null, 31@t1)

(b) Communication-aware Bidirectional Thread Tree

Use	Predef	Redef
14@t3	7@t2	29@t1
24@t3	18@t2	33@t1
26@t3	20@t2	

(c) Expected-order Usage Table

Fig. 3 An illustrative example.

2. An Illustration Example

We use in this section a simple example to illustrate how our bidirectional predictive trace analysis exposes the order violation bugs. The execution of a program in Fig. 3 (a) contains three threads, *t1*, *t2* and *t3*, and three shared variables *x*, *y* and *z*. Let us use the global index *K* as the identifier of each event. During this execution, thread *t1* creates two child threads *t2* and *t3* by event e_2 and event e_3, and joins back these child threads by event e_{31} and event e_{32}, respectively. In addition, these three threads must do some simple cooperation. Specifically, event e_{10} must wait for the signal that notified by event e_9, event e_{12} must wait for the signal that notified by event e_{11} and event e_{22} must wait for the signal that notified by event e_{17}. We use *Use-Predef*(e_j, e_i) for representing the *Use-Predef* relation between read event e_j and prior write event e_i, and *Use-Redef*(e_j, $e_{i'}$) for representing the *Use-Redef* relation between read event e_j and posterior write event $e_{i'}$. For this execution, it contains the following expected inter-thread data flows *Use-Predef*(e_{14}, e_7), *Use-Redef*(e_{14}, e_{29}), *Use-Predef*(e_{26}, e_{20}), *Use-Redef*(e_{24}, e_{33}), *Use-Predef*(e_{24}, e_{18}).

BIPED exposes the violations to these expected data flows by taking the following main four steps.

1) After recording a normal execution of a program, the first step is to take a recorded trace as the input to construct a layered prediction model. The layered prediction model of this sample trace is shown in Fig. 3:
   - The bottom layer is an expected-order usage table (EUT). Let us use the global index *K* and the

owned thread T as a unique identifier to denote an event owned by a thread. For each identifier $i@t'$ in the `Predef` column, it indicates a relation *Use-Predef(e_j, e_i)*, where the identifier $i@t'$ is in the same line with $j@t$ in the `Use` column. For each identifier $i'@t'$ in the `Redef` column, it indicates a relation *Use-Redef(e_j, $e_{i'}$)*, where the identifier $i'@t'$ is in the same line with $j@t$ in the `Use` column. Figure 3 (c) shows the expected inter-thread data flow information in the sample trace, identifying all the inter-thread *Use-Predef* relations and *Use-Redef* relations.

– The upper layer is a communication-aware bidirectional tree (`CBT`). Figure 3 (b) shows such tree of the sample trace built by BIPED. Each node in a `CBT` branch `CBT(t)` is also identified by the above unique identifier and can be (1) a shared-memory accessing (`MEM`) node corresponds to a **read** or **write** event; (2) a mutually exclusive synchronization (`MES`) node corresponds to a **lock** or **unlock** event; (3) an inter-thread order synchronization (`IOS`) node corresponds to a **wait**, **notify**, **fork**, **start**, **join** or **end** event. The attribute *local predecessor* (`lp`) and *local successor* (`ls`) of each node is used to bidirectionally denote a thread-local order relation $\prec_L$, such as a thread-local order relation $e_2 \prec_L e_3$ in the trace Fig. 3 (b), the identifier 2 is referred as the attribute `lp` of node $n_{3@t1}$, meanwhile, the identifier 3 is referred as the attribute `rs` of node $n_{2@t1}$. Each `IOS` node contains two auxiliary attributes (i.e., *remote predecessor* (`rp`) and *remote successor* (`rs`)) that are used to bidirectionally denote an inter-thread happens-before relation $\prec_R$, such as, for a happens-before relation $e_2 \prec_R e_4$ in the trace Fig. 3 (b), the identifier $2@t1$ is referred as the attribute `rp` of node $n_{4@t2}$, meanwhile, the identifier $4@t2$ is referred as the attribute `rs` of node $n_{2@t1}$.

2) The second step is to recognize the candidate OUIs and PUIs on the basis of the layered prediction model.
For each *Use-Predef* relation in the `EUT`, BIPED recognizes a backward sequence of consecutive nodes in the `CBT` as the corresponding candidate OUI to change the relative-order of this relation.

– A straightforward way to recognize the candidate OUI corresponding to a *Use-Predef* relation is to only enclose the consecutive nodes between the dependent node pair. For instance, the candidate OUI of the relation *Use-Predef(e_{26}, e_{20})* can be shown as follows:

$$OUI(26, t3, 20, t2) = n_{20@t2} \leftarrow n_{21@t2} \leftarrow n_{22@t3} \leftarrow n_{23@t3} \leftarrow n_{24@t3} \leftarrow n_{25@t3} \leftarrow n_{26@t3},$$

where this backward sequence is denoted as $OUI(26, t3, 20, t2)$ since the interval-latest node is node $n_{26@t3}$, the interval-earliest node is node $n_{20@t2}$, and the goal of this interval is to move event e_{26} to a position before event e_{20}.

– For some special cases, the straightforward way is insufficient for determining whether the relative order of the *Use-Predef* relation can be changed or not, because of the affection by the mutual exclusive constraints. For instance, the candidate OUI of the relation *Use-Predef(e_{24}, e_{18})* is as follows:

$$n_{18@t2} \leftarrow n_{19@t2} \leftarrow n_{20@t2} \leftarrow n_{21@t2} \leftarrow n_{22@t3} \leftarrow n_{23@t3} \leftarrow n_{24@t3}.$$

It cannot make sure that event e_{24} can move to the position before e_{18} even if there does not exist any order constraint between e_{24} and e_{18} because the position immediately before event e_{18} is restricted to this movement by the acquired lock of e_{18}. Thus, we adopt some recognition tactics given in Sect. 4.1 to expand the candidate OUI of this relation as follows:

$$OUI(25, t3, 16, t2) = n_{16@t2} \leftarrow n_{17@t2} \leftarrow n_{18@t2} \leftarrow n_{19@t2} \leftarrow n_{20@t2} \leftarrow n_{21@t2}, n_{22@t3} \leftarrow n_{23@t3} \leftarrow n_{24@t3} \leftarrow n_{25@t3}.$$

For each inter-thread *Use-Redef* relation in the `EUT`, the candidate PUI of an Use-Redef relation is a forward sequence of consecutive nodes of the CBT that encloses this relation at least. The recognition tactics for the corresponding candidate PUI given in Sect. 4.1 are symmetrical to that of inter-thread Use-Predef relation, we can recognize the following candidate PUIs, which correspond to the relations *Use-Redef(e_{14}, e_{29})* and *Use-Redef(e_{24}, e_{33})*, respectively.

