Hindawi
Security and Communication Networks
Volume 2018, Article ID 6241921, 11 pages
https://doi.org/10.1155/2018/6241921

Research Article

A Constraint-Aware Optimization Method for Concurrency Bug Diagnosis Service in a Distributed Cloud Environment

Lili Bo [1,2] **and Shujuan Jiang** [1,2]

[1] *School of Computer Science and Technology, China University of Mining and Technology, Xuzhou 221116, China*
[2] *Engineering Research Center of Mine Digitalization of Ministry of Education, Xuzhou 221116, China*

Correspondence should be addressed to Shujuan Jiang; shjjiang@cumt.edu.cn

Received 23 August 2018; Accepted 23 September 2018; Published 9 October 2018

Guest Editor: Xuyun Zhang

The advent of cloud computation and big data applications has enabled data access concurrency to be prevalent in the distributed cloud environment. In the meantime, security issue becomes a critical problem for researchers to consider. Concurrency bug diagnosis service is to analyze concurrent software and then reason about concurrency bugs in them. However, frequent context switches in concurrent program execution traces will inevitably impact the service performance. To optimize the service performance, this paper presents a static constraint-aware method to simplify concurrent program buggy traces. First, taking the original buggy trace as the operation object, we calculate the maximal sound dependence relations based on the constraint models. Then, we iteratively check the dependent constraints and move forward current event to extend thread execution intervals. Finally, we obtain the simplified trace that is equivalent to the original buggy trace. To evaluate our approach, we conduct a set of experiments on 12 widely used Java projects. Experimental results show that our approach outperforms other state-of-the-art approaches in terms of execution time.

1. Introduction

Cloud computing organizes and integrates different computing resources (including software and hardware), providing end-users with different services in remote location over the Internet. Testing-as-a-Service (TaaS) based on cloud platform provides automated software testing services, saving capacity and reducing expense [1, 2]. With the increasing popularity of service computing, a vast amount of services-related business applications has emerged, such as service composition [3, 4], service recommendation [5–8], service evaluation [9–11], and service optimization [12–15]. As an important guarantee to the QoS (Quality of Service), such as test effectiveness and efficiency, service optimization has attracted much attention of researchers in software engineering.

Prevalent multicore architecture and big data applications today accelerate the development of concurrent systems [16, 17]. To fully utilize multicore CPUs, multiple execution flows can run simultaneously, i.e., data access concurrency. However, that is more likely to suffer from concurrency bugs, which can pose a great threat to the security and privacy in cloud [18–24]. Furthermore, more scalable and efficient anomaly detection and intrusion detection techniques are needed in big data applications [25–28].

Previous studies have proposed a lot of approaches to expose and detect all kinds of concurrency bugs [29], such as deadlocks [30, 31], data races [32, 33], atomicity violations [34, 35], and order violations [36]. Also, the studies have obtained many excellent results. In addition, a variety of record-replay systems are implemented to replay concurrency bugs effectively [37–39]. However, few researches focus on concurrency bug diagnosis. Concurrency bug is difficult to diagnose as frequent context switches hinder developers to understand concurrent program execution traces. In a concurrent program execution trace, most context switches are the fine-grained thread interleaves which are conflicted on accessing the shared memory. The order of accessing the shared memory for two threads forms a dependence relation. The more dependence relations there are, the more difficult it is to reason about concurrency bugs. Additionally, a series of

operations will happen when CPU executes a context switch, including preserving the current site and loading the next site. These operations obviously bring tremendous performance consumption.

Therefore, it is necessary to introduce an optimization technique that can reduce the shared memory dependences and increase the granularity of thread interleaving with the promise of replaying the same bugs. In this paper, we present a static constraint-aware approach to optimize the process of concurrency bug diagnosis. We analyze the original buggy trace offline and simplify it automatically to get a new equivalent trace with less context switches. Our experiments are conducted on 12 widely used Java concurrent benchmarks. Experimental results show that our approach performs better than or is comparable to the compared method (i.e., SimTrace [40]) in reduction as well as performance.

In summary, the main contributions of this paper are listed as follows:

(1) We present a static constraint-aware optimization method-CAOM for obtaining simplified traces which are equivalent to the original buggy traces.

(2) We demonstrate the effectiveness and efficiency of our approach with extensive evaluation on a suit of popular multithreaded projects.

The remainder of this paper is organized as follows. Section 2 summarizes the related work. Section 3 describes the problem formulation and the research motivation. Section 4 presents our constraint-aware optimization method for concurrency bug diagnosis service. In Section 5, we conduct an empirical study to show its validity and finally, in Section 6, we conclude the paper.

2. Related Work

How to optimize cloud users' service invocation cost is always a hot research topic in cloud computing. With considering the service's past invocation costs, FL-FL method was proposed in [41] to evaluate and predict a cloud user's service invocation cost. Unfortunately, it cannot generate an accurate service invocation cost. Work in [42] presented a cost-benefit-aware cloud service scheduling method Cost-plus, but it failed to minimize the service invocation cost. Some researchers focus their work on minimizing the service invocation cost. For example, Li et al. [43] proposed FCFS, which utilized the role of "Fist Come Fist Serve" to reduce the waiting time of user job to optimize the service invocation cost. However, these methods neglected many important factors, such as user job size. Recently, CS-COM was put forward in [12] with considering multiple factors, which significantly optimized the service invocation cost.

In addition to the service invocation cost optimization, researchers also put forward many methods to optimize the performance of other services, such as concurrency bug diagnosis service. Concurrency bug diagnosis attempts to reason about concurrency bugs in buggy traces. An effect optimization approach for improving the performance of concurrency bug diagnosis is to simplify the buggy trace. Trace simplification techniques can be divided into online analysis and offline analysis.

