Int J Parallel Prog (2015) 43:260–285
DOI 10.1007/s10766-014-0304-y

Collaborative Technique for Concurrency Bug Detection

Zhendong Wu · Kai Lu · Xiaoping Wang · Xu Zhou

Received: 26 June 2013 / Accepted: 17 January 2014 / Published online: 22 February 2014

Abstract Concurrency bugs hidden in deployed software can cause severe failures and real-world disasters. They are notoriously difficult to detect during in-house testing due to huge and non-deterministic interleaving space. Unfortunately, the multicore technology trend worsens this problem. Unlike previous work that detects particular concurrency bugs (e.g., data races and atomicity violations), we target harmful concurrency bugs that cause program failures. In order to detect harmful concurrency bugs effectively and efficiently, we propose an innovative, collaborative approach called ColFinder. First, ColFinder statically analyzes the program to identify potential concurrency bugs. ColFinder then uses static program slicing to get smaller programs with respect to potential concurrency bugs. Finally, ColFinder dynamically controls thread scheduler to force multiple threads access the same memory location, verifying whether the potential concurrency bug will cause program failure. If a failure occurs, a harmful concurrency bug is detected. We have implemented ColFinder as a prototype tool and have experimented on a number of real-world concurrent programs. The probability of bug manifestation in these programs is only 0.64 % averagely during

Z. Wu (✉) · K. Lu · X. Wang · X. Zhou
Science and Technology on Parallel and Distributed Processing Laboratory,
National University of Defense Technology, Changsha, China
e-mail: wuzhendong@nudt.edu.cn

K. Lu
e-mail: kailu@nudt.edu.cn

X. Wang
e-mail: xiaopingwang@nudt.edu.cn

X. Zhou
e-mail: zhouxu@nudt.edu.cn

Z. Wu · K. Lu · X. Wang · X. Zhou
College of Computer, National University of Defense Technology, Changsha, China

native execution. It is significantly raised to 90 % with ColFinder. The runtime overhead imposed by many previous approaches is always more than 10×. The overhead of ColFinder is more acceptable, with an average of 79 %. Additionally, to our knowledge, this is the first technique that introduces program slicing to reduce the time of bug manifestation, with an average of 33 %.

Keywords Collaborative · Concurrency bug detection · Static program slicing · Dynamic active tester

1 Introduction

1.1 Motivations

Multicore hardware is making concurrent programs increasingly pervasive. However, concurrent programs are notorious for hidden bugs that are difficult to detect and diagnose due to huge and non-deterministic interleaving space. Many concurrency bugs have caused severe failures and real-world disasters such as Northeast blackout of 2003 [1]. As multicore machines grow increasingly popular, detecting concurrency bugs is vital. Unfortunately, concurrency bugs are difficult to detect due to a combination of two conditions to manifest. First, they require the right set of program inputs, which is exponential in number. Second, they also require the right thread interleaving, which is again exponential in number.

According to the comprehensive real world concurrency bug characteristic study in [2], there are three types of concurrency bugs, atomicity-violation bugs, order-violation bugs, and deadlock. Because a data race can also be a benign race in many cases, e.g., while-flag, we take such benign races as false positives. Unlike some previous research that focuses on detecting deadlock bugs [3,4] or atomicity-violation bugs [5–7], we target concurrency bugs that cause program failures. This idea is similar to ConMem [10] and ConSeq [11]. The major difference is that our bug detection technique is forward, and their techniques are backward.

Recently, many static approaches have been proposed to detect concurrency bugs. These approaches analyze the program without running the program [8,9]. They can detect difficult-to-catch concurrency bugs. The major advantage of these approaches is that they seldom miss concurrency bugs due to an overall knowledge of the program. Unfortunately, the static approaches are notorious for reporting many false positives, because they do not reason about the program's full runtime execution context.

To alleviate the above challenge, many dynamic approaches have been proposed to detect concurrency bugs more accurately. These approaches work in two phases. In the first phase, they use predictive dynamic program analysis to identify potential concurrency bugs. In the second phase, they control thread scheduler to trigger specific interleaving to increase the probability of bug manifestation [7,10]. The major advantage of these approaches is that the results are true concurrency bugs due to the precise runtime information. Unfortunately, for large software, each input maps to billions of interleaving, and a concurrency bug may only be exposed by several specific interleavings. Many concurrency bugs are easily missed due to the huge input space

and huge interleaving space. In addition, dynamic approaches always instrument the program that imposes great runtime overhead [8,11,12]. For instance, some dynamic approaches impose more than 20× overhead when detecting memory intensive programs such as FFT [11,12].

To alleviate the limitations of static approaches and dynamic approaches, many researchers adopted other techniques to leverage the advantages of both. Some researchers have combined static analysis and model checking to analyze concurrent programs [13]. However, model checking fails to scale for large programs due to the exponential increase in the number of thread schedules with execution length. Additionally, some researchers have proposed hybrid approaches which interact iteratively to test concurrent programs [9]. Unfortunately, runtime overhead is still a severe problem in dynamic approaches.

Although much progress has been made in this direction, limitations in detecting concurrency bugs remain:

1. *False negatives* Dynamic approaches suffer from coverage issue due to the special input and non-determinism, making many concurrency bugs cannot be covered. In addition, limited patterns of buggy interleaving also make the dynamic approaches missing several unusual bugs, such as some order violations [2] and multi-variable concurrency bugs [14].
2. *False positives* Static approaches report many false positives due to imprecise information. 84 % of the races reported by RELAY [17] are not true races. Some dynamic race detection tools also report benign data races. According to Narayanasamy et al. [15], only about 2–10 % of data races are harmful. Additionally, a similar rate holds for other concurrency bugs [7,10].
3. *Runtime overhead* Runtime overhead is imposed when instrumenting shared memory access instructions. During exercising suspicious interleavings, additional time is imposed due to program instrumentation and conditional delay [11]. Therefore, detecting memory intensive programs such as FFT imposes unacceptable runtime overhead. Some previous work shows that the overhead imposed is more than 20× [10–12].

1.2 Contributions

This paper proposes ColFinder, a collaborative technique to detect concurrency bugs effectively and efficiently. ColFinder first analyzes the program to identify potential concurrency bugs. Each potential concurrency bug is represented by two statements that access the same memory location and at least one of them is a write. ColFinder then uses static program slicing to prune the statements that are irrelevant to potential concurrency bugs. Finally, ColFinder runs a slice, controlling thread scheduler to verify whether the potential concurrency bugs could cause program failures or prune them as false positives.

ColFinder has several useful features.

Manifesting concurrency bugs more efficiently ColFinder uses static program slicing to get a smaller program with respect to the potential concurrency bugs so that verification can be more efficient. This enables the concurrency bugs to be manifested

more efficient. Our experiments show that the time of bug manifestation is reduced significantly, with an average of 33 %. Moreover, it helps the user to understand the concurrency bugs, reducing the time of fixing.

Classifying harmful bugs from benign bugs ColFinder verifies potential concurrency bugs through actual execution so that harmful bugs that could lead to failures get detected. This enables the user to automatically separate harmful bugs from benign bugs, which is otherwise done through manual inspection. Our experiments show that all the harmful bugs are separated from benign bugs successfully.

Improving the probability of bug manifestation ColFinder actively controls thread scheduler to trigger the concurrency bugs so that the hidden bugs which cannot be manifested during native execution are easily detected. For instance, the concurrency bug hidden in LU is not triggered during 100 native runs. ColFinder makes the probability raised to 88 %.

Lower runtime overhead ColFinder instruments only a few instructions which are related to potential concurrency bugs so that lower runtime overhead is imposed. For instance, ColFinder imposes only 122 % overhead for FFT, much better than ConSeq which imposes 2,724 % [11].

We have implemented ColFinder as a prototype tool and have experimented on a number of real-world concurrent programs. The results of these experiments show that all known concurrency bugs in known benchmarks are detected successfully. Overall, our work makes the following contributions:

1. *We firstly use static program slicing to improve the efficiency of concurrency bug detection.* To the best of our knowledge, ColFinder is the first technique that prunes unrelated statements to make dynamic verification of concurrent programs more efficient. Unlike previous work that executes the whole program, ColFinder executes fewer statements that achieves better performance. In addition, ColFinder provides a new perspective on concurrency bug detection and debugging.
2. *We present a three-stage bug-detection tool that leverages characteristics from all three phases of the concurrency bug detection process.* This design makes it easy to leverage advanced static-analysis techniques, such as slicing and imprecise static data race detection. In particular, ColFinder makes the problem of false negatives and false positives alleviated significantly through collaborating static and dynamic techniques.
3. *We use static-analysis to guide dynamic instrumentation that imposes lower runtime overhead.* Unlike previous dynamic bug detectors that instrument all the memory access instructions, ColFinder only instrument a few instructions according to the results of static-analysis. This collaborative design makes ColFinder imposes much lower overhead than previous detectors.