$$PUI(30, t1, 13, t3) = n_{13@t3} \rightarrow n_{14@t3} \rightarrow n_{15@t3} \rightarrow n_{16@t2} \rightarrow n_{17@t2} \rightarrow n_{18@t2} \rightarrow n_{19@t2} \rightarrow n_{20@t2} \rightarrow n_{21@t2} \rightarrow n_{22@t3} \rightarrow n_{23@t3} \rightarrow n_{24@t3} \rightarrow n_{25@t3} \rightarrow n_{26@t3} \rightarrow n_{27@t3} \rightarrow n_{28@t1} \rightarrow n_{29@t1} \rightarrow n_{30@t1}$$

$$PUI(33, t1, 23, t3) = n_{23@t3} \rightarrow n_{24@t3} \rightarrow n_{25@t3} \rightarrow n_{26@t3} \rightarrow n_{27@t3} \rightarrow n_{28@t1} \rightarrow n_{29@t1} \rightarrow n_{30@t1} \rightarrow n_{31@t1} \rightarrow n_{32@t1} \rightarrow n_{33@t1}.$$

3) The third step is to find the possible OUIs/PUIs through checking the candidate OUIs/PUIs based on the predecessor/successor information in the upper model.
For each candidate OUI, BIPED checks whether this OUI is possible through backward computing the predecessor relation between $n_{k_\top @t_\top}$ and $n_{k_\bot @t_\bot}$. A straightforward way to check the predecessor relation is to only check between two owner threads of the relation pair. For some special cases, the transitive predecessor relation between the relation pair cannot be immediately found by adopting the straightforward way.

For instance, *OUI(15, t3, 6, t2)* is impossible because there is a transitive predecessor relation between $n_{6@t2}$ and $n_{15@t3}$:

$$n_{6@t2} \prec_L n_{9@t2} \prec_R n_{11@t1} \prec_R n_{15@t3}.$$

This relation is difficult to be found because this relation depends on the intermediate thread, i.e., t2. Therefore, besides two owner threads of the relevant nodes, we should check the predecessor relation relevant to the intermediate threads. A detailed explanation of the check algorithm is given in Sect. 4.2. Consequently, BIPED can predict that only one following OUI is possible:

$$OUI(26, t3, 20, t2)$$

For each candidate PUI, BIPED checks whether this PUI is possible through forward computing the successor relation between $n_{k_\top @t_\top}$ and $n_{k_\perp @t_\perp}$. For instance, BIPED can determine *PUI(33, t1, 23, t3)* is impossible because there is a transitive successor relation between $n_{33@t1}$ and $n_{23@t3}$:

$$n_{23@t3} \succ_L n_{27@t3} \succ_R n_{32@t1} \succ_R n_{33@t1}.$$

Consequently, BIPED can predict that only the following PUI is possible:

$$PUI(30, t1, 13, t3)$$

4) Taking these possible OUIs/PUIs, the fourth step is to generate concrete possible schedules.
For the possible interval *OUI(26, t3, 20, t2)*, BIPED permutates the original subtrace within the corresponding interval into a read-backward-move subtrace as follows:

$$\langle e_{22}, e_{23}, e_{24}, e_{25}, e_{26}, e_{20}, e_{21} \rangle.$$

For the possible interval CPUI(13, t3, 30, t1), BIPED permutates the original subtrace within the corresponding interval into a read-forward-move subtrace as follows:

$$\langle e_{28}, e_{29}, e_{30}, e_{13}, e_{14}, e_{15}, e_{16}, e_{17},$$
$$e_{18}, e_{19}, e_{20}, e_{21}, e_{22}, e_{23}, e_{24}, e_{25}, e_{26}, e_{27} \rangle.$$

It can be seen that the above concrete schedules can expose the real order violations according to a normal execution trace. Compared with other dynamic analysis techniques, for an execution, BIPED can expose all OUs and PUs effectively in this execution trace, instead of only one order violation for each shared-memory read event at most. Compared with other PTA techniques, BIPED can predict the OUs and PUs simultaneously, It can improve the prediction performance.

3. Prediction Model

Based on an execution trace, we present a new prediction model called expectation-based bidirectional constraint model to simultaneously predict OU and PU bugs. In this section, the representation of the trace is formalized first and then the layered prediction model is defined.

3.1 Trace

Given an execution E of a multi-threaded program P, it reflects a multi-threaded program execution E = $\langle e_k \rangle$ over the set of events. An event is one of the following forms.

- (k, t, *read*, *sv*) is the k^{th} event that read the shared variable *sv*.
- (k, t, *write*, *sv*) is the k^{th} event that wrote to the shared variable *sv*.
- (k, t, *lock*, l) is the k^{th} event that acquired the lock l.
- (k, t, *unlock*, l) is the k^{th} event that released the lock l.
- (k, t, *fork*, t') is the k^{th} event that created the thread t'.
- (k, t, *join*, t') is the k^{th} event that joined back thread t'.
- (k, t, *wait*, m) is the k^{th} event that finished waiting for some signal m.
- (k, t, *notify*, m) is the k^{th} that notified an event waiting on m.
- (k, t, *start*) is the k^{th} event that started a thread t.
- (k, t, *end*) is the k^{th} event that terminated a thread t.

In our presentation, we use *t(k)*, *et(k)* to denote the owner thread, the event type of the k^{th} event, respectively.

3.2 Expectation-Based Bidirectional Constraint Model

In this research we define an *expectation-based bidirectional constraint model* to represent the concurrent behaviors.

Let *Tr* denote a multi-threaded execution trace of a program. The expectation-based bidirectional constraint model of *Tr* contains two layers: the bottom layer model uses a table for depicting the expected data flow cross threads, the upper layer model uses tree structure for describing how multiple threads are restricted into *Tr*.

3.2.1 The Bottom Layer

To represent the expected data flow cross threads, we first define two types of expected use-definition relations.

Definition 1: The expected use-definition relations contain two types of association relations

$$Use - Predef(e_j, e_i)$$

$$Use - Redef(e_j, e_{i'})$$

where event $e_i/e_{i'}$ is the last/first write event before the read event e_j, they access the same shared variable and are owned by different threads.

An expected-order usage table (`EUT`) is designed to represent these two types of expected use-definition relations. This table contains three columns ⟨`Use`, `Predef`, `Redef`⟩, where the column `Use` is the key column. Each line *(j@t(j),* *i@t(i), i′@t(i′))* can indicate an *Use-Predef* relation between e_j and e_i and an *Use-Redef* relation between e_j and $e_{i'}$. We should note that every element in the EUT combines the global index and the owned thread of the event, as the combination can facilitate the identification of the corresponding nodes in the upper layer model (see Sect. 3.2.2).