Online trace simplification technique uses vector clock or lock assignment and then groups variables with transitive reduction and thread/spatial locality. The author in [44] proposed to record only the conflicting shared memory dependences with transitive reduction, reducing the time and space overhead of recording. To further reduce the record overhead, Xu et al. proposed FDR (Flight Data Recorder) [45] and RTR (Regulated Transitive Reduction) [46], which record the strict vector dependences based on hardware. In [47], an execution reduction system was developed combined with checkpoint, which removes the events irrelevant to errors. Recently, a software-only algorithm (bisectional coordination protocol) was presented in [48] to reduce the shared memory dependences. Experimental results indicated that the software-only approach was effective and efficient in trace simplification.

Offline trace simplification technique first obtains a complete buggy trace and then simplifies it offline. SimTrace [40] is a classical offline trace simplification technique, but it consumed too much time on constructing the dependence graph and random selection. Jalbert and Sen were the first to present a heuristic dynamic trace simplification approach, Tinertia [49]. To reduce the context switches in the buggy trace, they performed three operations (i.e., Remove Last and Two-Stage Consolidate Up and Consolidate Down) iteratively and constantly replayed the middle trace to validate the equivalence, which increased the runtime overhead seriously. To speed up replay, the authors in [50] simplified the process of replaying concurrency bugs using replay-supported execution reduction. However, multiple replay verification reduced the simplification performance.

In view of the limitations of the existing approaches, we propose a new constraint-aware static trace simplification approach to optimize concurrency bug diagnosis service, as elaborated in the next section.

3. Problem Formulation and Motivation

In this section, we first formulate the problem of trace simplification for concurrent programs. Then, we present an example to motivate our research.

3.1. Problem Formulation. Trace simplification technique attempts to obtain a simplified trace with less context switches yet still equivalent to the original buggy trace. Next, we give the relevant definitions in detail.

(1) Event. A minimum execution unit in that cannot be interrupted in a concurrent program execution. If this event is an access to the shared variables, it is a global event. Otherwise, it is a local event.

(2) Trace. A trace, denoted by $tr = <e_1, e_2, \ldots e_i, \ldots>$, is an event sequence of a program execution.

(3) Context Switch (CS). In a trace, a context switch occurs when two consecutive events are performed by two different threads.

```
//Thread 1                    //Thread 2
If(count==0){
   lis. add(name);            print(count);
   count++;                   if (count == 1){
   print(count);                 lis.clear();
   lis.get(count);            }
}
```

Figure 1: An example program.

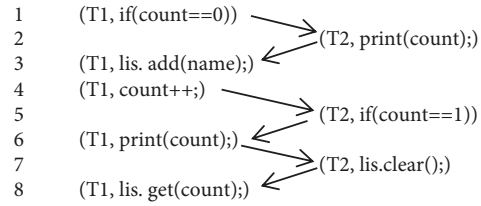


Figure 2: A buggy trace of the example program (CS=6).

(4) Dependence Relation. A dependence relation is the minimum transitive closure over the events in the trace, denoted as $e_i \longrightarrow e_j$.

The dependence relation can be divided into local dependence and remote dependence according to the fact that whether two events occur in different threads. If e_i immediately precedes e_j in the same thread, e_i and e_j are in local dependence relation. Otherwise, they are in remote dependence relation.

Note that two events belonging to remote dependence relation must access the same shared variable. Therefore, the remote dependence relation can be further classified into conflict order and synchronization order according to the types of events. Conflict order contains *read⟶write*, *write⟶read*, and *write⟶write*. Synchronization order consists of *unlock⟶lock*, *fork⟶start*, *exit⟶join*, and *notify⟶wait*.

(5) Equivalent Trace. The original trace is equivalent to the simplified trace if and only if they arrive at the same final state from the same initial program state. The simplified trace is called the equivalent trace of the original trace.

(6) Thread Execution Interval (TEI). The largest set of consecutive events in a thread is a thread execution interval. As we can see, the relationship between the number of context switches and the number of threads execution interval is

$$|TEI| = |CS| + 1 \quad (1)$$

$|TEI|$ and $|CS|$ represent the number of *TEI* and the number of *CS* in the trace, respectively. The goal of trace simplification problem is to make $|CS|$ as small as possible, that is, to make *TEI* as large as possible. Therefore, in the process of trace simplification, under the premise of ensuring trace equivalent, we can put together as many adjacent events in a thread as possible.

3.2. Research Motivation. We use the example in Figure 1 to illustrate trace simplification problem. Assume that the variables *count* and *lis* are initialized to zero and null, respectively. There are two threads accessing the shared variables *count* and *lis* concurrently under the sequence consistency memory model (*SC*). All statements are executed atomically. A null pointer exception will happen in the case that Thread2 executing "list.clear()" occurs between Thread1 executing "count++" and "list.get(count)". In fact, this is a concurrency bug. Figure 2 shows an execution sequence obtained after running the example program. Like [40], we call this sequence a buggy trace. In Figure 2, there are six context switches.

When developers debug concurrent programs, they may run them many times. Each time the program gets error, such as crash, hang, or inconsistent results, developers have to reason about the concurrency bugs along with frequent context switches. That undoubtedly consumes too much time and energy. Trace simplification technique can alleviate this problem effectively. However, two challenges arise in trace simplification: (1) the program semantics are easy to be changed by mistakes and (2) the efficiency of simplification is reduced tremendously because of too many instruments and dynamic verification.

In view of these challenges, we propose a new static constraint-aware approach to simplify concurrent program execution traces and optimize the process of concurrency bug diagnosis. The detailed description of our approach will be given in the next section.

4. A Constraint-Aware Optimization Method for Concurrency Bug Diagnosis Service

In this section, a constraint-aware trace simplification method is proposed to optimize the performance of concurrency bug diagnosis. We first briefly describe the overview of our method. Then, we present the algorithm and corresponding explanation.