2 Related Work

Recently, concurrency bug detection has attracted extensive research efforts in multi-core era. Many of them have improved software reliability much. In this section, we discuss several key approaches that made important contributions on concurrency bug detection.

2.1 Dynamic Approaches

Recently, many dynamic approaches have been proposed for concurrency bug detection [4–8,10–12,25,26]. Traditional dynamic approaches comprise two phases: a prediction phase and a validation phase. In the first phase, approximate bug detectors are used to predict potential concurrency bugs with a given input in a program. In the second phase, a thread scheduler is actively controlled to exercise a suspicious interleaving in a real execution to verify whether this potential concurrency bug is a true bug.

These approaches predict certain types of concurrency bugs according to interleaving patterns, and verify the potential concurrency bugs through actual runs. Access interleaving invariants are proposed to detect atomicity violations in AVIO [5]. Through automatically extracting access interleaving invariants and detecting violations at run time, AVIO can detect more bugs with much fewer false positives than previous algorithms. Unfortunately, AVIO focuses on atomicity violations, other types of concurrency bugs cannot be detected. Similar to AVIO, AssetFuzzer [26], AtomFuzzer [6], CTrigger [7], MUVI [14] are good at detecting atomicity violations, limited to other types of concurrency bugs. Unlike these tools, we focus on concurrency bugs that cause program failures. Other tools such as RaceFuzzer [25], DeadlockFuzzer [4] are suitable for detecting particular type of concurrency bugs. ColFinder could complement these tools to detect more types of concurrency bugs. ConSeq [11] and ConMem [10] focus on certain effect of concurrency bugs, instead of a specific interleaving pattern, making that these tools instrument only effect related memory access instructions. Different from these two tools, ColFinder focuses on instructions that are related to potential concurrency bugs. Additionally, one of the big problems of previous dynamic approaches is that high runtime overhead is imposed, especially in memory intensive programs. For instance, the runtime overhead imposed by AVIO is more than 15×. In ColFinder, only a few memory access instructions are instrumented so that lower overhead is imposed, with an average of 79 %.

In order to detect data races more efficiently, parallel happens-before and lockset race detectors that rely on uniparallelism are proposed to speed up race detection [35]. However, these two detectors need to instrument all the shared memory access instructions. Unlike parallel method, ColFinder uses program slicing to prune irrelevant codes, making concurrency bug detection more efficient. Griffin [36] provides a way to explain concurrency bugs using additional information over existing fault-localization techniques, and bridges the gap between fault-localization and fault-fixing techniques. However, Griffin's goal is to aid developers understanding the concurrency bugs, and it is built on existing detectors. Unlike Griffin, ColFinder uses existing detectors to aid concurrency bug detection, improving the efficiency. CHESS [37] guides its interleaving testing by bounding the number of preempting context switches to small numbers. However, it still faces the challenge of performance, because its testing space increases polynomially with the program execution length. Sandeep et al. [38] presents variable and thread bounding to rank thread schedules for systematic testing of concurrent programs. The ranking significantly aids early discovery of concurrency bugs. Different from thread bounding, ColFinder does not rank the potential concurrency bugs, but it uses program slicing to discover concurrency bugs early. CLAP [39] can

reproduce concurrency bugs through recording thread local executions that achieves significant advances over previous approaches. Different from CLAP, ColFinder is proposed to detect concurrency bugs, not reproduce bugs. RaceMob [40] is a new data race detector which achieves low overhead and good accuracy by combining real-user crowdsourcing with a new on-demand dynamic race validation technique. It gathers validation results from a large base of users. Unlike RaceMob, ColFinder is designed for in-house testing, and it does not need other resources to help bug detection.

2.2 Static Approaches

Static approaches are explicitly designed to detect bugs in large and complex multi-threaded systems. Racerx [21] builds a control-flow graph of the entire program, adding or removing locks to the lockset when encountering lock or unlock statements. When Racerx encounters accesses of thread-shared variables, it verifies whether their locks are in current lockset, producing a warning or not. LOCKSMITH [16] is designed to detect data races, using context sensitivity to reduce the number of false positives. RELAY [17] is a tool that infers data races based on four ingredients: relative locksets, guarded accesses, function summaries and symbolic execution. However, the static approaches always report many false positives due to imprecise runtime information. In ColFinder, the outputs of static approaches are verified by dynamic active tester, which greatly reduces the false positives. The major difference between static approaches and ColFinder is that ColFinder would verify the outputs of static approaches to make the results of bug detection more accurate. Therefore, ColFinder can complement many previous static approaches.

3 Design

In this section, firstly, we discuss some assumptions for ColFinder. Then, we use an example to illustrate the idea of ColFinder. Finally, we discuss ColFinder's architecture and its three components in detail.

3.1 Assumptions

The goal of ColFinder is to expose as many concurrency bugs as possible during detecting. We design ColFinder based on three assumptions:

1. *The manifestation of concurrency bugs involves two threads.* According to the study on real world concurrency bug characteristics in [2], even though some programs use hundreds of threads, in most cases, only two threads are involved in the manifestation of a concurrency bug. Therefore, ColFinder can focus on execution orders among accesses from every pair of threads, which reduces detecting complexity without losing bug exposing capability much.
2. *Bug triggering inputs are given.* Concurrency bugs require a combination of two unlikely conditions to manifest: (1) right set of inputs, (2) right thread interleavings [27]. Even if a test input can trigger a concurrency bug, it is often difficult to expose

the infrequently occuring buggy thread interleaving, because there can be many correct interleavings for that input. Many previous research assumes that input design is out of the scope of concurrency bug detection [5,11,12], and ColFinder follows the same assumption.

3. *The bugs which cause program failures are harmful concurrency bugs.* Previous work focuses on bugs caused by specific interleavings [5,7,26], we target concurrency bugs that lead the program to crash or output incorrect result. In an earlier work [10], a similar idea is used to detect concurrency bugs. The data races or atomicity-violations that have not make program failure occurs are considered false positives in this paper. ColFinder would stop detecting and report a concurrency bug when the program crashes or outputs incorrect result.

3.2 An Example

Figure 1 shows the example simplified from a concurrency bug in Apache-2.0.48 [28]. This is a data race and also an atomicity violation bug. The bug is obvious. In the '*ap_buffered_log_writer*' function, '*buf->outcnt*' is the index to the end of the log buffer. It is read, checked, and then used to help append the log buffer. After the log buffer update, the '*buf->outcnt*' is increased. However, the whole process is not protected by lock. As a result, the other threads which do the same operation may cause wrong '*buf->outcnt*'.

We want to detect this concurrency bug which is hidden deeply. Traditional dynamic bug detection tools will execute Apache once with particular inputs that may trigger the bug, collecting traces of this correct execution. Then, a large space of thread inter-

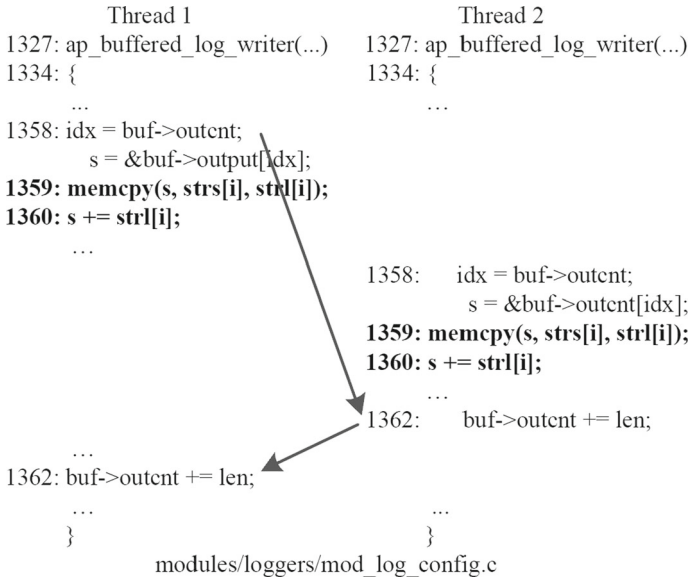

Fig. 1 A concurrency bug in Apache-2.0.48

leaving is analyzed to identify suspicious interleavings. The suspicious interleavings are exercised to verify whether there is a true bug. The limitation is that instrumentation imposes high runtime overhead. We consider if fewer memory access instructions are instrumented, lower overhead is imposed.

