Maple: A Coverage-Driven Testing Tool for Multithreaded Programs

Jie Yu Satish Narayanasamy
University of Michigan
{jieyu, natish}@umich.edu

Cristiano Pereira Gilles Pokam
Intel Corporation
{cristiano.l.pereira, gilles.a.pokam}@intel.com

Abstract

Testing multithreaded programs is a hard problem, because it is challenging to expose those rare interleavings that can trigger a concurrency bug. We propose a new thread interleaving coverage-driven testing tool called Maple that seeks to expose untested thread interleavings as much as possible. It memoizes tested interleavings and actively seeks to expose untested interleavings for a given test input to increase interleaving coverage.

We discuss several solutions to realize the above goal. First, we discuss a coverage metric based on a set of interleaving idioms. Second, we discuss an online technique to predict untested interleavings that can potentially be exposed for a given test input. Finally, the predicted untested interleavings are exposed by actively controlling the thread schedule while executing for the test input. We discuss our experiences in using the tool to expose several known and unknown bugs in real-world applications such as Apache and MySQL.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging

General Terms Design, Reliability

Keywords Testing, Debugging, Concurrency, Coverage, Idioms

1. Introduction

Testing a shared-memory multi-thread program and exposing concurrency bugs is a hard problem. For most concurrency bugs, the thread interleavings that can expose them manifest only rarely during an unperturbed execution. Even if a programmer manages to construct a test input that 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.

One common practice for exposing concurrency bugs is *stress-testing*, where a parallel program is subjected to extreme scenarios during a test run. This method is clearly inadequate, because naively executing a program again and again over an input tends to unnecessarily test similar thread interleavings and has less likelihood of exposing a rare buggy interleaving. An alternative to stress testing is *systematic testing* [13], where the thread scheduler systematically explores all legal thread interleavings for a given test input. Though the number of thread schedules could be reduced by using partial-order reduction [10, 12] and by bounding the number of context-switches [32], this approach does not scale well for long running programs.

Another recent development is *active testing* [36, 38, 50]. Active testing tools use approximate bug detectors such as static data-race detectors [7, 43] to predict buggy thread interleavings. Using a test input, an active scheduler would try to excercise a suspected buggy thread interleaving in a real execution and produce a failed test run to validate that the suspected bug is a true positive. Active testing tools target specific bug types such as data-races [38] or atomicity violations [17, 23, 35, 36, 40], and therefore are not generic. For a given test input, after actively testing for all the predicted buggy thread interleavings, a programmer may not be able to determine whether she should continue testing other thread interleavings for the same input or proceed to test a different input.

In this paper, we propose a tool called Maple that employs a *coverage-driven* approach for testing multithreaded programs. An interleaving coverage-driven approach has the potential to find different types of concurrency bugs, and also provide a metric for the programmers to understand the quality of their tests. While previous studies have attempted to define coverage metrics for mulithreaded programs based on synchronization operations [2] and inter-thread memory dependencies [22, 24, 39], synergistic testing tools that can help programmers achieve higher coverage for those metrics have been lacking.

OOPSLA'12, October 19–26, 2012, Tucson, Arizona, USA.
Copyright © 2012 ACM 978-1-4503-1561-6/12/10. . . $10.00

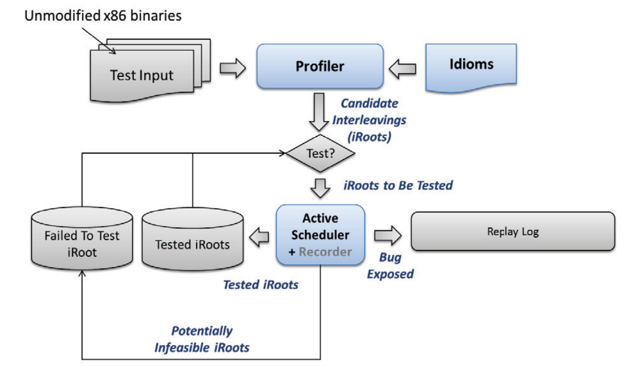


Figure 1. Overview of the framework.

The first contribution of this paper is the set of *interleaving idioms* which we use to define coverage for mulithreaded programs. An interleaving idiom is a pattern of inter-thread dependencies through shared-memory accesses. An instance of an interleaving idiom is called an iRoot which is represented using a set of static memory instructions. The goal of Maple is to expose as many new iRoots as possible during testing.

We define our set of interleaving idioms based on two hypothesis. One is the well-known small scope hypothesis [18, 29] and the other is what we refer to as the *value-independence hypothesis*. Small scope hypothesis [18, 29] states that most concurrency bugs can be exposed using a small number of preemptions. CHESS [32] exploits this observation to bound the number of preemptions to reduce the test space. We apply the same principle to bound the number of inter-thread memory dependencies in our interleaving patterns to two. Our empirical analysis of several concurrency bugs in real applications show that a majority of them can be triggered if at most two inter-thread memory dependencies are exposed in an execution.