4.1. Overview. The overview of our method is described in Figure 3. It mainly consists of three steps. The first step is

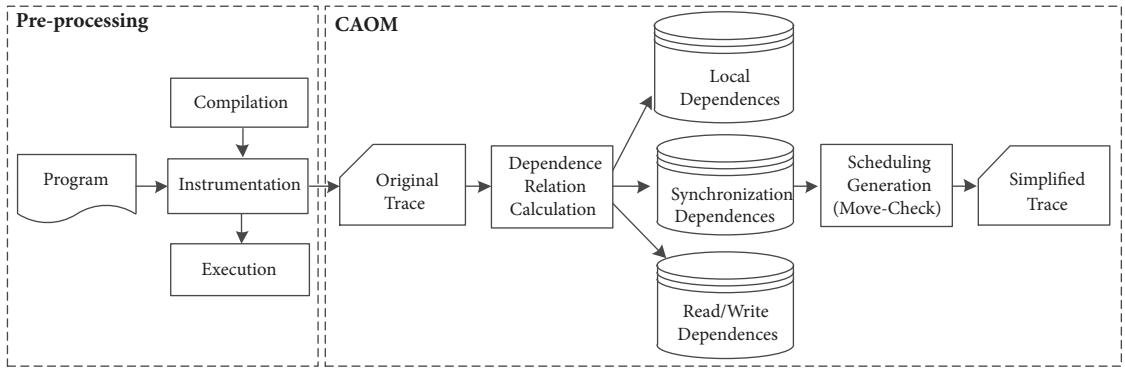


Figure 3: Overview of CAOM.

preprocessing. Test program is compiled to be the corresponding bytecode. After this bytecode is instrumented by Soot [51], we run the instrumented test program and collect the buggy execution trace. In the second step, we calculate local dependences, synchronization dependences, and read/write dependences. The final step is scheduling generation. We take the original trace as input and iteratively check the move forward condition for each event with the constraint dependence relations. If the condition is satisfied, the event is moved forward to extend the thread execution externally. Finally, we obtain the simplified trace.

Note that, in the process of preprocessing, we use the existing instrumentation and record tools to collect original traces. Therefore, we mainly focus on the last two steps: dependence relation calculation and scheduling generation.

4.2. Dependence Relation Calculation. The root cause of nondetermination for thread scheduling is the shared memory access. This leads to the fact that multiple threads can access the same shared variable simultaneously. Therefore, it is a precondition for trace simplification to accurately identify the dependence relations between events in the original buggy trace.

Calculating local dependences only needs to divide the events into several sequences in order according to the threads they belong to. The number of sequences is equal to that of threads. Synchronization dependences can be obtained during collecting the original trace. Then, in this step, we focus on calculating remote read/write dependences. In order to get accurate remote read/write dependence relations, we successively deal with two adjacent accesses on the same shared variable to ensure that the value by read access is always written by the latest write access.

First, we traverse the original trace and divide events into different lists according to the shared variables they access. Then, for the access sequence of every shared variable, we check two successive events in sequence from the first event. If the current two events belong to different threads, they form a remote dependence relation. Furthermore, if two events are both write accesses, they form a remote write-write dependence. If a read event precedes a write event, they form a remote read-write dependence. If a write event precedes a read event, they form a remote write-read dependence.

4.3. Scheduling Generation. Scheduling generation attempts to reduce the number of context switches by two operations (i.e., check and move forward) without breaking all the dependence relations in original traces. A natural thought is to employ a constraint solver that solves three constraints (i.e., write-write dependences, read-write dependences, and write-read dependences). Although the obtained trace satisfies dependence relations in the original trace, the number of context switches may not be reduced. Therefore, we directly take the original trace as the operate object. We check and move forward the atomic events in sequence. Checking is to maintain constraints. Moving is to extend thread execution interval, reducing as many context switches as possible. The detailed process of scheduling generation is shown in Algorithm 1.

Algorithm 1 takes the original trace and dependence relations as input and takes the simplified scheduling sequence as output. Algorithm begins from the second event. If the current event has no dependent event (synchronization dependences or remote read/write dependences), it is moved forward to the location behind the latest event which belongs to the same thread (lines (9)-(10)). If the dependent event is before the latest event which belongs to the same thread with current event, the current event is moved forward to the location behind the latest event (lines (14)-(15)). Otherwise, the events are not moved.

According to Theorem 1 in [40], we know that any rescheduling of events in a trace respecting the dependence relation generates an equivalent trace. In our method, all the feasible events were moved without breaking the dependence constraints. First, we check the dependence relations of the current event. Then, we move it forward under the constraint conditions. That is, all the dependence relations of every event in the new scheduling are the same as that in the original trace. Therefore, the generated trace simplified by our method is equivalent to the original trace.

5. Experiments

5.1. Experimental Configurations. In this section, we conducted experiments on 12 widely used Java multithreaded programs. The details are listed in Table 1. For every program, its lines of code (LOC), number of threads (#Thread),

```
(1)  input: δ - the original trace
(2)         deps - map of sv to its dependence relations
(3)  output: scheduleSeq - sequence of thread schedule
(4)  begin
(5)    for i ⟵ 1 to |δ| − 1 do
(6)      t_δ[i] ⟵the thread identifier of the event δ[i];
(7)      e_latest ⟵the latest event whose thread identifier is t_δ[i];
(8)      loc_e_latest ⟵the location of e_latest in the new trace;
(9)        if δ[i].getDep() = null then
(10)             insert δ[i] immediately after loc_e_latest;
(11)       else
(12)             e_dep ⟵ δ[i].getDep();
(13)             loc_e_dep ⟵the location of e_dep in the new trace;
(14)             if loc_e_dep < loc_e_latest then
(15)                 insert δ[i] immediately after loc_e_latest;
(16)             end if
(17)       end if
(18)   end for
(19) end
```