Fortunately, static approaches can find the statements that bug-involved. In this example, lines 1358–1362 in *mod_log_config.c* should execute atomically, and other threads execute the same code simultaneously may trigger the bug. Therefore, line 1362 is reported as potential concurrency bug for verification. We can only instrument memory access instructions of this statement, saving much time when exercising particular interleaving.

Then, we observe that it is unnecessary to execute the whole program to verify whether it is a harmful bug because many statements in the program are irrelevant to the potential concurrency bugs. In this example, lines 1359 and 1360 are used to append a string. However, these two statements will not affect the value of '*buf->outcnt*'. Similarly, many other statements are also irrelevant to the potential concurrency bugs. Interestingly, program slicing can extract a smaller program which called a slice, preserving same behavior of the original program with respect to interested variables and a given program point.

Finally, to verify potential concurrency bugs, we observe that particular thread interleaving should be exercised to improve the probability of concurrency bug manifestation. In this example, line 1362 executed in thread 2 may hardly interleave between lines 1358 and 1362 executed in thread 1. We consider to control thread scheduler to exercise particular interleaving. Therefore, we need to design a dynamic active tester, which can suspend one thread for a while that is going to execute instruction that related to the interleaving, to wait for the arrival of other instructions that related to the same interleaving. When these instructions are ready to execute, particular interleaving is forced to verify whether it is a harmful bug.

3.3 Architecture

Figure 2 presents the architecture of ColFinder, comprising three components: (1) the static analyzer, (2) the program slicer, (3) the dynamic active tester.

Static analyzer. ColFinder's static analyzer can identify potential concurrency bugs and provide a report that contains a list of warnings. Each warning consists of a pair of statements. For instance, <12 in select.c, 23 in profile.c> means that line 12 in select.c and line 23 in profile.c executed in different threads may lead to a concurrency bug. Then, the report is sent to program slicer for slicing.

Program slicer. With respect to the potential concurrency bugs and their places, the program slicer extracts several slices from original program. After slicing, the remaining statements are executed to verify whether the potential concurrency bugs will cause program failures. All the slices can be compiled and executed successfully. Then, the slices are sent to the dynamic active tester for verification.

Dynamic active tester. The dynamic active tester exercises specific interleavings, detecting harmful concurrency bugs. It instruments slice's dynamic instructions which are related to potential concurrency bugs and controls thread scheduler to force mul-

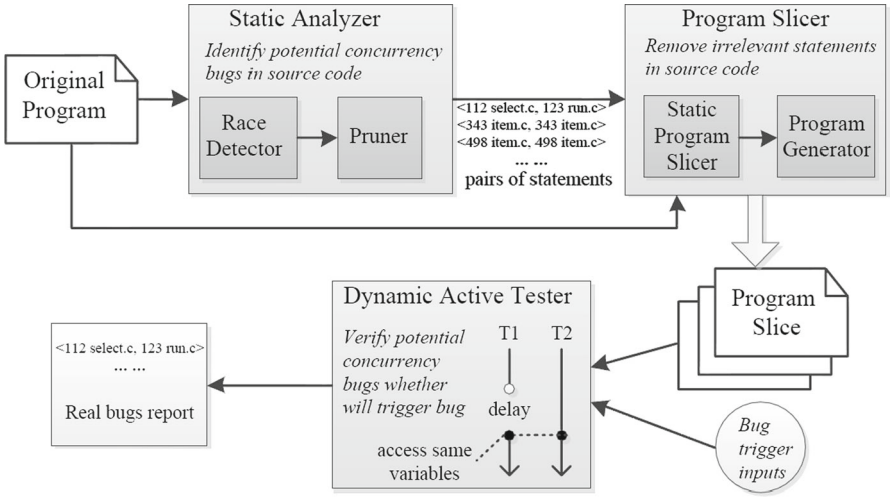

Fig. 2 Architecture of ColFinder

tiple threads access the same memory location simultaneously, improving the probability of bug manifestation. A harmful bug is detected if the slice crashes or outputs incorrect result.

3.4 Static Analyzer

A potential concurrency bug is represented by two statements that access the same memory location and at least one of them is a write. These two statements may be executed in multiple threads that will cause program failures. ColFinder's static analyzer can identify potential concurrency bugs in source code. It has two goals: (1) to identify potential concurrency bugs automatically, (2) to accomplish (1) with better accuracy.

To achieve these two goals, ColFinder follows two design principles:

1. *Use static analysis to automatically analyze every statement that read or write shared variables.* Dynamic analysis may miss some statements due to limited execution path. Therefore, we use static analysis to analyze the whole program to achieve good coverage.
2. *The result of static analysis should be more accurate.* Static analysis tools such as static race detectors always report many false positives due to imprecise runtime information (e.g., synchronous information). We consider synchronous statements to optimize the results.

As shown in Fig. 2, static analyzer consists of two components: (1) race detector, (2) pruner. Race detector can statically identify statements that may access the same memory location by multiple threads. However, it always produces many false positives. The pruner will prune false positives as many as possible.

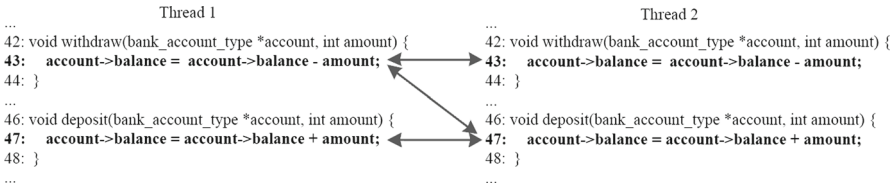

Fig. 3 Potential concurrency bug in bank_account.c

3.4.1 Race Detector

Data races occur when multiple threads are going to access the same memory location and at least one of those accesses is a write. Some races can lead to hard-to-reproduce bugs that are time consuming to debug and fix. An atomicity violation occurs when the desired serializability among multiple memory accesses is violated. An order violation occurs when the desired order between two memory accesses is flipped. Both atomicity violation and order violation occur when multiple threads access the same memory location simultaneously. Therefore, we use data race detector to identify statements that may access the same memory location by multiple threads. These statements may lead to bugs (atomicity violation, order violation or others), or have no effect on the behavior of the program.

Figure 3 shows a simplified example from *bank_account.c*. It is obvious that thread 1 and thread 2 may access variable '*account->balance*' simultaneously, causing the program outputs incorrect result. The race detector walks through the whole program, identifying that line 43 and 47 in *bank_account.c* read and write the same variable '*account->balance*'. Then, the detector reports that this pair of statements is a potential concurrency bug. Therefore, two threads respectively execute lines 43 and 47 may trigger bug.

Fortunately, previous research of static analysis has focused on this problem, some good solutions have been proposed. For instance, RELAY [17] is a static race detector that can infer potential concurrency bugs. It reports pairs of statements that may access the same memory location simultaneously and at least one of them is a write. Such pairs of statements are the root cause of concurrency bugs. In ColFinder, we use RELAY to statically detect data races. We write a plugin that converts RELAY reports to the format required by our program slicer and dynamic active tester.

3.4.2 Pruner

Unfortunately, some static race detectors such as RELAY have not considered synchronization that causes many false positives. For instance, statement *S1* reads global variable *v* before barrier *b* and statement *S2* writes global variable *v* after barrier *b*. RELAY will report *S1* and *S2* is a potential concurrency bug. Actually, *S1* and *S2* will never access global variable *v* simultaneously. Aim at the results reported by RELAY, the pruner will prune the pair of statements that both of them will never access the same memory location simultaneously by synchronization analysis. Pruner analyzes the location of synchronization statements whether make that a pair of statements iso-

```
Algorithm of pruning infeasible potential concurrency bugs.
Input: synList - locations of all synchronous statements
Input: potList - locations of all potential concurrency bugs
Output: remList - remaining potential concurrency bugs
begin
  for aPair in potList
    firstLine = aPair.getFirstLine()
    secondLine = aPair.getSecondLine()
    bool isPairIsolated = FALSE
    for aSyn in synList
      synLine = aSyn.getLine()
      if(isIsolated(firstLine, secondLine, synLine))
        isPairIsolated = TRUE
        break
      end
    end
    if(!isPairIsolated)
      remList.add(aPair)
    end
  end
  return remList
end
```

Fig. 4 Algorithm of pruning infeasible potential concurrency bugs

lated. Figure 4 shows the algorithm of how to prune such infeasible pairs of statements. The function '*isIsolated*' is responsible for checking whether *firstLine* and *secondLine* are isolated by *synLine*.