3.2.2 The Upper Layer

Many race detection techniques use a hybrid constraint model [28] that combines the lockset condition [1] and the happens-before relation [29] to predict potential races. Based on the hybrid constraint model, we encode a happens-before relation bidirectionally to define a bidirectional hybrid constraint model in order to predict possible OUIs and PUIs, respectively.

Specifically, two events e_i and e_j are independent iff

1. Mutual exclusive constraint:
   they do not hold a common lock.
2. Order constraint:
   - e_i is not a *predecessor* of e_j, where $i < j$.
   - e_j is not a *successor* of e_i, where $j > i$.

The *predecessor* and *successor* relations are defined as follows:

Definition 2: Given two events e_i and e_j (i < j), event e_i is the *predecessor* of event e_j, meanwhile, event e_j is the *successor* of event e_i, if one of the following conditions holds:

- Thread-local order constraints: e_i and e_j are events from the same thread,
- Inter-thread order constraints:
  - e_i is the event that forked the thread t, and e_j is the start event of thread t.
  - e_i is the end event of thread t and e_i is the event that joined back thread t.
  - e_i is the event that notified the signal g and e_j is the event that finished waiting the signal g.
- The *predecessor* and *successor* relations are transitively closed.

We design a communication-aware bidirectional tree to encode the lockset and the predecessor successor relation of every node bidirectionally.

Definition 3: A Communication-aware Bidirectional Tree (`CBT`) (`Root`, `CBT(t1)`, `CBT(t2)`, ..., `CBT(tn)`) contains a root node and a set of branches, where each branch `CBT(t)` corresponds to a thread `t`. A node in a path `CBT(t)` which corresponds to an event, can be one of the following nodes:

- An inter-thread order synchronization node ⟨`k@t`, `et`, `L`, `rp`, `lp`, `ls`, `rs`⟩ representing an inter-thread order synchronization event, `et` ∈ {**wait**, **notify**, **fork**, **join**, **start**, **end**},
- A mutual exclusive synchronization node ⟨`k@t`, `et`, `L`, `lp`, `ls`⟩, `et` ∈ {**lock**, **unlock**},
- A shared-memory accessing node ⟨`k@t`, `et`, `L`, `lp`, `ls`⟩, `et` ∈ {**read**, **write**},

where

- `k` is the global order of event e in the trace,
- `t` is the owner thread executing event e,
- `L` is the acquired locks of event e,
- `rp` refers to the node index of a remote predecessor node,
- `lp` refers to the node index of a local predecessor node,
- `ls` refers to the node index of a local successor node,
- `rs` refers to the node index of a remote successor node.

A `CBT` provides engineers with supports in helping identify a set of trace intervals.

Definition 4: A trace interval, denoted as $\mathrm{I}(k_\perp, t(k_\perp), k_\top, t(k_\top))$, is defined as a bidirectional sequence of nodes between the interval-earliest node $n_{k_\perp @ t(k_\perp)}$ and the interval-latest node $n_{k_\top @ t(k_\top)}$:

$$n_{k_\perp @ t(k_\perp)} \rightleftarrows \ldots \rightleftarrows n_{k_\top @ t(k_\top)}.$$

In the following presentation, *n.k* and *n.t* denote the global index and the owner thread associated with a node *n*, respectively. Moreover, *RP(t)* and *RS(t)* denote the set of remote predecessor nodes and remote successor nodes of the thread *t* in `CBT`, and $I(k_\perp, t(k_\perp), k_\top, t(k_\top)).nodes(t)$ denotes the nodes owned by thread *t* within this interval.

Based on the model, the prediction demands can be separated into a pair of sets for predicting their respective ones. For each demand set, the cost is linear to the length of the corresponding interval of trace and quadratic to the number of inter-thread order synchronization events.

4. Expectation-Violated Bidirectional Prediction

Based on Lamports reduction theory [29], the idea behind bidirectional prediction is that the relative positions of independent events in the trace can be changed. It simulates the different thread scheduling effects. Specifically, a *Use-Predef/Use-Redef* relation is witnessed in a normal trace, as long as an interval of the trace enclosing this relation allows to change the relative order of this relation, we can generate a concrete schedule to expose the OU/PU.

To facilitate our discussion, we first define two concepts called candidate *OUI* and *PUI* that will be used in the explanation of our approach:

Definition 5: For a relation Use-Redef(e_j, e_i), the *candidate OUI* corresponding to it, denoted as $\mathrm{OUI}(k_\top, t(k_\top), k_\perp, t(k_\perp))$, is a backward trace interval from the interval-latest node $n_{k_\top @ t(k_\top)}$ to the interval-earliest node $n_{k_\perp @ t(k_\perp)}$:

$$n_{k_\perp @ t(k_\perp)} \leftarrow \ldots \leftarrow n_{k_\top @ t(k_\top)},$$

where $k_\perp < k_\top$, $k_\perp \le i$ and $k_\top \ge j$.

Definition 6: For a relation Use-Redef(e_j, $e_{i'}$), the *candidate PUI* corresponding to it, denoted as $\mathrm{PUI}(k_\top, t(k_\top), k_\perp, t(k_\perp))$, is a forward trace interval from the interval-earliest

Algorithm 1: LookforSafePos($k_\top$, $t(k_\top)$, i, $t(i)$)

```
begin                                                                1
    $n_{x@t(i)} \leftarrow n_{i@t(i)}$;                               2
    while $n_{x@t(i)} \neq Root$ do                                   3
        $count \leftarrow 1$;                                         4
        for each $n \in |I(i, t(i), k_\top, t(k_\top)).nodes(t(i))|$ do   5
            if $n_{x@t(i)}.lp.L \cap n.L == \emptyset$ then           6
                $count \leftarrow count + 1$;                         7
            else                                                      8
                break;                                                9
        if $count == |I(i, t(i), k_\top, t(k_\top)).nodes(t(i))|$ then   10
            return $n_{x@t(i)}$                                       11
        $n_{x@t(i)} \leftarrow n_{x@t(i)}.lp$;                        12
    return null                                                       13
```

node $n_{k_\perp @t(k_\perp)}$ to the interval-latest node $n_{k_\top @t(k_\top)}$:

$$n_{k_\perp @t(k_\perp)} \rightarrow \ldots \rightarrow n_{k_\top @t(k_\top)},$$

where $k_\perp < k_\top$, $k_\perp \leq j$ and $k_\top \geq i'$.