Algorithm 1: GenScheduling (δ, *svs*).

number of shared variables (#SV), and the origin (Origin) are summarized. It involves large-scale (LOC>10,000), middle-scale (10,000>LOC>1000), and small programs (LOC<1000). The program scales in terms of LOC vary from 73 for Critical to 17,596 for SpecJBB-2005. The number of shared variables is obtained using escape analysis [52]. Specifically, the number of threads or shared variables is not integer by accident. The reasons are that (1) the results were averaged over 50 runs for each program and (2) different paths may be chosen during program execution due to the natural character of dynamic analysis and the dynamic thread creation of Java. In addition, each subject has at least a concurrency bug. For example, Critical has 16 data races and 14 atomicity violations.

To evaluate the effectiveness and efficiency of our approach, we compared it with the state-of-the-art approach named SimTrace. Concretely, we designed three groups of experiments to validate the following three questions:

(1) Effectiveness: how many context switches can be reduced in trace simplification for CAOM?

(2) Efficiency: how much time does it consume in trace simplification for CAOM?

(3) Comparison: does CAOM perform better than SimTrace?

The experiments are conducted on a Samsung notebook running 64-bit Ubuntu-14.04 and jdk1.7 with 3.06 GHz Intel Core 4 processor and 4 GB memory. We utilize Soot to instrument bytecode programs. To collect original traces and replay concurrency bugs, we employ the existing record-replay tool LEAP [54]. We first use random testing to generate an original buggy trace for each subject. All the results are averaged over running 50 times.

5.2. Experimental Results and Analysis. Experimental evaluation is conducted in terms of effectiveness, efficiency, and comparison to answer the above three questions, respectively.

Profile 1 (Effectiveness). The effectiveness of trace simplification technique can be shown by the reduction of context switches. For better understanding, CAOM preserves all the program information; that is, we do not conduct any delete operations to subjects.

Table 2 lists the number of threads (#Thread), the length of original trace (Size), the number of context switches in the original buggy trace (#CSori), the number of context switches in the simplified trace (#CSsim), and the reduction (Reduction(%)), where the length of trace is the total number of synchronization operations and memory accesses. As CAOM does not conduct any delete operations for subjects, the length of trace stays the same before and after simplification. However, we can see that context switches are reduced obviously. The context switches in the simplified trace are reduced by 27.36%~99.97% (54.39% averaged) compared to that of the original buggy trace. Specifically, for the large-scaled subject SpecJBB-2005, the context switches are reduced from 124200.3 to 37.6, and the reduction is 99.97%. Besides, we can find that the more threads and more synchronization operations or memory accesses there are in the original trace, the higher the reduction we can get, such as Manager, Tsp, Cache4j, and SpecJBB-2005.

Profile 2 (Efficiency). The efficiency of trace simplification technique can be shown by the time consumption. This can affect whether it can be applied in practice. The time consumption of CAOM consists of three perspectives: data loading, dependence relations calculation, and scheduling generation. As CAOM is an offline approach and the original

Table 1: Experimental subjects.

Program	LOC	#Thread	#SV	Origin
Critical	73	4.3	1.4	IBM ConTest benchmark suit [53]
Account	148	3.0	4.0	IBM ConTest benchmark suit
Loader	148	4.0	2.0	IBM ConTest benchmark suit
Manager	212	4.4	3.0	IBM ConTest benchmark suit
BuggyProgram	385	4.0	5.0	IBM ConTest benchmark suit
ReadersWriters	103	4.0	4.1	SIR (http://sir.unl.edu/content/sir.php.)
Tsp	709	5.0	12.0	SIR
StringBuffer	1320	3.0	5.0	Suns JDK 1.4.2
LinkedList	5979	3.0	15.0	Suns JDK 1.4.2
ArrayList	5866	3.0	5.0	Suns JDK 1.4.2
Cache4j	3897	4.0	5.0	[40]
SpecJBB-2005	17,596	4.0	116.0	SPEC's benchmark (http://www.spec.org/web2005.)

Table 2: Experimental results I: effectiveness.

Program	#Thread	Size	#CSori	#CSsim	Reduction(%)
Critical	4.3	40.1	6.1	4.5	27.36
Account	3.0	73.0	8.1	4.9	39.31
Loader	4.0	64.0	4.2	3.0	27.88
Manager	4.4	1.4 K	110.1	11.9	**89.19**
BuggyProgram	4.0	228.5	6.5	4.2	36.00
ReadersWriters	4.0	327.3	7.4	3.4	54.59
Tsp	5.0	1001 K	24.0	5.5	**76.92**
StringBuffer	3.0	97.0	3.2	2.0	37.50
LinkedList	3.0	427.2	3.8	2.0	46.81
ArrayList	3.0	334.0	3.0	2.0	33.33
Cache4j	4.0	1190 K	140.0	22.7	**83.78**
SpecJBB-2005	4.0	1148 K	124 K	37.6	**99.97**

buggy trace is collected using instrument and record in the preprocessing step, the complete trace information needs to be loaded before starting simplification.

Table 3 lists experimental results in terms of the time consumption. Columns 4-7 represent the time consumed in data loading, dependence calculation, scheduling generation, and the total time, respectively. As we can see, for the 12 Java multithreaded programs, the maximum time consumed is no more than 30 min, which indicates good efficiency of our method. For example, for Tsp whose length of trace is 1001 K, the total simplification time is only 2.7 min.