Being imprecise in nature, many static race detectors require manual inspection to see if a race is real or not. The pruner can roughly prune some potential concurrency bugs which are infeasible automatically. We cannot make sure that all the infeasible potential concurrency bugs are pruned. However, the pruner can prune many infeasible potential concurrency bugs, which improves the efficiency of verification significantly.

3.5 Program Slicer

The variables in potential concurrency bugs may be accessed by multiple threads simultaneously. To manifest the concurrency bugs more efficient, ColFinder extracts all the statements which are relevant to these variables and a given program point. ColFinder's program slicer uses static analysis to identify the relevant statements automatically, comprising two components: (1) the static program slicer, (2) the generator. The static program slicer can identify information of all statements which are relevant to potential concurrency bugs. According to the information, the program generator will generate some smaller programs.

3.5.1 Static Program Slicer

Figure 5 shows a simplified example from *bank_account.c* that statements irrelevant to variable '*account->balance*' are pruned. Lines 43 and 47 are identified by static analyzer because they read and write the same variable '*account->balance*'. Line 52 reads this variable that has no impact on the result of lines 43 and 47. Line 54 outputs

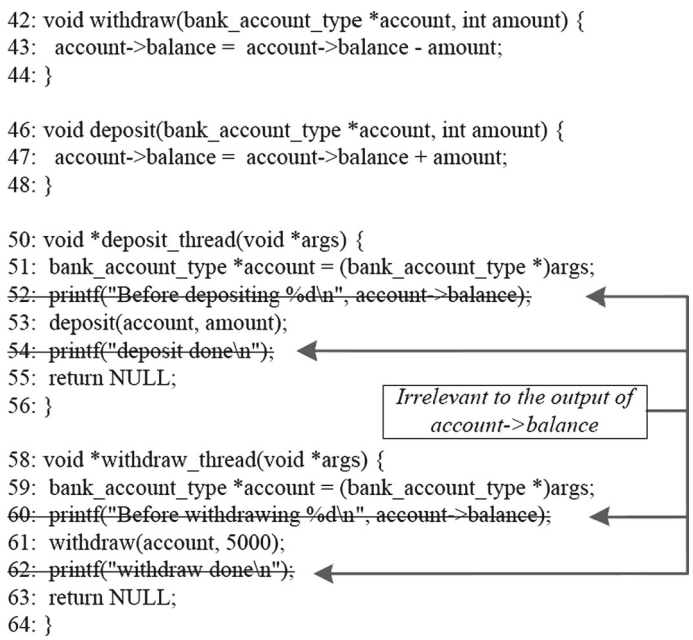

Fig. 5 Irrelevant code to the output of account in bank_account.c

a string that of course has no impact on the identified statements. Therefore, lines 52 and 54 are irrelevant to the result of lines 43 and 47. Pruning these two statements will not affect the result of verification.

Fortunately, many researchers have made great contributions on program slicing and some tools are publicly available. For instance, Unravel [24] is a static program slicing tool that can highlight the statements with respect to interested variables and a given program point.

3.5.2 Generator

The limitation is that Unravel cannot extract the slices automatically. Therefore, generator can produce slices according to the results of Unravel. The results of Unravel are saved into a file. The content of this file is simple description of relevant code to the interested variables (e.g., 43 in *bank_account.c*, 47 in *bank_account.c*). Then, the generator walks through the whole program, selecting the statements that described in the file, exporting these statements into a new source file.

A slice is a smaller program with respect to the interested variables and a given program point. Compare with the original program, the remaining statements are related to the interested variables. If the program crashes because of the interfered memory access operations to the interested variables, the harmful concurrency bug is detected. If the outputs of the interested variables are incorrect due to interfered accesses, the harmful concurrency bug is detected too. Therefore, the output of the program is not important for bug detection. We focus on the interested variables to detect harmful concurrency bugs.

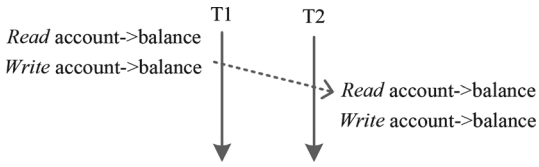

Fig. 6 Usual instruction sequence from bank_account

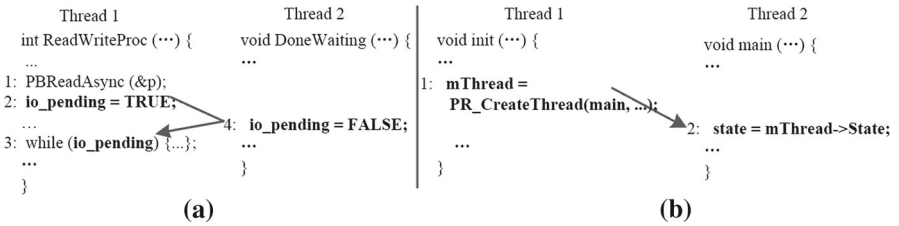

Fig. 7 Two order violation bugs from Mozilla, **a** from Mozilla macio.c, macthr.c, **b** from Mozilla nsthread.cpp

3.6 Dynamic Active Tester

ColFinder's dynamic active tester can exercise particular thread interleavings to verify whether potential concurrency bugs can cause program failure or prune them as false positives. The concurrency bug in *bank_account.c* is obvious. However, in most cases, this bug will not manifest during native execution. Figure 6 shows the usual instruction sequence of the potential concurrency bug by two threads in *bank_account.c*. We have run the original program 1,000 times, and none of them trigger the bug. The main reason is that instructions '*read account->balance*' and '*write account->balance*' execute consecutively and immediately in one thread, making that the same instructions in other threads hardly execute simultaneously. ColFinder controls the thread scheduler to force instructions '*read account->balance*' and '*write account->balance*' execute simultaneously by multiple threads. Therefore, these two instructions executed in one thread may be interleaved by other instructions in other threads, making the value of variable '*account->balance*' is incorrect. Then, we verify whether the potential concurrency bug will cause program failure.

The concurrency bug in *bank_account.c* can be classified as an atomicity violation. Figure 7 shows two order violation bugs from Mozilla. The solid arrows represent correct interleavings. In Fig. 7a, the programmer's order intention is that thread 2 is expected to write *io_pending* to be FALSE sometime after thread 1 initializes it to be TRUE. However, line 4 could unexpectedly execute before line 2, in which case the FALSE value of *io_pending* would be overwritten too early. In Fig. 7b, the programmer's intention is that thread 2 should not read *mThread* until thread 1 initializes *mThread*. However, thread 2 may be very quick and dereference *mThread* before *mThread* is initialized. This unexpected order leads to program crash. ColFinder controls the thread scheduler to make this unexpected order execute more probable. In summary, the goal is to control the thread scheduler to improve the probability of bug manifestation.

To achieve this goal, we describe the core idea based on the bugs described above. Figure 8 shows the core idea of dynamic active tester on *bank_account.c*. ColFinder can

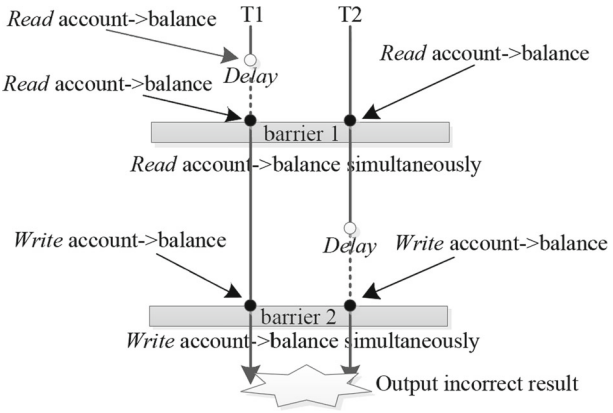

Fig. 8 Core idea of dynamic active tester on bank_account.c

force multiple threads access the same memory location simultaneously. We have mentioned that only two threads are involved in bug manifestation. During the execution of concurrent program, most of the threads are irrelevant to the bug manifestation. We force two threads access the same memory location simultaneously, verifying whether a failure occurs. According to the comprehensive study on real world concurrency bug characteristics [2], 96 % of the examined concurrency bugs are guaranteed to manifest if certain partial order between 2 threads is enforced. Therefore, this assumption reduces detecting complexity without losing bug detection result much.