This section afterwards describes the bidirectional predictive trace analysis technique involving the following steps:

- Lockset-based recognizing the candidate OUIs and PUIs.
- Checking whether these candidate OUIs and PUIs are possible, respectively.
- Generating concrete OU and PU schedules according to the possible OUIs and PUIs.
- Pruning false OUIs and PUIs.

4.1 Recognizing Candidate OUIs and PUIs

Since the original trace is a possible schedule (i.e., satisfying the order constraints and mutual exclusive constraints), it only needs to make sure a candidate OUI/PUI does not violate the constraints, instead of the entire trace. The key problem is how to recognize the interval-earliest node and the internal-latest node of the candidate OUI/PUI. Without loss of generality, we use the relation *Use-Predef*(e_{24}, e_{18}) in Fig. 3 to illustrate how our tactics respects the interval-earliest node and the internal-latest node.

Recognizing the interval-latest node is relatively simple. If we assign node $n_{24@t3}$ as the interval-latest node of the candidate OUI, it means that the nodes corresponding to the to-be-moved events (i.e., $n_{22@t3}$, $n_{23@t3}$ and $n_{24@t3}$) should be allowed to place some positions before $n_{18@t2}$ to change the relative order of *Use-Predef*(e_{24}, e_{18}). As the unlock event (corresponding to node $n_{25@t3}$) releasing the acquired lock of event e_{24} isn't placed together with these to-be-moved events, it can lead to the result that there exists an *unmatched* lock and unlock event pair in this interval, so that the generated schedule may violate the mutual exclusive constraints. To address this problem, *whenever we enclose a lock node to this candidate OUI, we should also make sure its corresponding unlock node is enclosed to this one*. Thus, our recognition tactics on the interval-latest node looks for the nearest lock-free position (NLF), i.e., $n_{25@t3}$ releases the required locks of the thread t3, and assigns this unlock node as the internal-latest node of the candidate OUI, i.e., $n_{k_\top @t_{k_\top}} = n_{25@t3}$.

Recognizing the interval-earliest node is much more complicated. We have already recognized the internal-latest node of the candidate OUI, i.e., $n_{25@t3}$, it means that the to-be-moved nodes are *I(18, t2, 25, t3).nodes(t3)*. It is important to make sure that these to-be-moved nodes can move to some safe positions without violating the mutual exclusive constraints. Although there might be many possible ways which can implement this movement, it is sufficient for us to adopt one safest and simplest way to reduce the computational complexity of the candidate OUI recognition. Specifically, we *look for a safe position before the prior relation node, where all the to-be-moved events can move to*. With this tactics, we define a safe position as follows:

Definition 7: Given the to-be-moved nodes *I*($k_\perp$, $t(k_\perp)$, $k_\top$, $t(k_\top)$)*.nodes*($t(k_\top)$), a safe position just before the node $n_{x@t(k_\perp)}$ does not have any common lock with each of the to-be-moved nodes.

Specifically, given a set of the to-be-moved nodes *I(18, t2, 25, t3).nodes(t3)*, we try to look for a safe position before $n_{x@t2}$, following the branch `CBT(t2)`, there is no common lock with each of the to-be-moved nodes (Algorithm 1). Finally, we find $n_{x@t2} = n_{16@t2}$.

Table 1 summarizes the tactics to recognize the candidate OUI/PUI for each *Use-Predef/Use-Redef* relation. Since each candidate OUI/PUI is recognized with the safest tactic, it can ensure that the interval without violating the mutual exclusive constraints, we then check whether this candidate OUI can satisfy the order constraints.

4.2 Interval-Restricted Bidirectional Check

After recognizing the candidate OUIs and PUIs, BIPED next checks whether these intervals can be possible, that is, there exists a predecessor/successor relation within each interval.

We first define the four data structures and four primitive operations on these data structures:

- The set *n.VRP* consists of all remote predecessors of the node n within the interval between n and $n_{k_\top @t(k_\top)}$, which can be obtained according to the `rp` attributes of branch `CBT(n.t)`.
- The set *n.VRS* consists of all remote successors of the node n within the interval between n and $n_{k_\perp @t(k_\perp)}$, which can be obtained according to the `rs` attributes of branch `CBT(n.t)`.
- The set *n.CRP* consists of the current remote predecessors of node n.
- The set *n.CRS* consists of the current remote successors of node n.

Table 1 Recognition tactics.

IN	Use-Predef(e_j, e_i)		Use-Redef(e_j, $e_{i'}$)	
L-condition 1	L(j) ≠ null	L(j) = null	L(i′) ≠ null	L(i′) = null
$K_\top$	$k_\top$ = NLF(j)	$k_\top$ = j	$k_\top$ = NLF(i′)	$k_\top$ = i′
$T_\top$	$t_\top$ = t(j)	$t_\top$ = t(j)	$t_\top$ = t(i′)	$t_\top$ = t(i′)
L-condition 2	L(i) ≠ null	L(i) = null	L(j)≠ null	L(j) = null
$K_\perp$	$k_\perp$ = LookforSafePos($k_\top@t(k_\top)$, i@t(i))	$k_\perp$ = i	$k_\perp$ = LookforSafePos($k_\top@t(k_\top)$, j@t(j))	$k_\perp$ = j
$T_\perp$	$t_\perp$ = t(i)	$t_\perp$ = t(i)	$t_\perp$ = t(j)	$t_\perp$ = t(j)

Algorithm 2: CanBackwardMutate($k_\top$, $t_\top$, $k_\perp$, $t_\perp$)

```
begin                                                                         1
    $n_{k_\top@t_\top}.CRP \leftarrow getVRP(k_\perp, n_{k_\top@t_\top})$;    2
    while $n_{k_\top@t_\top}.CRP \neq \varnothing$ do                         3
        for $n \in n_{k_\top@t_\top}.CRP$ do                                  4
            if $IsLP(n_{k_\perp@t_\perp}, n)$ then                            5
                return false;                                                 6
        for $n \in n_{k_\top@t_\top}.CRP$ do                                  7
            $n.VRP \leftarrow getVRP(k_\perp, n)$;                            8
            if $n.VRP \neq \varnothing$ then                                  9
                $n_{k_\top@t_\top}.CRP \leftarrow n_{k_\top@t_\top}.CRP \cup n.VRP$ ;   10
            $n_{k_\top@t_\top}.CRP \leftarrow n_{k_\top@t_\top}.CRP/n$ ;      11
    return true                                                               12
```