Concretely, for most middle-scaled and small programs, the time is mainly consumed in data loading. For example, for ArrayList and Loader, the time consumed in data loading accounts for 84.49% and 84.16% of their total time, respectively. However, for Tsp and Cache4j, the time consumed in dependence relation calculation and scheduling generation is far more than that of data loading. The reason is that there are many synchronization operations, memory accesses, and context switches in the original trace, which leads to the fact that the dependence relations are much more complex; then, frequent check and move operations have to be conducted. Specifically, for Cache4j, the time consumed in dependence relation calculation is more than that of scheduling generation. The reason is that there are many lock dependence relations in which two locks are adjacent and belong to the same thread, saving many move operations.

For large-scaled programs, the time consumed in trace simplification increases because of the large trace size and a vast amount of context switches. However, our method still has good efficiency as it does not conduct multiple iterations and replay validation. For example, for SpecJBB-2005, the time consumption for the whole simplification is less than 30 min.

Profile 3 (Comparison Analysis). To evaluate that our approach performs better than the state-of-the-art approaches, we compared CAOM with SimTrace. Both SimTrace and CAOM reduce trace simplification problem to the combinatorial optimization problem. The difference is that SimTrace takes it as an optimization problem with graph merging.

Comparison results between SimTrace and CAOM on the reduction of context switches are presented in Figure 4. Figure 4 shows that, for most programs, CAOM can reduce more context switches compared with SimTrace. For example, for BuggyProgram, CAOM increases the reduction by 14.15%.

Table 3: Experimental results II: efficiency.

Program	LOC	Size	Data loading(ms)	Dependence calculation(ms)	Scheduling generation(ms)	Total time(ms)
Critical	73	40.1	37.6	2.3	2.7	45.7
Account	148	73.0	30.9	1.7	2.3	37.5
Loader	148	64.0	25.5	1.4	1.7	30.3
Manager	212	1.4 K	46.6	6.5	20.1	75.2
BuggyProgram	385	228.5	26.9	1.5	3.1	33.4
ReadersWriters	103	327.3	36.6	2.1	3.9	44.9
Tsp	709	1001 K	6.0E+03	7.1E+04	8.6E+04	1.63E+05
StringBuffer	1320	97.0	26.4	1.5	1.7	31.5
LinkedList	5979	427.2	32.6	1.9	3.0	39.1
ArrayList	5866	334.0	30.5	1.7	2.1	36.1
Cache4j	3897	1190 K	6.0E+03	2.39E+05	2.9E+04	2.74E+05
SpecJBB-2005	17,596	1148 K	8.0E+03	8.1E+04	9.31E+05	1.020E+06 (<30 min)

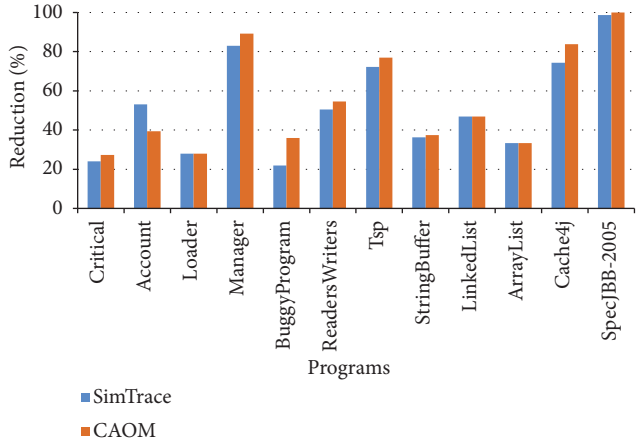
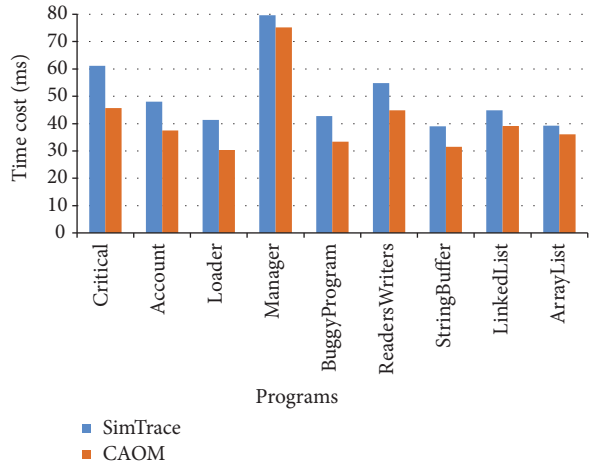
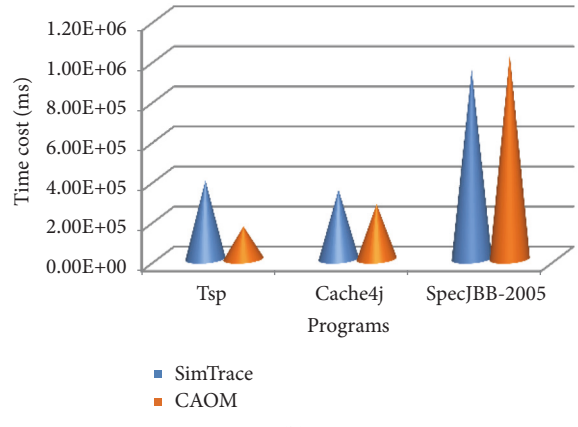


Figure 4: Comparison results between SimTrace and CAOM on reduction of CSs.



(a)



(b)

Figure 5: Comparison results between SimTrace and CAOM on time cost.

However, there is an exception to the overall result. For Account, our approach reduces the reduction by 13.83%. The reason is that SimTrace mainly pursuits performance improvement, which leads to a risk of replay failure. This situation cannot exist in our approach as we calculate the strict dependence relations between two successive events and simplify the original buggy trace in the promise of replay.