During the execution, ColFinder instruments the program binary and inserts conditional delays with time-outs before every instruction of the potential concurrency bugs. At run time, the instrumented code suspends for a while thread T1 that is going to execute '*read account->balance*', to wait for the arrival of the same instruction in thread T2. When both of these instructions are ready to execute, the instrumented code will randomly select one of them to execute first. After barrier 1, thread T2 that is going to execute '*write account->balance*' is suspended for a while by the instrumented code, to wait for the arrival of the same instruction in thread T1. When both of these instructions are ready to execute, one of them is selected to execute first. Therefore, the instructions in thread T2 execute between the consecutive instructions in thread T1 is more probable than native execution. Similarly, probability that the instructions in thread T1 execute between the consecutive instructions in thread T2 is also improved.

Figure 9 shows the core idea of dynamic active tester on Mozilla, improving the probability of multiple threads access the same memory location simultaneously. In Fig. 9a, the instrumented code suspends for a while thread T1 that is going to execute '*write io_pending*', to wait for the arrival of '*write io_pending*' in thread T2. When both of these instructions are ready to execute, the instrumented code will randomly select one of them to execute first. Then, thread T2 may write *io_pending* before thread T1, making that the value written by thread T1 is invalid. In Fig. 9b, the instrumented code suspends for a while thread T2 that is going to execute *'write mThread->State'*, to wait for the arrival of '*read mThread->State*' in thread T1. Then, the instrumented

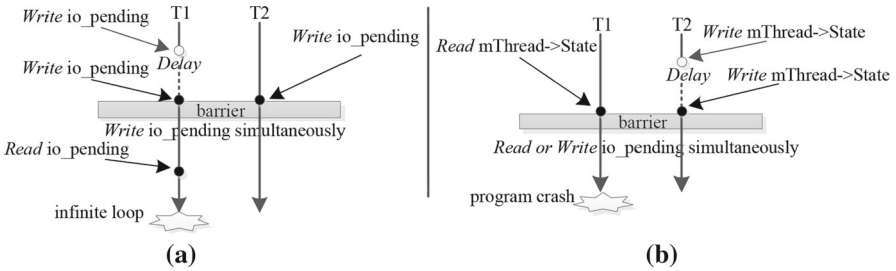

Fig. 9 Core idea of dynamic active tester on buggy Mozilla, **a** from Mozilla macio.c, macthr.c, **b** from Mozilla nsthread.cpp

```
Thread 1                                    Thread 2
...                                         ...
42: for (int i = 0; i < SIZE; i++) {        48: for (int j = 0; j < LENGTH; j++) {
43:     sum += numbers[i];          <-->    49:     sum += values[j];
44: }                                       50: }
45: printf("sum is %d\n", sum);             ...
...
```

Fig. 10 A simple example for loop optimization

code will select one of them to execute first, making that thread T1 may read an invalid value. When program crash occurs, ColFinder reports that a harmful concurrency bug is detected.

The difference between ColFinder's dynamic active tester and traditional dynamic active testers is that the instrumented code imposes different runtime overhead. The traditional dynamic active testers run the program with bug-triggering inputs, collecting the trace during correct executions. Then, they analyze the trace to get suspicious interleavings according to the interleaving patterns. Based on the suspicious interleavings, every shared memory access of the whole program is still instrumented to match the suspicious interleavings during verification. If matched, they control thread scheduler to exercise the suspicious interleaving to improve the probability of bug manifestation. In ColFinder, static analyzer identifies the potential concurrency bugs that will guide dynamic active tester to instrument only the instructions related to potential concurrency bugs. Therefore, the runtime overhead imposed by ColFinder is lower.

We apply ColFinder on a test suite during in-house testing to expose hidden concurrency bugs, so the source code should be compiled with debug information (e.g., with -g option by gcc). Like many previous tools [10,11,29], ColFinder cannot provide 100 % guarantee to trigger every concurrency bug. However, we can run ColFinder several times to trigger the bug close to 100 %.

4 Optimization

Although we have described the design of ColFinder in detail, there is still a large space to achieve better performance and better accuracy. Therefore, in this section, we discuss two important strategies to improve performance and accuracy.

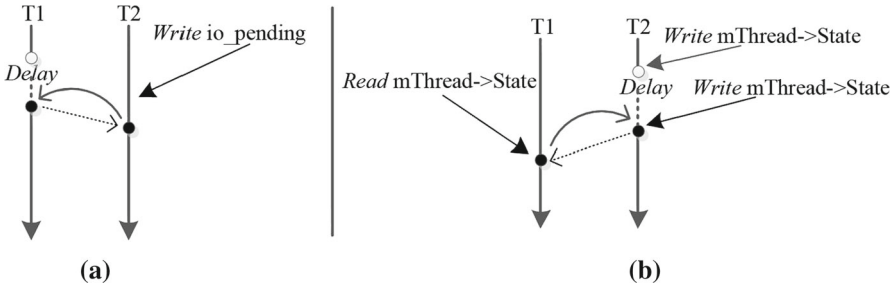

Fig. 11 Bug is missed due to short delay, **a** Mozilla (macio.c, macthr.c), **b** Mozilla (nsthread.cpp)

4.1 Loop Optimization

We observe that it is unnecessary to verify potential concurrency bugs in loop again and again. Figure 10 shows a simple example that will be optimized. It is obvious that line 43 and line 49 execute in thread 1 and thread 2 respectively will lead to incorrect result. ColFinder instruments the program binary. The instrumented code suspends for a while thread 1 that is going to execute '*read sum*' or '*write sum*'. Unfortunately, lines 43 and 49 are in loop that will cause the instrumented code repeatedly execute 1,000 times. Of cause the instrumented code is unnecessary to execute 1,000 times.

To alleviate this problem, we extend the static analyzer to infer whether the potential concurrency bugs are in loop. If the potential concurrency bugs are in loop, ColFinder makes the instrumented code only execute several times. Therefore, although lines 43 and 49 execute 1,000 times, the instrumented code only execute several times. This strategy makes that ColFinder has better performance than traditional tools, because the traditional tools almost instrument every memory access in loop which causes poor performance.

4.2 Adaptive Delay

ColFinder makes multiple threads access the same memory location simultaneously more probable. The probability of bug manifestation is significantly improved. However, if the thread is suspended too short, the bug will not be triggered. Figure 11a shows that bugs in Mozilla is not manifested (Solid and dotted arrows represent incorrect and correct sequence, respectively). The instrumented code suspends for a while thread T1 that is going to execute '*write io_pending*', to wait for the arrival of instruction '*write io_pending*' executed in thread T2. Unfortunately, instruction '*write io_pending*' executed in thread T2 is not arrived when thread T1 times out for waiting. Then, thread T1 executes instruction '*write io_pending*' continuously. Therefore, the order violation is not triggered. If the delay time is long enough, instruction '*write io_pending*' in thread T2 is arrived before thread T1 executes instruction '*write io_pending*'. Then, these two threads will access the same memory location simultaneously. The bug will be manifested more probable. Figure 11b is similar.

To alleviate this problem, we introduce adaptive delay in ColFinder. Figure 12 shows the core idea of this strategy. Figure 12a shows that the order violation bug is

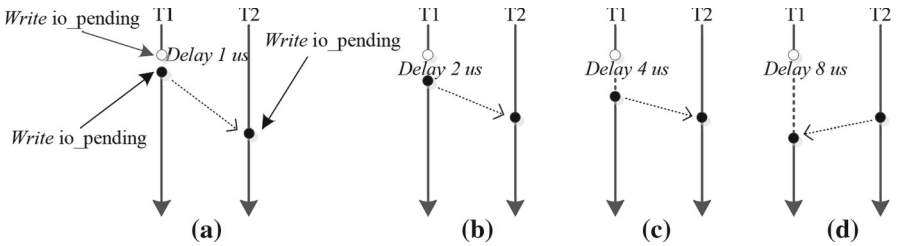

Fig. 12 Increase delay time to trigger bug (order violation in Mozilla)

not triggered when time-out for waiting is 1 ms, ColFinder will change the time-out for waiting by itself. Figure 12b shows that the bug is still not triggered when time-out for waiting is 2 ms. Then, ColFinder will change the time-out for waiting again. Figure 12c shows that the bug is still not triggered when time-out for waiting is 4 ms. Then, ColFinder change the time-out for waiting again. Finally, Fig. 12d shows that the bug is triggered when time-out for waiting is 8 ms, and ColFinder need not to change the time-out for waiting. If the time-out for waiting is changed 10 times, and the bug is still not triggered, ColFinder reports it as a false positive. We set the time-out for waiting to 2^n ms, n is a integer variable, from 0 to 9.