Algorithm 3: CanForwardMutate($k_\top$, $t_\top$, $k_\perp$, $t_\perp$)

```
begin                                                                         1
    $n_{k_\perp@t_\perp}.CRS \leftarrow getVRS(n_{k_\perp@t_\perp}, k_\top)$;  2
    while $n_{k_\perp@t_\perp}.CRS \neq \varnothing$ do                       3
        for $n \in n_{k_\perp@t_\perp}.CRS$ do                                4
            if $IsLS(n, n_{k_\top@t_\top})$ then                              5
                return false;                                                 6
        for $n \in n_{k_\perp@t_\perp}.CRS$ do                                7
            $n.VRS \leftarrow getVRS(n, k_\top)$;                             8
            if $n.VRS \neq \varnothing$ then                                  9
                $n_{k_\perp@t_\perp}.CRS \leftarrow n_{k_\perp@t_\perp}.CRS \cup n.VRS$;   10
            $n_{k_\perp@t_\perp}.CRS \leftarrow n_{k_\perp@t_\perp}.CRS/n$;   11
    return true;                                                              12
```

- The operation *getVRP($k_\perp$, $n_{y@t}$)* returns the set of visible remote predecessors of node $n_{y@t}$. The global indexes of the nodes $n_{y@t}.VRP$ are greater than $k_\perp$ and belong to the remote predecessors of thread t, i.e.,

$$\{\forall n \in n_{y@t}.VRP | (n \in RP(t)) \land (k_\perp < n.k < y)\}.$$

- The operation getVRS($n_{y@t}$, $k_\top$) returns the set of visible remote successors of node $n_{y@t}$. The global indexes of $n_{y@t}.VRS$ are smaller than $k_\top$ and belong to the remote successors of thread t, i.e.,

$$\{\forall n \in n_{y@t}.VRS | (n \in RS(t)) \land (y < n.k < k_\top)\}.$$

- The operation *IsLP($n_{k_\perp@t(k_\perp)}$, $n_{y@t}$)* checks whether $n_{k_\perp@t(k_\perp)}$ is a local predecessor of node $n_{y@t}$. If node $n_{y@t}$ is owned by thread $t(k_\perp)$ and y is greater than $k_\perp$, this operation returns true, otherwise, returns false.
- The operation *IsLS($n_{y@t}$, $n_{k_\top@t(k_\top)}$)* checks whether $n_{k_\top@t(k_\top)}$ is a local successor of node $n_{y@t}$. If node $n_{y@t}$ is owned by thread $t(k_\top)$ and y is smaller than $k_\top$, this operation returns true, otherwise, returns false.

A candidate interval *OUI($k_\top, t_\top, k_\perp, t_\perp$)* is impossible if there exists a predecessor relation between $n_{k_\top@t(k_\top)}$ and $n_{k_\perp@t(k_\perp)}$. We transform the feasibility check on a candidate OUI to backward compute the predecessor relation from the interval-latest node $n_{k_\top@t(k_\top)}$ to node $n_{k_\perp@t(k_\perp)}$. Algorithm 2 shows our backward check algorithm for finding possible OUIs. Given an interval *OUI($k_\top, t_\top, k_\perp, t_\perp$)*, set $n_{k_\top@t(k_\top)}.CRP$ is first initialized to all of its visible remote predecessors (line 1). The algorithm then enters the iterative process. In lines 4-6, the algorithm calls operation *IsLP* to check whether the interval-earliest node $n_{k_\perp@t(k_\perp)}$ is a local predecessor of each node in set $n_{k_\top@t(k_\top)}.CRP$. If the application of operation *IsLP* returns true, it means that node $n_{k_\perp@t(k_\perp)}$ is a predecessor of node $n_{k_\top@t(k_\top)}$ because the following predecessor relation can be found:

$$n_{k_\perp@t(k_\perp)} \prec_L n \prec_R n_{k_\top@t(k_\top)}.$$

This algorithm returns false and is terminated. Otherwise, this algorithm proceeds to the following progressive check. For each node n in set $n_{k_\top@t(k_\top)}.CRP$, $n_{k_\top@t(k_\top)}.CRP$ adds the nodes returned from operation *getVRP($k_\perp$, n)*, if the returned set is not empty. After that, node n is removed from set $n_{k_\top@t(k_\top)}.CRP$ as the node cannot introduce any newly predecessors (see line 7-11). The algorithm executes the iterative process until set $n_{k_\top@t(k_\top)}.CRP$ is an empty set and then returns true.

A candidate interval *PUI($k_\top, t_\top, k_\perp, t_\perp$)* is impossible if there exists a successor relation between $n_{k_\top@t(k_\top)}$ and $n_{k_\perp@t(k_\perp)}$. We transform the feasibility check on an PUI to forward compute the successor relation between $n_{k_\top@t(k_\top)}$ and $n_{k_\perp@t(k_\perp)}$, starting from the interval-earliest node $n_{k_\perp@t(k_\perp)}$. Algorithm 3 shows our forward check algorithm for finding possible PUIs. Given an interval *PUI($k_\top, t_\top, k_\perp, t_\perp$)*, set $n_{k_\perp@t_\perp}.CRS$ is initialized to all of its visible remote successors (line 1). The algorithm then enters the iterative process. In lines 4-6, the algorithm calls operation *IsLS* to check whether the interval-latest node $n_{k_\top@t_\top}$ is a local successor of each node n in set $n_{k_\perp@t_\perp}.CRS$. If the application of operation *IsLS* returns true, it means that $n_{k_\top@t_\top}$ is a successor of $n_{k_\perp@t_\perp}$ because the following successor relation can be found:

$$n_{k_\perp@t_\perp} \succ_R n \succ_L n_{k_\top@t_\top}.$$

This algorithm returns false and is terminated. Otherwise, this algorithm proceeds to the following progressive check. For each node n in set $n_{k_\perp @ t_\perp}.CRS$, $n_{k_\perp @ t_\perp}.CRS$ adds the nodes returned from operation *getVRS(n, $k_\top$)*, if the returned set is not empty. After that, node n is removed from set $n_{k_\perp @ t_\perp}.CRS$ since the node cannot introduce any newly successor (see line 7-11). This algorithm executes the iterative process until set $n_{k_\perp @ t_\perp}.CRS$ is an empty set and returns true.