Comparison results between SimTrace and CAOM on time cost are shown in Figure 5. We separate three programs (shown in Figure 5(b)) from 12 subjects to show their time

cost as they cost too much time in the process of trace simplification. Figure 5 shows, for all the middle-scaled and small programs, CAOM is superior to SimTrace in terms of efficiency, because our method does not need to construct any abstract models. For example, for Tsp, CAOM improves the performance by 58.60%. But for SpecJBB-2005, CAOM is inferior to SimTrace. The reason is that the context switches are significantly reduced by our method, which implies that it suffers from frequent check and move operations, consuming too much time.

To sum up, we can conclude the following points based on the above experiments, which can also answer three questions in Section 5.1.

(1) CAOM is effective in trace simplification. It can reduce context switches by 27.36%~99.97% (54.39% on average).

(2) CAOM is efficient in trace simplification. Even for the large-scaled programs, it can finish the simplification within 30 min.

(3) CAOM performs better than SimTrace on both reduction of context switches and time cost.

5.3. Time Complexity Analysis. The time complexity of CAOM is $O(n^2)$, where n represents the number of events in the original trace. Given the original buggy trace, we first calculate the dependence relations for each event, whose time complexity is $O(n)$. Then, in scheduling generation, for each event, we need to search for the location of its dependence node and the latest node in the same thread, whose time complexity is $O(n^2)$. Dependence relation calculation and scheduling generation are executed in sequence. Therefore, with the above analysis, we can conclude that the time complexity of our method is $O(n^2)$.

5.4. Discussion. Based on our experiments, we can find that CAOM can simplify concurrent program execution traces with a length of million magnitudes effectively and efficiently, which is helpful to be applied in practice. Specifically, in theory, our method can provide enlightenment for the new optimization algorithms design about concurrency bug diagnosis service. In practice, experimental results show that CAOM can use less time but reduce more context switches, which implies that developers can save much time on debugging concurrent programs. Thus, our proposed method can speed up the concurrent software development.

However, there are still a few directions that may further improve our method. First, for efficiency, we only considered a one-way checking and moving. A bidirectional or a two-stage simplification [55] may further improve the effectiveness and reduce more context switches. Second, in the step of preprocessing, we used escape analysis to identify the shared variables. Both Static-TSA and Dynamic-TSA [56] are more precise and efficient than escape analysis. Moreover, they are scalable to real-world large multithreaded applications. Next, we will employ Static-TSA or Dynamic-TSA to improve the availability of our method. Third, for completeness, we calculated the dependence relations of all the events, which consumed much redundant time. We plan to utilize programming slicing [57] or Collaborative Filtering [58, 59] to extract the critical variables and the corresponding events.

6. Conclusions

The advance of cloud computation and big data applications accelerates current software development and produces various concurrency cloud services in the distributed cloud environment. However, it is a great challenge to guarantee both service quality and service performance. Concurrency bug diagnosis service is to reason about vulnerabilities in concurrent applications. The existing trace simplification techniques are either online analysis or based on the complex graph structures, which limits the performance of service optimization. In this paper, we present a novel static constraint-aware optimization method for concurrency bug diagnosis service in the distributed cloud environment. We obtain a simplified trace by iteratively checking dependence constraints and moving forward feasible events if the condition is satisfied. With the constraint-aware idea, we can guarantee that the simplified trace is equivalent to the original buggy trace. Furthermore, the effectiveness and efficiency of trace simplification have been significantly improved as we optimized concurrency bug diagnosis service offline without any complex structures. Finally, through a set of experiments conducted on 12 widely used java projects, we further demonstrate that our proposed CAOM outperforms other state-of-the-art approaches.

Data Availability

The data used to support the findings of this study are available from the corresponding author upon request.

Conflicts of Interest

The authors declare that they have no conflicts of interest regarding the publication of this paper.

Acknowledgments

This paper is partially supported by Natural Science Foundation of China [No. 61673384 and No. 61502497].

References

[1] D. Savchenko, N. Ashikhmin, and G. Radchenko, "Testing-as-a-service approach for cloud applications," in *Proceedings of the IEEE/ACM International Conference on Utility and Cloud Computing (UCC '17)*, pp. 428-429, Shanghai, China, December 2016.

[2] S. Herbold and A. Hoffmann, "Model-based testing as a service," *International Journal on Software Tools for Technology Transfer*, vol. 19, no. 3, pp. 271–279, 2017.

[3] T. Fissaa, H. Guermah, M. E. Hamlaoui, H. Hafiddi, and M. Nassar, "A synergy of semantic and context awareness for service composition in ubiquitous environment," *Computer and Information Science*, vol. 11, no. 2, pp. 88–98, 2018.

[4] L. Barakat, S. Miles, and M. Luck, "Adaptive composition in dynamic service environments," *Future Generation Computer Systems*, vol. 80, pp. 215–228, 2018.

[5] Y. Xu, L. Qi, W. Dou, and J. Yu, "Privacy-preserving and scalable service recommendation based on simhash in a distributed cloud environment," *Complexity*, vol. 2017, Article ID 3437854, 9 pages, 2017.

[6] X. Xia, J. Yu, S. Zhang, and S. Wu, "Trusted service scheduling and optimization strategy design of service recommendation," *Security & Communication Networks*, vol. 2017, Article ID 9192084, 9 pages, 2017.

[7] L. Qi, X. Zhang, W. Dou, and Q. Ni, "A distributed locality-sensitive hashing-based approach for cloud service recommendation from multi-source data," *IEEE Journal on Selected Areas in Communications*, vol. 35, no. 11, pp. 2616–2624, 2017.

[8] W. Gong, L. Qi, and Y. Xu, "Privacy-aware multidimensional mobile service quality prediction and recommendation in distributed fog environment," *Wireless Communications & Mobile Computing*, vol. 2018, Article ID 3075849, 8 pages, 2018.