5 Implementation

We have implemented ColFinder which can be divided into three components (static analyzer, program slicer and dynamic active tester). In this section, we briefly describe the implementation of these components.

The static analyzer is implemented based on RELAY that is a static and scalable race detection tool [17]. We implemented pruner to optimize the results. The optimization strategy is pruning some warnings that are infeasible to occur. The pruner analyzes all the potential concurrency bugs reported by RELAY, saving a more accurate result into a file.

The program slicer is implemented based on Unravel that is a static program slicing tool [24]. The limitation of Unravel is that it only highlights the relevant statements with respect to the interested variables and a given program point. We implemented Generator to extract a slice according to the results of Unravel.

The dynamic active tester is implemented using the PIN [20] binary-instrumentation framework. At run time, it intercepts interested dynamic instructions that related to the potential concurrency bugs provided by static analyzer, and injects delay to trigger the bug or report a false positive.

6 Experimental Analysis

The goal of our work is to provide an effective and efficient collaborative concurrency bug detection approach. Accordingly, our evaluation aims at answering the follow two questions:

1. *Effectiveness—Can ColFinder detect concurrency bugs effectively?* To answer this question, we used 14 programs that each of them has a known concurrency bug. Whether all the bugs can be detected was to evaluate the effectiveness of ColFinder.
2. *Efficiency—How efficient is ColFinder for detecting concurrency bugs ?* To answer this question, we compared the runtime overhead of ColFinder with traditional bug detection tools. Additionally, we checked whether program slicing can reduce the time of bug manifestation.

All the experiments were conducted on an AMD server with a 2.2 GHz, 12-core CPU (AMD Opteron 6174) and 16 GB physical memory, running Linux kernel version 2.6.31.5.

6.1 Experimental Setup

We evaluated ColFinder on 14 programs that each of them has a known concurrency bug. With bug triggering input, ColFinder can quickly expose the bug. The first column in Table 1 shows the ID of programs. The second column in Table 1 shows the programs in our experiments. The third column in Table 1 shows the bug description of each program. Among these benchmarks, 4 (#1 to #4) are kernel programs studied in Atom-aid [30] and Maple [12], 1 (#5) is a code snippet extracted from a real buggy program, 7 (#6 to #12) are scientific programs from splash2 [31] that each of them has a real bug, 1 (#13) is a parallel implementation of the bzip2 block-sorting file compressor, 1 (#14) is a parallel file scanning tool that provides the combined functionality of find, xargs, and fgrep in a parallel way.

Many experiments have been done to evaluate ColFinder. To evaluate the probability of bug manifestation, the results are computed across 100 runs. The reported performance numbers are the average across 20 runs (we did 30 runs, 5 maximum and 5 minimum were removed).

For comparison, we used the experimental results directly from state-of-the-art concurrency bug detection approaches. Because they are not publicly available and the performance may change a lot in our implementation. We roughly compared the runtime overhead of these approaches with ColFinder on the same benchmarks.

6.2 Effectiveness

Table 2 shows that all the known concurrency bugs can be detected effectively by ColFinder. Column 3 reports the buggy statements in program by static analyzer (e.g., 45, 52 means that line 45 and line 52 may be executed in multiple threads to trigger the concurrency bug). Column four reports the detection results by dynamic active tester before slicing (Y means it is a true bug, N means it is a false positive). Column five reports the detection results by dynamic active tester after slicing. As we can see, ColFinder detected all the known bugs in known benchmarks. Additionally, program slicing will not affect the results of bug detection.

Table 3 shows that the probability of bug manifestation is improved significantly with the help of dynamic active tester in ColFinder. Columns 3–4 report the probability

Table 1 Benchmarks

ID	programs	Bug description
1	BankAccount	Simultaneous withdrawal and deposit with incorrectly synchronized.
2	CircularList	Removing, processing, and adding work units to list non-atomically
3	LogProcSweep	Threads inconsistently manipulate shared log
4	Stringbuffer	On append, not all required locks are held
5	MySQL-LogMiss	MySQL log system bug
6	LU	Multi threads increment global id without locks
7	FFT	Multi threads read and write global id without locks
8	Radix	Wrong synchronous barrier cause wrong outputs
9	Ocean	Multi threads read and write global variables without locks
10	Cholesky	Multi threads modify shared variable cause memory error
11	Barnes	Multi threads read and write global variables without locks, causing wrong outputs
12	FMM	Multi threads add a global variable without locks, causing program crash
13	pbzip2	Main thread does not join the consumer threads, causing an order violation
14	pfscan	Multi threads read and write a shared variable without locks, causing program crash

Table 2 Bug detection results before and after slicing

ID	Programs	Buggy lines	Is true concurrency bug?	
			Before slicing	After slicing
1	BankAccount	45, 52	Y	Y
2	CircularList	104, 105	Y	Y
3	LogProcSweep	69, 78	Y	Y
4	Stringbuffer	162, 174	Y	Y
5	MySQL-LogMiss	88, 97	Y	Y
6	LU	351, 351	Y	Y
7	FFT	435, 435	Y	Y
8	Radix	502, 502	Y	Y
9	Ocean	861, 861	Y	Y
10	Cholesky	4,615, 4,615	Y	Y
11	Barnes	2,428, 2,428	Y	Y
12	FMM	3,483, 3,484	Y	Y
13	pbzip2	903, 1,052	Y	Y
14	pfscan	762, 933	Y	Y

Table 3 Probability of bug manifestation with and without dynamic active tester in ColFinder

ID	Programs	Without dynamic active tester		With dynamic active tester	
		Before slicing (%)	After slicing (%)	Before slicing (%)	After slicing (%)
1	BankAccount	0	0	71	87
2	CircularList	0	0	78	91
3	LogProcSweep	0	0	85	92
4	StringBuffer	0	0	86	92
5	MySQL-LogMiss	0	0	62	84
6	LU	0	0	89	88
7	FFT	0	0	96	88
8	Radix	7	9	87	86
9	Ocean	0	0	92	95
10	Cholesky	2	4	62	96
11	Barnes	0	0	92	90
12	FMM	0	0	78	98
13	pbzip2	0	0	98	95
14	pfscan	0	0	65	73
Average		0.64	0.93	82	90

before and after slicing without dynamic active tester respectively. Columns 5–6 report the probability before and after slicing with dynamic active tester respectively. As we can see, the bugs are hidden in programs deeply except radix and cholesky, exposing these bugs is almost impossible during native execution. In most cases, the probability after slicing is higher than the probability before slicing. The special cases are the bugs in radix and cholesky caused by wrong synchronous barrier. Triggering these two bugs which can lead to wrong output or memory error during native execution is more probable than others.

Table 4 shows that ColFinder has good accuracy in concurrency bug detection. Column 3 reports the number of warning from static analysis. The warnings are considered as potential concurrency bugs. Of course there are many false positives in these warnings. Column 4 reports the number of true bug detected by ColFinder. As we can see, dynamic active tester prunes all the false positives, exposing the known bug successfully.

From Tables 2, 3, 4, we conclude that ColFinder can detect all the known bugs effectively. Comparing to static bug detection tools, ColFinder has much better accuracy. Therefore, ColFinder can complement many previous static bug detectors.

6.3 Efficiency

ColFinder is designed for in-house testing and goes through three phases. According to the evaluation methodology in [11], we only consider the efficiency of dynamic active tester. The first phase of ColFinder uses static analysis to identify potential concurrency

Table 4 False positives greatly reduced by dynamic active tester

ID	Programs	Warnings	True bugs
1	Bankccount	3	1
2	CircularList	17	1
3	LogProcSweep	4	1
4	StringBuffer	3	1
5	MySQL-LogMiss	3	1
6	LU	49	1
7	FFT	76	1
8	Radix	65	1
9	Ocean	197	1
10	Cholesky	184	1
11	Barnes	273	1
12	FMM	308	1
13	pbzip2	101	1
14	pfscan	58	1

Table 5 Runtime overhead without slicing in ColFinder

ID	Programs	Native (us)	ColFinder (us)	Overhead(%)
1	LU	1,248,821	2,446,179	96
2	FFT	2,135,627	4,732,538	122
3	Radix	1,644,712	2,931,351	78
4	Ocean	7,967,246	9,061,864	14
5	Cholesky	1,152,787	1,771,838	54
6	Barnes	840,114	1,785,865	113
7	FMM	1,409,419	2,140,126	52
8	pbzip2	1,050,198	2,053,096	96
9	pfscan	605,316	1,131,458	87
Average				79

bugs. This step is not performance-critical, because it is conducted only once for each piece of code. The second phase of ColFinder uses static program slicing to extract a set of program statements consists of all statements that may trigger the concurrency bugs. This step is still not performance-critical, because it is conducted offline, making no impact on runtime overhead. The slice can be re-used across different testing runs and different inputs. The third phase of ColFinder uses dynamic active tester to control the thread scheduler to improve the probability of bug manifestation. This step is conducted online, affecting the runtime overhead greatly.