Let us revisit the process on checking the interval *OUI(15, t3, 6, t2)* in our sample example. To check the interval *OUI(15, t3, 6, t2)*, the key steps are illustrated as follows:

1. Given the interval-latest node, i.e., node $n_{15@t3}$, the visible remote predecessors of $n_{15@t3}$, i.e., $\{n_{11@t1}\}$, which can obtain by following the branch `CBT(t1)`. Then, the current remote predecessors of node $n_{15@t3}$ is initialized to set $n_{15@t3}.VRP$.
2. Operation *IsLP($n_{6@t2}$, $n_{15@t3}.CRP$)* returns true, since node $n_{6@t2}$ is not a local predecessor of node $n_{11@t1}$. This check process thus proceeds to step 3.
3. All visible remote predecessors of node $n_{11@t3}$ are obtained by following the branch `CBT(t3)`, i.e., $\{n_{9@t2}\}$, and add it to the set of current remote processors of $n_{15@t3}$, i.e., $n_{15@t3}.CRP$. Whereafter, node $n_{11@t3}$ is removed from set $n_{15@t3}.CRP$, that is, $n_{15@t3}.CRP$ is updated to $\{n_{9@t2}\}$.
4. Recall step 2. This check returns false and is terminated because the node $n_{6@t2}$ is a local predecessor of node $n_{9@t2}$.

Therefore, BIPED can determine that *OUI(15, t3, 6, t2)* is impossible.

4.3 Possible Schedule Generation

For each Possible OUI/PUI, BIPED statically generates a Possible thread schedule that is used to deterministically direct an execution for exposing the OU/PU bug. To generate a concrete schedule, BIPED takes the recorded trace and the possible OUIs/PUIs as input, and use the read-backward-move/read-forward-move sequences to represent the OU/PU schedules. For each possible *OUI/PUI($k_\top$, t($k_\top$), $k_\perp$, t($k_\perp$))*, BIPED moves all the events owned by $t_\perp$ within this interval to the position before event $e_{k_\top}$ to generate a corresponding schedule. The generated schedules are guaranteed to possible, without violating mutual exclusion constraints or the order constraints.

With aspect to order constraints, the OUI/PUI corresponding to a generated schedule has already checked that there does not exist a predecessor/successor relation between the interval-earliest event and the interval-latest event. Moreover, the to-be-moved thread-local events within the interval are moved together to the same position. With aspect to mutual exclusion constraints, a safe position ensures that there is no common lock with all of the to-be-moved events. Since all the schedules are generated by moving the sequences of events to different positions, the worst case time complexity of the total of two schedule generation algorithms is linear in the length of the trace.

4.4 Pruning False OU/PU Schedules

The possible OUIs/PUIs are sound in term of satisfying thread schedule constraints but neglect the program control constraints, i.e., it may generate impossible schedules for false violations. BIPED can automatically prune all the false OU/PU schedules away during the re-execution phase. Similar to J. Huang's work [27], we control the thread scheduling of the re-execution to strictly follow an input generated schedule by comparing the events between the two schedules. When we observe that some thread has executed a new event that does not exist in the input schedule, which means the thread has taken a different branch from the original observed execution. We immediately stop the re-execution and remove this OUI/PUI. In this way, we are able to prune all the false violations as we only report successful re-executions.

5. Implementation and Evaluation

5.1 Implementation

BIPED provides programmers with support in bidirectional predictive trace analysis of Java programs by the following six steps:

1. Use soot [30] instrument all the program points that may involve the relevant events in the trace. Specifically, we instrument all the possible shared variable access points, all the monitor entry and exit points to track the shared variable access events, the lock acquisition and release events, during the program execution. To track the thread communication events, we instrument thread fork and join, thread start and exit, and object wait and notify points.
2. One event vector is used for recording the global order of all the events, and it is maintained to collect the trace. For each inter-thread order synchronization event, the global ID, event type, the owned thread ID and the relevant object of it are recorded to encode the remote predecessor and successor relations. For each memory accessing event, the global ID, event type, the owned thread ID and the relevant object instance are recorded to collect the expected-order data flow information. The lockset associated with every event is computed offline to save runtime cost.
3. The prediction model is constructed by visiting the collected trace.
4. Based on the prediction models, the candidate OUIs/PUIs are recognized and checked.
5. Generate the corresponding schedulers according to the possible OUIs/PUIs.
6. For the replaying process, following the generated schedule read-backward-move/read-forward-move se-

Table 2 Experimental results.

Program	Trace			Order violations			Consumed Time (ms)		
	Thr	SV	E	BIPED	CALLFUZZER	PECAN	BIPED	CALLFUZZER	PECAN
Critical	3	3	105	7	7	7	3.84	13.3	3.99
BuggyPrg	5	5	460	6	6	6	8.2	18.9	10.2
Shop	4	4	450	7	7	7	6	13	8
Mergesort	6	10	692	37	32	37	15	62.3	21
Bubblesort	26	25	5069	68	54	68	115	1002	213

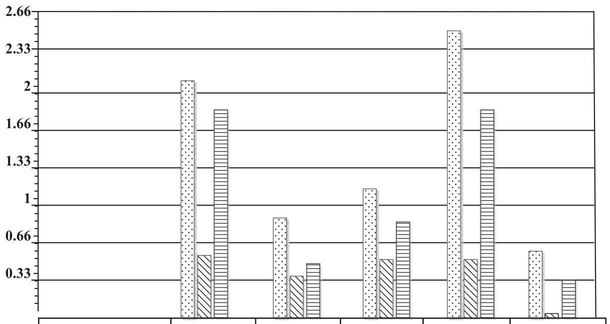
quences, we use two replayers to deterministically control the scheduling of threads for exposing OU and PU bugs, respectively.

5.2 Evaluation

To evaluate the effectiveness and efficiency of our proposed technique, we experimented on a number of multi-threaded Java benchmarks of widely-used multi-threaded Java benchmarks, IBM ConTest benchmark suite [31], including: Critical, BuggyPrg, Shop, Mergesort, Bubblesort. All experiments were carried out on a laptop equipped with a Intel 2.4GHz i3-2370M processor and 4GB memory, using Red Hat′s Fedora 64-bit Linux (version 2.6.27) and Oracle's Java HotSpot 64-bit Server VM (version 1.6.0). To properly compare our technique to the state of the art, we have also implemented the following techniques: a predictive dynamic analysis approach CALLFUZZER [32], and a general static predictive trace analysis approach PECAN [27]. Both of them also use the trace information to detect concurrency bugs in Java programs. In order to make them comparable to our technique, we extended CALLFUZZER capability and specified the order violation patterns to PECAN.

We summarizes the experiment results of the related techniques in Table 2. We use the number of threads, the number of shared variables and the number of critical events to denote the scale of the analyzed trace. The column `Order violations` shows respectively the number of the found order violation bugs by BIPED, CALLFUZZER and PECAN. The column `Comsumed time` shows respectively the consumed time by BIPED, CALLFUZZER and PECAN, where the consumed time of CALLFUZZER is the time that the trace is executed 100 times. According to the experimental results, BIPED and PECAN can detect a smaller number of order violation bugs than , because it is essentially a randomized technique that dynamically explores certain specific thread schedules from an ocean of thread interleavings, its capability of exposing real bugs is subjected to the randomness and some bugs cannot be found in a limited times of execution. Just as we can recognize that in program Bublesort, BIPED and PECAN detect 68 faults while CALLFUZZER only seeks 54 faults. In addition, through both of BIPED and PECAN can found the same number of the bugs, PECAN requires the users to specify their buggy patterns for predicting these order violations, BIPED doesn't need to any pattern information.