[9] Z. Ma, R. Jiang, M. Yang, T. Li, and Q. Zhang, "Research on the measurement and evaluation of trusted cloud service," *Soft Computing*, vol. 22, no. 4, pp. 1247–1262, 2018.

[10] H. Alabool, A. Kamil, N. Arshad, and D. Alarabiat, "Cloud service evaluation method-based Multi-Criteria Decision-Making: A systematic literature review," *The Journal of Systems and Software*, vol. 139, pp. 161–188, 2018.

[11] M. Tang, X. Dai, J. Liu, and J. Chen, "Towards a trust evaluation middleware for cloud service selection," *Future Generation Computer Systems*, vol. 74, pp. 302–312, 2017.

[12] L. Qi, J. Yu, and Z. Zhou, "An invocation cost optimization method for web services in cloud environment," *Scientific Programming*, vol. 2017, Article ID 4358536, 9 pages, 2017.

[13] C. Vorhemus and E. Schikuta, "Blackboard meets Dijkstra for optimization of web service workflows," *Computing Research Repository*, 2017, https://arxiv.org/abs/1801.00322.

[14] H. Cui, X. Liu, and T. Yu, "Cloud service scheduling algorithm research and optimization," *Security & Communication Networks*, vol. 2017, Article ID 2503153, 7 pages, 2017.

[15] X. Guo, S. Chen, Y. Zhang, and W. Li, "Service composition optimization method based on parallel particle swarm algorithm on spark," *Security & Communication Networks*, vol. 2017, Article ID 9097616, 8 pages, 2017.

[16] Y. Liu and X. Sun, "CaL: Extending Data Locality to Consider Concurrency for Performance Optimization," *IEEE Transactions on Big Data*, vol. 4, no. 2, pp. 273–288, 2018.

[17] G. Pinto, W. Torres, B. Fernandes, F. Castor, and R. S. M. Barros, "A large-scale study on the usage of Java's concurrent programming constructs," *The Journal of Systems and Software*, vol. 106, pp. 59–81, 2015.

[18] J. Li, Y. K Li, X. Chen, P. P. C. Lee, and W. Lou, "A hybrid cloud approach for secure authorized deduplication," *IEEE Transactions on Parallel Distributed Systems*, vol. 26, no. 5, pp. 1206–1216, 2015.

[19] J. Li, X. Chen, C. Jia, and W. Lou, "Identity-based encryption with outsourced revocation in cloud computing," *IEEE Transactions on Computers*, vol. 64, no. 2, pp. 425–437, 2015.

[20] J. Shen, Z. Gui, S. Ji, J. Shen, H. Tan, and Y. Tang, "Cloud-aided lightweight certificateless authentication protocol with anonymity for wireless body area networks," *Journal of Network and Computer Applications*, vol. 106, no. 15, pp. 117–123, 2018.

[21] L. Yang, Z. Han, Z. Huang, and J. Ma, "A remotely keyed file encryption scheme under mobile cloud computing," *Journal of Network and Computer Applications*, vol. 106, no. 15, pp. 90–99, 2018.

[22] J. Li, Z. Liu, X. Chen, F. Xhafa, X. Tan, and D. S. Wong, "L-EncDB: A lightweight framework for privacy-preserving data queries in cloud computing," *Knowledge-Based Systems*, vol. 79, pp. 18–26, 2015.

[23] Z. Cai, H. Yan, P. Li, Z.-A. Huang, and C. Gao, "Towards secure and flexible EHR sharing in mobile health cloud under static assumptions," *Cluster Computing*, vol. 20, no. 3, pp. 2415–2422, 2017.

[24] L. Qi, S. Meng, X. Zhang et al., "An Exception Handling Approach for Privacy-Preserving Service Recommendation Failure in a Cloud Environment," *Sensors*, vol. 18, no. 7, p. 2037, 2018.

[25] X. Zhang, W. Dou, Q. He et al., "LSHiForest: A Generic Framework for Fast Tree Isolation Based Ensemble Anomaly Analysis," in *Proceedings of the the 33rd IEEE International Conference on Data Engineering (ICDE'17)*, pp. 983–994, San Diego, Calif, USA, April 2017.

[26] K. Peng, V. C. M. Leung, and L. Zheng, "Intrusion detection system based on decision tree over big data in fog environment," *Wireless Communications Mobile Computing*, vol. 2018, Article ID 4680867, 10 pages, 2018.

[27] K. Peng, V. C. M. Leung, and Q. Huang, "Clustering approach based on mini batch kmeans for intrusion detection system over big data," *IEEE Access*, vol. 6, pp. 11897–11906, 2018.

[28] W. Meng, E. W. Tischhauser, Q. Wang, Y. Wang, and J. Han, "When intrusion detection meets blockchain technology: a review," *IEEE Access*, vol. 6, pp. 10179–10188, 2018.

[29] Z. Wu, K. Lu, and X. Wang, "Surveying concurrency bug detectors based on types of detected bugs," *Science China Information Sciences*, vol. 60, no. 3, 2017.

[30] Y. Cai and W. K. Chan, "MagicFuzzer: Scalable deadlock detection for large-scale applications," in *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*, pp. 606–616, Zurich, Switzerland, June 2012.

[31] Y. Cai and Q. Lu, "Dynamic testing for deadlocks via constraints," *IEEE Transactions on Software Engineering*, vol. 42, no. 9, pp. 825–842, 2016.

[32] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: a dynamic data race detector for multithreaded programs," *ACM Transactions on Computer Systems*, vol. 15, no. 4, pp. 391–411, 1997.

[33] P. Kang, "Software analysis techniques for detecting data race," *IEICE Transaction on Information and Systems*, vol. E100-D, no. 11, pp. 2674–2682, 2017.