Table 5 shows that the overhead imposed by ColFinder is acceptable. Column 3 reports the native execution time for base line. Column 4 reports the execution time under dynamic active tester. Column 5 reports the overhead imposed by dynamic active tester. As we can see, the runtime overhead imposed by ColFinder ranges from 14 to 122 %. It is much lower than the traditional concurrency bug detectors. We roughly compared the runtime overhead imposed by ColFinder with several previous tools.

Table 6 Runtime overhead comparison

Programs	Concurrency bug detection tools						
	ColFinder (%)	AVIO	ConMem	ConSeq	Maple	HB-detector	L-detector
LU	96	23×	–	–	–	29×	23×
FFT	122	42×	16×	27×	16.3×	48×	40×
Radix	78	15×	–	–	17.8×	5×	4×
Ocean	14	–	–	–	–	30×	23×
FMM	52	19×	–	–	–	–	–
pbzip2	96	–	–	–	45.4×	22×	13×
pfscan	87	–	–	–	27.7×	–	–

("–" means it is not evaluated)

Since the previous tools are not publicly available, and the performance may change a lot in our implementation, we directly use the experimental results from state-of-the-art bug detection tools.

Table 6 shows the comparison results. HB-detector and L-detector are shorten for parallel happens-before race detector and parallel lockset race detector, respectively [35]. In ConSeq, the runtime overhead imposed is 2,724 % when detecting FFT [11]. ConSeq monitors one run of the program to find suspicious interleavings although it is usually a correct run. Then, ConSeq controls thread scheduler to exercise the suspicious interleavings that can trigger the bugs or prune them as false positives. However, great runtime overhead is imposed due to instrumenting most of the shared memory accesses. Different from ConSeq, ColFinder only instruments a few memory accesses which are identified by static analyzer, imposing lower overhead. Similar to ConSeq, ConMem [10] imposes 16× runtime overhead that is still much worse than ColFinder. Additionally, Maple [12] imposes 16.3× when detecting FFT which is only 122 % by ColFinder, 17.8× when detecting radix which is only 78 % by ColFinder. Software implementation of AVIO imposes 23× when detecting LU which is only 96 % by ColFinder, 42× when detecting FFT which is only 122 % by ColFinder, 15× when detecting radix which is only 78 % by ColFinder. HB-detector and L-detector impose at least 4× runtime overhead, which is greatly higher than ColFinder. In summary, the performance of ColFinder is much better than traditional concurrency bug detection tools especially memory intensive programs.

Table 7 shows that program slicing can reduce the time of bug manifestation. Column 3–4 report the time of dynamic active tester before and after slicing, respectively. Column 5 reports the time reduced by slicing, with an average of 33 %. Many statements irrelevant to potential concurrency bugs are pruned by slicing. Another benefit of program slicing is that developers can debug and fix the concurrency bugs more quickly.

Figure 13 shows that the execution time of each program is stable. To manifest the bug, ColFinder control thread scheduler, improving the probability of bug manifestation. Execution time of FFT and ocean are only a little unstable due to the complexity of program behavior. The others are much more stable.

Table 7 Time of dynamic active tester reduced by slicing

ID	Programs	Dynamic active tester (us)		
		Before slicing	After slicing	Time reduced (%)
1	BankAccount	1,694,885	1,350,375	20
2	CircularList	1,865,889	1,535,437	18
3	LogProcSweep	1,649,485	1,356,572	18
4	StringBuffer	1,846,378	1,496,160	19
5	MySQL-LogMiss	1,913,119	1,508,029	21
6	LU	2,446,179	1,339,206	45
7	FFT	4,732,538	1,356,345	71
8	Radix	2,931,351	2,365,832	19
9	Ocean	9,061,864	3,256,984	64
10	Cholesky	1,771,838	1,458,528	18
11	Barnes	1,785,865	1,285,105	28
12	FMM	2,140,126	919,006	57
13	pbzip2	2,053,096	1,409,669	31
14	pfscan	1,131,458	814,871	28
Average				33

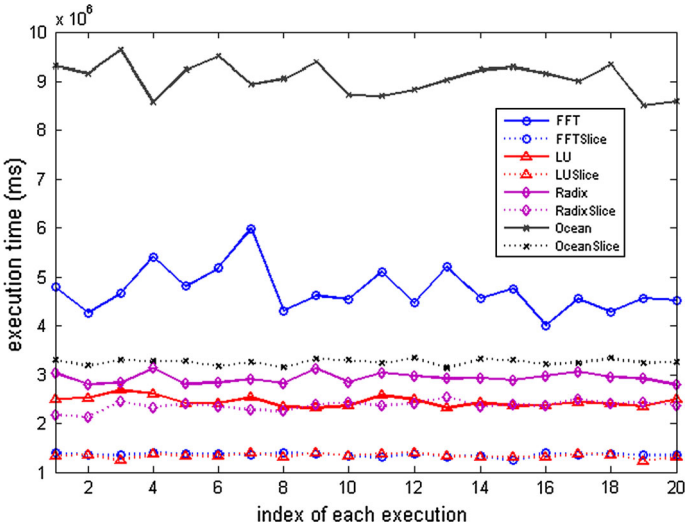

Fig. 13 Stability of ColFinder (time of each execution is stable)

7 Conclusion and Future Work

This paper has presented an innovative, collaborative approach to detect concurrency bugs. There are three phases in bug detection. First, we use static analysis to identify potential concurrency bugs. Second, according to the statements identified by static

analysis, we use program slicing to prune irrelevant codes. Third, we actually execute each slice to verify whether the potential concurrency bugs will cause program failure. ColFinder is implemented as a prototype tool. With the help of static analysis, ColFinder instruments only a few memory accesses. Therefore, runtime overhead is greatly reduced, especially memory intensive programs such as FFT. With program slicing, ColFinder verifies the potential concurrency bugs more efficient.

As the emerging parallel programming becomes more popular due to the multi-core technology trend, concurrency bug detection will become increasingly urgent. To the best of our knowledge, comprehensive, low-overhead, collaborative approaches in detecting concurrency bugs are never provided. Additionally, program slicing is also firstly introduced to improve the efficiency of bug detection.

ColFinder still has limitations. Currently, ColFinder can only detect concurrency bugs in C programs. ColFinder cannot detect all types of concurrency bugs. Although ColFinder can statically identify the statements that may access the same memory location, it cannot detect deadlocks. Static program slicing is not scalable for large multi-threaded programs and may miss to identify some statements that are related by the execution environment. Therefore, it may limits the applicability of ColFinder.

Future work on ColFinder will concern several aspects. First, we plan to improve the efficiency of ColFinder especially the dynamic active tester. Second, we plan to design our own static analysis tool and program slicing tool to support more languages and achieve better efficiency. Third, we plan to use deterministic techniques [32] to eliminate nondeterminism of multi-threaded programs to help concurrent program testing.

Acknowledgments We appreciate the kind comments and professional criticisms of the anonymous reviewers. This work was partially supported by National High-tech R&D Program of China (863 Program) under Grants 2012AA01A301 and 2012AA010901, by program for New Century Excellent Talents in University and by National Science Foundation (NSF) China 61272142, 61103082, 61003075, 61170261 and 61103193. Moreover, it is a part of Innovation Fund Sponsor Project of Excellent Postgraduate Student B130608. An earlier version of this paper [34] appeared in The 13th International Conference on Quality Software.