In addition, Fig. 4 shows a comparison among BIPED, CALLFUZZER and PECAN on the prediction speed in creating real order violations, where the prediction speed (PS) is defined as a ratio of the number of the found order violation bugs to the corresponding consumed time. According to the experimental results, the prediction speed of BIPED is 2.3x-10.98x relative to CALLFUZZER and 1.2x-1.8x relative to PECAN. Hence, with respect to find order violations, BIPED can be more efficient than CALLFUZZER and PECAN.



	Critical	Buggy	Shop	Mergesort	Bubblesort
PS_{BIPED} ⊡	2.19	0.73	1.17	2.47	0.59
$PS_{CALLFUZZER}$ ▧	0.54	0.32	0.47	0.51	0.05
PS_{PECAN} ⊟	1.75	0.59	0.88	1.76	0.32
$\frac{PS_{BIPED}}{PS_{CALLFUZZER}}$	4.06	2.3	2.5	4.8	10.98
$\frac{PS_{BIPED}}{PS_{PECAN}}$	1.24	1.24	1.33	1.4	1.85

Fig. 4 The comparison of BIPED, CALLFUZZER and PECAN.

5.2.1 Limitations of BIPED

Through our experimental results, we have clearly demonstrated the efficient concurrency bug prediction capability of BIPED with aspect to the order violations, we also observed some limitations of BIPED that we plan to address in our future work.

Sensitivity to the normal trace The quality of the predicted OUs and PUs is dependent on the original trace. Techniques such as DefUse [21] is effective in obtaining the crucial definition-use relations by statistical extracting the data flows. Concerning the future work, we plan to integrate BIPED with this school of techniques to tackle the trace sensitivity issue and to improve the bug detection precise of BIPED.

Redundant exploration BIPED currently has too many buggy schedules to expose order violations. It can expose soundly the PU and OU bugs in an execution, there exist a lot of the same bugs that correspond to the same statement. This limitation can be solved by removing the equivalent events in the trace before applying BIPED, such as filtering the equivalent events to only keep key events.

6. Related Work

6.1 Order-Related Bug Detection

Recently, Y. Shi et al. proposed a method to detect order violation bugs by using definition-use invariants collected during correct runs [21]. Their approach extracts the set of correct definitions associated with the concerned read during the training of program execution, and captures only those interleavings violated the invariants at runtime. It needs to re-exercise the executions for exposing the bugs. J. Yu and S. Narayanasamy proposed an innovative method to avoid concurrency bugs by using data dependency information collected during correct runs [20]. Their approach encodes the set of tested correct interleavings in a program's binary executable, and enforces only those interleavings at runtime. They use Predecessor Set (PSet) constraints to capture the tested interleavings between two dependent memory instructions. Specifically, for each shared memory instruction, PSet specifies the set of all valid remote memory instructions that can be immediately dependent upon.

6.2 Predictive Trace Analysis

C. Wang et al. developed a symbolic analysis model: Universal Causality Graph, for finding concurrency errors, such as data races, atomicity violations [25], [33]–[35]. The model encodes the causal dependencies among events, the program control structure, and the property of concurrency errors in an uniform way of symbolic constraints and uses a satisfiability solver to verify the existence of property violations. This approach can statically check whether a property holds in all possible permutations of events in the given execution trace. J. Huang et al. [27] proposed a persuasive prediction of concurrency access anomalies. Their approach encodes the order constraint and the temporal order information in a partial and temporal order graph to search the data races, atomicity violation, and atomic-set serializability patterns and check whether a pattern can be implemented in a possible permutation of the given execution trace. However, order violation bugs are special, it needs to more effective ways to expose them.

6.3 Active Testing

An active randomized testing technique [32] that also uses the trace information to detect and create data races [7], dead locks [3], atomic-set serializability violations [36] through checking hybrid constraints collected during a random run. Their approach encodes the set of predicted specific thread schedules and tries to control these specific schedules actively at runtime. Its capability of exposing real concurrency bugs is subjected to the randomness, instead of deterministically exposing every real concurrency bug.

7. Conclusion

In this paper we have introduced BIPED which is a bidirectional predictive trace analysis approach to exposing the order violations in a recorded trace. BIPED performs a bidirectional prediction of the possible OUIs and PUIs by constructing a layered prediction model with encoding the order constraints bidirectionally, and adopting dual predictors to perform a quick check of the candidate OUIs and PUIs. We have designed the safe tactics to support the bidirectional predictive analysis via recognizing candidate OUIs and PUIs which enclose the corresponding Use-Predef and Use-Redef data flows. We have also developed a BIPED tool and conducted the experiments to compare PECAN and CALLFUZZER. The experimental results show that the bidirectional predictive trace analysis approach of BIPED achieves higher efficiency than a classical predictive dynamic analysis approach CALLFUZZER; and higher performance than both a recent predictive trace approach PECAN and CALLFUZZER does in terms of detecting order violations.

In the future, we would like to extend BIPED to iteratively predict the buggy interleaves to expose high-risk bugs, because BIPED can predict all possible OUIs and PUIs in a recorded execution that expose numerous order violations during re-executing, while a majority of them are benign.

Acknowledgments

We are grateful to the anonymous reviewers for their suggestions to improve the paper and to Prof. Xiaoguang Mao and Dr. Jianxin Xue for their insightful discussions. This work was supported in part by the National Basic Research Program of China (973) (Grant No. 2015CB352203), the National Natural Science Foundation of China (NSFC) (Grant No. 91118004, 61272102, 61472240) and the Funding Scheme for Training Young Teachers in Shanghai Colleges (No.ZZegd14001).

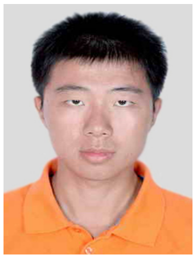
References

[1] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T.E. Anderson, "Eraser: A dynamic data race detector for multithreaded programs," ACM Trans. Comput. Syst., vol.15, no.4, pp.391–411, 1997.
[2] M. Naik, C.-S. Park, K. Sen, and D. Gay, "Effective static deadlock detection," ICSE, 2009, pp.386–396, 2009.
[3] P. Joshi, C.-S. Park, K. Sen, and M. Naik, "A randomized dynamic program analysis technique for detecting real deadlocks," PLDI, 2009, pp.110–120, 2009.
[4] P. Joshi, M. Naik, K. Sen, and D. Gay, "An effective dynamic analysis for detecting generalized deadlocks," SIGSOFT FSE, 2010, pp.327–336, 2010.
[5] C. Flanagan and S.N. Freund, "Atomizer: A dynamic atomicity checker for multithreaded programs (summary)," IPDPS, 2004, 2004.