[34] C. Flanagan and S. N. Freund, "Atomizer: a dynamic atomicity checker for multithreaded programs," *Science of Computer Programming*, vol. 71, no. 2, pp. 89–109, 2008.

[35] Q. Shi, J. Huang, Z. Chen, and B. Xu, "Verifying synchronization for atomicity violation fixing," *IEEE Transactions on Software Engineering*, vol. 42, no. 3, pp. 280–296, 2016.

[36] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 2, pp. 329–339, 2008.

[37] Y. Jiang, T. Gu, C. Xu, X. Ma, and J. Lu, "CARE: cache guided deterministic replay for concurrent Java programs," in *Proceedings of the International Conference on Software Engineering (ICSE '14)*, pp. 457–467, Hyderabad, India, May 2014.

[38] S.-B. Tang, F.-L. Song, S. Zhang, D.-R. Fan, and Z.-Y. Liu, "Reducing log of dependencies based on global synchronous logic clock," *Chinese Journal of Computers*, vol. 37, no. 7, pp. 1487–1499, 2014.

[39] Y. Chen, S. Zhang, Q. Guo, L. Li, R. Wu, and T. Chen, "Deterministic replay: a survey," *ACM Computing Surveys*, vol. 48, no. 2, pp. 1–47, 2015.

[40] J. Huang and C. Zhang, "An efficient static trace simplification technique for debugging concurrent programs," in *Proceedings of the 18th International Conference on Static Analysis (SAS '11)*, pp. 163–179, Venice, Italy, September 2011.

[41] Q. Liu, W. Cai, J. Shen, Z. Fu, X. Liu, and N. Linge, "A speculative approach to spatial-temporal efficiency with multi-objective optimization in a heterogeneous cloud environment," *Security and Communication Networks*, vol. 9, no. 17, pp. 4002–4012, 2016.

[42] W.-H. Choi and K.-S. Kang, "A Study on deciding optimal price of bioinformatics services," *Journal of the Korea Safety Management and Science*, vol. 18, no. 1, pp. 203–208, 2016.

[43] M. Li, D. Subhraveti, A. R. Butt, A. Khasymski, and P. Sarkar, "CAM: a topology aware minimum cost flow based resource manager for MapReduce applications in the cloud," in *Proceedings of the 21st ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC '12)*, pp. 211–222, Delft, The Netherlands, June 2012.

[44] R. H. B. Netzer, "Optimal tracing and replay for debugging shared-memory parallel programs," in *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging (PADD '93)*, pp. 1–11, San Diego, Calif, USA, May 1993.

[45] M. Xu, R. Bodik, and M. Hill, "A "flight data recorder" for enabling full-system multiprocessor deterministic replay," in *Proceedings of the Annual International Symposium on Computer Architecture (ISCA'03)*, pp. 122–133, San Diego, Calif, USA, June 2003.

[46] M. Xu, M. D. Hill, and R. Bodik, "A regulated transitive reduction (RTR) for longer memory race recording," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 5, pp. 49–60, 2006.

[47] S. Tallam, C. Tian, R. Gupta, and X. Zhang, "Enabling tracing of long-running multithreaded programs via dynamic execution reduction," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA '07)*, pp. 207–218, London, UK, July 2007.

[48] Y. Jiang, C. Xu, D. Li, X. Ma, and J. Lu, "Online shared memory dependence reduction via bisectional coordination," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'16)*, pp. 822–832, Seattle, WA, USA, November 2016.

[49] N. Jalbert and K. Sen, "A trace simplification technique for effective debugging of concurrent programs," in *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'10)*, pp. 57–66, Santa Fe, NM, USA, November 2010.

[50] J. Huang and C. Zhang, "LEAN: Simplifying concurrency bug reproduction via replay-supported execution reduction," *ACM SIGPLAN Notices*, vol. 47, no. 10, pp. 451–465, 2012.

[51] "Soot-a Java bytecode optimization framework," https://github.com/Sable/soot.

[52] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff, "Escape analysis for Java," *SIGPLAN Notices (ACM Special Interest Group on Programming Languages)*, vol. 34, no. 10, pp. 1–19, 1999.

[53] E. Farchi, Y. Nir, and S. Ur, "Concurrent bug patterns and how to test them," in *Proceedings of the 17th International Symposium on Parallel and Distributed Processing (IPDPS '03)*, pp. 1–7, Nice, France, April 2003.

[54] J. Huang, P. Liu, and C. Zhang, "LEAP: lightweight deterministic multi-processor replay of concurrent java programs," in *Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '10)*, pp. 385-386, Santa Fe, NM, USA, November 2010.

[55] L. Qi, X. Zhang, W. Dou, C. Hu, C. Yang, and J. Chen, "A two-stage locality-sensitive hashing based approach for privacy-preserving mobile service recommendation in cross-platform edge environment," *Future Generation Computer Systems*, vol. 88, pp. 636–643, 2018.

[56] J. Huang, "Scalable thread sharing analysis," in *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*, pp. 1097–1108, Austin, TX, USA, May 2016.

[57] Z. Wu, K. Lu, X. Wang, and X. Zhou, "Collaborative technique for concurrency bug detection," *International Journal of Parallel Programming*, vol. 43, no. 2, pp. 260–285, 2015.

[58] J. Breese, D. Heckerman, and C. Kadie, "Empirical analysis of predictive algorithms for collaborative filtering," in *Proceedings of the 14th Conference on Uncertainty in Artificial Intelligence (UAI '98)*, pp. 43–52, Madison, WI, USA, July 1998.

[59] L. Qi, Z. Zhou, J. Yu, and Q. Liu, "Data-sparsity tolerant web service recommendation approach based on improved collaborative filtering," *IEICE Transactions on Information & Systems*, vol. E100-D, no. 9, pp. 2092–2099, 2017.