References

1. Securityfocus: Software bug contributed to blackout. http://www.securityfocus.com/news/8016 (2013). Accessed 1 May 2013
2. Lu, S., Park, S., Seo, E., Zhou, Y.: Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In: Architectural Support for Programming Languages and Operating Systems (ASPLOS 2008), pp. 329–339. Seattle, Washington, USA (2008)
3. Koskinen, E., Herlihy, M.: Dreadlocks: efficient deadlock detection. In: Proceedings of the 12th annual symposium on Parallelism in algorithms and architectures, Munich, Germany, pp. 297–303 (2008)
4. Joshi, P., Park, C.-S., Sen, K., Naik, M.: A randomized dynamic program analysis technique for detecting real deadlocks. In: 30th annual ACMSIGPLAN conference on Programming Language Design and Implementation (PLDI 2009), Dublin, Ireland, pp. 110–120 (2009)
5. Lu, S., Tucek, J., Qin, F., Zhou, Y.: AVIO: detecting atomicity violations via access interleaving invariants. In: Architectural Support for Programming Languages and Operating Systems (ASPLOS 2006), pp. 37–48. San Jose, CA, USA (2006)
6. Park, C.-S., Sen, K.: Randomized active atomicity violation detection in concurrent programs. In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering(FSE), pp. 135–145. Atlanta, Georgia, USA (2008)

7. Park, S., Lu, S., Zhou, Y.: CTrigger: exposing atomicity violation bugs from their hiding places. Ctrigger: exposing atomicity violation bugs from their hiding places. In: 15th Architectural Support for Programming Languages and Operating Systems (ASPLOS 2009), pp. 25–36. Washington, DC, USA (2009)
8. Huang, J., Zhang, C.: Persuasive prediction of concurrency access anomalies. In: Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA 2011), pp. 144–154. Toronto, ON, Canada (2011)
9. Chen, J., MacDonald, S.: Towards a better collaboration of static and dynamic analyses for testing concurrent programs. In: Proceedings of the 6th workshop on Parallel and distributed systems: testing, analysis, and debugging (PADTAD 2008). Seattle, WA, USA (2008)
10. Zhang, W., Sun, C., Lu, S.: ConMem: detecting severe concurrency bugs through an effect-oriented approach. In: 15th Architectural Support for Programming Languages and Operating Systems (ASPLOS 2010), pp. 179–192. Pittsburgh, PA, USA (2010)
11. Zhang, W., Lim, J., Olichandran, R., Scherpelz, J., Jin, G., Lu, S., Reps, T.: ConSeq: detecting concurrency bugs through sequential errors. In: 16th Architectural Support for Programming Languages and Operating Systems (ASPLOS 2011), pp. 251–264. Newport Beach, CA, USA (2011)
12. Yu, J., Narayanasamy, S., Pereira, C., Pokam, G.: Maple: a coverage-driven testing tool for multithreaded programs. In: Proceedings of the ACM international conference on Object oriented programming systems languages and applications (OOPSLA 2012), pp. 485–502. Tucson, AZ, USA (2012)
13. Brat, G., Visser, W.: Combining static analysis and model checking for software analysis. In: Proceedings of the 16th IEEE international conference on Automated software engineering (ASE 2001), pp. 262–269 (2001)
14. Lu, S., Park, S., Hu, C., Ma, X., Jiang, W., Li, Z., Popa, R.A., Zhou, Y.: MUVI: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In: Proceedings of the 21th ACM symposium on Operating systems principles (SOSP 2007), pp. 103–116. Stevenson, WA, USA (2007)
15. S. Narayanasamy, Z.W., J. Tigani, A. Edwards, and B. Calder: Automatically classifying benign and harmful data races using replay analysis. In: 28th annual ACMSIGPLAN conference on Programming Language Design and Implementation (PLDI 2007), pp. 22–31 (2007)
16. Pratikakis, P., Foster, J.S., Hicks, M.: Locksmith: practical static race detection for c. ACM Trans. Program. Lang. Syst. (TOPLAS) **33**(3), 1–55 (2011)
17. Voung, J.W., Jhala, R., Lerner, S.: RELAY: static race detection on millions of lines of code. In: 15th ACM SIGSOFT International Symposium on Foundations of software engineering (FSE 2007), pp. 205–214. Cavtat near Dubrovnik, Croatia (2007)
18. Giffhorn, D., Hammer, C.: Precise slicing of concurrent programs. J. Autom. Softw. Eng. doi:10.1007/s10515-009-0048-x pp. 197–234 (2009)
19. X. Zhang, R.G.: Cost effective dynamic program slicing. In: 25th annual ACMSIGPLAN conference on Progr-amming Language Design and Implementation (PLDI 2004), pp. 94–106 (2004)
20. Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation. In: 26th annual ACMSIGPLAN conference on Programming Language Design and Implementation (PLDI 2005), pp. 190–200. Chicago, Illinois, USA (2005)
21. Engler, D., Ashcraft, K.: RacerX: effective, static detection of race conditions and deadlocks. In: Proceedings of the 19th ACM symposium on Operating systems principles (SOSP 2003), pp. 237–252 (2003)
22. Balakrishnan, G., Gruian, R., Reps, T., Teitelbaum, T.: CodeSurfer/x86 - A platform for analyzing x86 executables. In: Proceedings of the 14th International Conference on Compiler Construction (CC 2005), pp. 250–254 (2005)
23. Balakrishnan, G., Reps, T.: Analyzing memory accesses in x86 executables. In: Proceedings of the 13th international conference on Compiler Construction (CC 2004), pp. 5–23 (2004)
24. Lyle, J.R., Wallace, D.R.: Using the unravel program slicing tool to evaluate high integrity software. In: Proceedings of 10th International Software Quality Week, vol. 301, pp. 975–3270 (1997)
25. Sen, K.: Race directed random testing of concurrent programs. In: 29th annual ACMSIGPLAN conference on Program-ming Language Design and Implementation (PLDI 2008), pp. 11–21. Tucson, Arizona, USA (2008)

26. Lai, Z., Cheung, S.C., Chan, W.K.: Detecting atomic-set serializability violations in multithreaded programs through active randomized testing. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE 2010), pp. 235–244. Cape Town, South, Africa (2010)
27. Chew, L., Lie, D.: Kivati: fast detection and prevention of atomicity violations. In: Proceedings of the 5th European conference on Computer systems (Eurosys 2010), pp. 307–320. Paris, France (2010)
28. Apache-2.0.48, https://issues.apache.org/bugzilla/show_bug.cgi?id=25520. Accessed 1 May 2013 (2013)
29. Narayanasamy, S., Wang, Z., Tigani, J., Edwards, A., Calder, B.: Automatically classifying benign and harmful data races using replay analysis. In: 28th annual ACMSIGPLAN conference on Programming Language Design and Implementation (PLDI 2007), pp. 22–31 (2007)
30. Lucia, B., Devietti, J., Strauss, K., Ceze, L.: Atom-aid: Detecting and surviving atomicity violations. In: Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA 2008), pp. 277–288 (2008)
31. Woo, S.C., Ohara, M., Torrie, E., Singh, J.P., Gupta, A.: The SPLASH-2 programs: characterization and methodological considerations. In: Proceedings of the 22th Annual Interna-tional Symposium on Computer Architecture (ISCA 1995), pp. 24–36 (1995)
32. Xu, Z., Kai L., Xiaoping W., Xu L.: Exploiting parallelism in deterministic shared memory multiprocessing. J. Parallel Distrib. Comput. (JPDC), pp. 716–727 (2012)
33. Kai L., Xu, Z., Tom B., Xiaoping W.: Efficient Deterministic Multithreading Without Global Barriers. In PPoPP (2014)
34. Zhendong W., Kai L., Xiaoping W., Xu Z.: ColFinder: Collaborative Concurrency Bug Detection. The 13th International Conference on Quality Software (QSIC 2013), pp. 208–211 (2013)
35. Benjamin, W., David D., Peter M. C., Jason F. and Satish N.: Parallelizing Data Race Detection. In: 18th Architectural Support for Programming Languages and Operating Systems (ASPLOS 2013), pp. 27–38 (2013)
36. Sangmin, P., Mary J. H., Richard V.: Griffin: Grouping Suspicious Memory-Access Patterns to Improve Understanding of Concurrency Bugs. In: Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA 2013), pp. 134–144 (2013)
37. Madanlal, M., Shaz, Q., Thomas, B., Gerard, B., Piramanayagam, A. N., Iulian, N.: Finding and Reproducing Heisenbugs in Concurrent Programs. In: 8th USENIX symposium on Operating Systems Design and Implementation (OSDI 2008), pp. 267–280 (2008)
38. Sandeep, B., Sorav, B., Akash, L.: Variable and Thread Bounding for Systematic Testing of Multithreaded Programs. In: Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA 2013)
39. Jeff, H., Charles, Z., Julian, D.: CLAP: Recording Local Executions to Reproduce Concurrency Failures. In: 34th annual ACMSIGPLAN conference on Program-ming Language Design and Implementation (PLDI 2013), Seattle, WA, USA (2013)
40. Baris, K., Cristian Z., George C.: RaceMob: Crowdsourced Data Race Detection. In: Proceedings of the 23th ACM symposium on Operating systems principles (SOSP 2013)