[6] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: A comprehensive study on real world concurrency bug characteristics," ASPLOS, 2008, pp.329–339, 2008.
[7] K. Sen, "Race directed random testing of concurrent programs," PLDI, 2008, pp.11–21, 2008.
[8] M. Naik, A. Aiken, and J. Whaley, "Effective static race detection for java," PLDI, 2006, pp.308–319, 2006.
[9] E. Bodden and K. Havelund, "Racer: effective race detection using aspectj," ISSTA, 2008, pp.155–166, 2008.
[10] D. Marino, M. Musuvathi, and S. Narayanasamy, "Literace: effective sampling for lightweight data-race detection," PLDI, 2009, pp.134–143, 2009.
[11] C. Flanagan and S. Freund, "Fasttrack: efficient and precise dynamic race detection," PLDI, 2009, pp.121–133, 2009.
[12] M.D. Bond, K.E. Coons, and K.S. McKinley, "Pacer: proportional detection of data races," PLDI, 2010, pp.255–268, 2010.
[13] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder, "Automatically classifying benign and harmful data racesallusing replay analysis," PLDI, 2007, pp.22–31, 2007.
[14] C.-S. Park and K. Sen, "Randomized active atomicity violation detection in concurrent programs," SIGSOFT FSE, 2008, pp.135–145, 2008.
[15] C. Flanagan, S.N. Freund, and J. Yi, "Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs," PLDI, 2008, pp.293–303, 2008.
[16] F. Sorrentino, A. Farzan, and P. Madhusudan, "Penelope: weaving threads to expose atomicity violations," SIGSOFT FSE, 2010, pp.37–46, 2010.
[17] S. Park, S. Lu, and Y. Zhou, "Ctrigger: exposing atomicity violation bugs from their hiding places," ASPLOS, 2009, pp.25–36, 2009.
[18] S. Lu, S. Park, and Y. Zhou, "Finding atomicity-violation bugs through unserializable interleaving testing," IEEE Trans. Softw. Eng., vol.38, no.4, pp.844–860, 2012.
[19] B. Lucia, L. Ceze, and K. Strauss, "Colorsafe: architectural support for debugging and dynamically avoiding multi-variable atomicity violations," ISCA, 2010, pp.222–233, 2010.
[20] J. Yu and S. Narayanasamy, "A case for an interleaving constrained shared-memory multi-processor," ISCA, 2009, pp.325–336, 2009.
[21] Y. Shi, S. Park, Z. Yin, S. Lu, Y. Zhou, W. Chen, and W. Zheng, "Do i use the wrong definition?: Defuse: definition-use invariants for detecting concurrency and sequential bugs," OOPSLA, 2010, pp.160–174, 2010.
[22] K. Sen, G. Rosu, and G. Agha, "Detecting errors in multithreaded programs by generalized predictive analysis of executions," FMOODS, 2005, pp.211–226, 2005.
[23] L. Wang and S. Stoller, "Accurate and efficient runtime detection of atomicity errors in concurrent programs," in PPOPP, 2006, pp.137–146, 2006.
[24] A. Farzan, P. Madhusudan, and F. Sorrentino, "Meta-analysis for atomicity violations under nested locking," CAV, 2009, pp.248–262, 2009.
[25] C. Wang, S. Kundu, M.K. Ganai, and A. Gupta, "Symbolic predictive analysis for concurrent programs," FM, 2009, pp.256–272, 2009.
[26] W. Zhang, C. Sun, and S. Lu, "Conmem: detecting severe concurrency bugs through an effect-oriented approach," ASPLOS, 2010, pp.179–192, 2010.
[27] J. Huang and C. Zhang, "Persuasive prediction of concurrency access anomalies," ISSTA, 2011, pp.144–154, 2011.
[28] R. O'Callahan and J.-D. Choi, "Hybrid dynamic data race detection," PPOPP, 2003, pp.167–178, 2003.
[29] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," Commun. ACM, vol.21, no.7, pp.558–565, 1978.
[30] "Soot-a java optimization framework," Proc. 1999 Conference of the Centre for Advanced Studies on Collaborative Research, pp.125–135, 1999.
[31] E. Farchi, Y. Nir, and S. Ur, "Concurrent bug patterns and how to test them," IPDPS, 2003, p.286, 2003.
[32] P. Joshi, M. Naik, C.-S. Park, and K. Sen, "Calfuzzer: An extensible active testing framework for concurrent programs," CAV, 2009, pp.675–681, 2009.
[33] C. Wang, M. Said, and A. Gupta, "Coverage guided systematic concurrency testing," ICSE, 2011, pp.221–230, 2011.
[34] N. Sinha and C. Wang, "Staged concurrent program analysis," SIGSOFT FSE, 2010, pp.47–56, 2010.
[35] V. Kahlon and C. Wang, "Universal causality graphs: A precise happens-before model for detecting bugs in concurrent programs," CAV, 2010, pp.434–449, 2010.
[36] Z. Lai, S.-C. Cheung, and W.K. Chan, "Detecting atomic-set serializability violations in multithreaded programs through active randomized testing," ICSE (1), 2010, pp.235–244, 2010.

Xi Chang is currently a Ph.D. candidate in Department of Computer Science and Engineering, Shanghai Jiao Tong University, 200240, Shanghai, China. Her research interests include program analysis for concurrent program testing.

Zhuo Zhang is currently a Master in computer science and technology at School of Computer, National University of Defense Technology, 410073, Changsha, China. His research interests include software debugging and program slicing.



Yan Lei is currently a Ph.D. candidate in computer science and technology at School of Computer, National University of Defense Technology, 410073, Changsha, China. His research interests include software debugging and program slicing.

Jianjun Zhao received a Ph.D. degree in Computer Science from Kyushu University (Japan) in 1997. After receiving his Ph.D., he joined the Department of Computer Science and Engineering, Fukuoka Institute of Technology (Japan) as an Assistant Professor, and then was promoted to be an Associate Professor in 2000. Since December 2005, he has been a Full Professor in Department of Computer Science and Engineering, Shanghai Jiao Tong University, 200240, Shanghai, China. His research interests include program analysis for software engineering and compiler optimization.