Our value-independence hypothesis is that a majority of concurrency bugs gets triggered if the erroneous inter-thread memory dependencies are exposed, irrespective of the data values of the shared variables involved in the dependencies. We leverage this hypothesis to test for an iRoot only once, and avoid testing the same thread interleaving (iRoot) again and again across different test input. Thus, the number of thread interleavings to test would progressively reduce as we test for more inputs.

A critical challenge is in exposing untested iRoots for a given test input. To this end, we built the Maple testing infrastructure comprised of an online profiler and an active scheduler shown in Figure 1.

Maple's *online profiler* examines an execution for a test input, and predicts the set of candidate iRoots that are feasible for that input but have not yet been exposed in any prior test runs. Predicted untested iRoots are given as input to Maple's *active scheduler*. The active scheduler takes the test input and orchestrates the thread interleaving to realize the predicted iRoot in an actual execution using a set of novel heuristics. If the iRoot gets successfully exposed, then it is memoized by storing it in a database of iRoots tested for the program. We also consider the possibility that certain iRoots may never be feasible for any input. We progressively learn these iRoots and store them in a separate database. These iRoots are given a lower priority when there is only limited time available for testing.

When the active scheduler for an iRoot triggers a concurrency bug causing the program produces an incorrect result, Maple generates a bug report that contains the iRoot. Our active scheduler orchestrates thread schedules on a uniprocessor, and therefore recording the order of thread schedule along with other non-deterministic system input, if any, could allow a programmer to reproduce the failed execution exposed by Maple.

We envision two usage models for Maple. One usage scenario is when a programmer has a test input and wants to test her program with it. In this scenario, Maple will help the programmer actively expose thread interleavings that were not tested in the past. Also, a programmer can determine how long to test for an input, because Maple's predictor would produce a finite number of iRoots for testing.

Another usage scenario is when a programmer accidentally exposed a bug for some input, but is unable to reproduce the failed execution. A programmer could use Maple with the bug triggering input to quickly expose the buggy interleaving. We helped a developer at Intel in a similar situation to expose an unknown bug using Maple.

We built a dynamic analysis framework using PIN [28] for analyzing concurrent programs. Using this framework, we built several concurrency testing tools including Maple, a systematic testing tool called CHESS [32] and tools such as PCT [3] that rely on randomized thread schedulers, which we compare in our experiments.

We perform several experiments using open-source applications (Apache, MySQL, Memcached, etc.). Though Maple does not provide hard guarantees similar to CHESS [29] and PCT [3], it is effective in achieving higher iRoot coverage faster than those tools in practice. We also show that Maple is effective in exposing 13 documented bugs faster than these prior methods, which provides evidence that achieving higher coverage for our metric based on interleaving idioms is effective in exposing concurrency bugs. We also discuss our experiences in using Maple to find 3 unknown bugs in `aget`, `glibc`, and `CNC`.

Our dynamic analysis framework for concurrent programs and all the testing tools we developed are made available to the public under the Apache 2.0 license. They can be downloaded from (`https://github.com/jieyu/maple`).

2. Coverage-Driven Testing Based on Interleaving Idioms

In this section we discuss a coverage-driven testing methodology for multithreaded programs. For sequential programs, metrics such as program statement coverage are commonly used to understand the effectiveness of a test suite and determine if additional testing is required. For multithreaded programs, however, a practically useful thread interleaving coverage metric has been lacking.

We define coverage for multithreaded programs based on a set of thread interleaving idioms. Sections 3 and 4 discuss our Maple tool that can help programmers achieve higher coverage for these interleaving idioms.

2.1 Interleaving Idioms

An interleaving idiom is a pattern of inter-thread dependencies and the associated memory operations. An inter-thread memory dependency (denoted using $\Rightarrow$) is an immediate (read-write or write-write) dependency between two memory accesses in two threads. A memory access could be either to a data or a synchronization variable. A dynamic instance of an idiom in a program's execution is called as an *interleaving root* (iRoot). A memory access in an iRoot is represented using the static address of the memory instruction. *Coverage* of a test suite for a program is simply calculated as the number of iRoots exposed in any of the test run.

Interleaving idioms should be generic enough that, by exposing their iRoots, most concurrency bugs could be triggered. At the same time, the coverage domain (number of iRoots that needs to be tested) should be small enough that, the probability of exposing an unknown concurrency bug is high when an untested iRoot is exposed. To meet these competing demands, we make an assumption that most concurrency bugs can be exposed using simple thread interleaving patterns. This assumption is inspired by the small scope hypothesis [18, 29].

We study a set of canonical idioms that can be constructed for one or two inter-thread dependencies (which implies there can be only one or two shared-variables) involving no more than two threads. Figure 2 enumerates the canonical set of idioms for two inter-thread dependencies and two threads. There are six idioms in total. We refer to idiom1 as a simple idiom, and the rest as compound idioms. For compound idioms, to reduce the coverage domain without significantly compromising the ability to expose a concurrency bug, we include two additional constraints. First, the number of instructions executed between two events in the same thread should be less than a threshold. We refer to this threshold as the *vulnerability window (vw)*. Second, in an idiom, if atomicity of two memory accesses in a thread T to a variable V is violated by accesses in another thread, we disallow accesses to V between those two accesses in the thread T. For example, in idiom3 we do not allow any access to the variable X

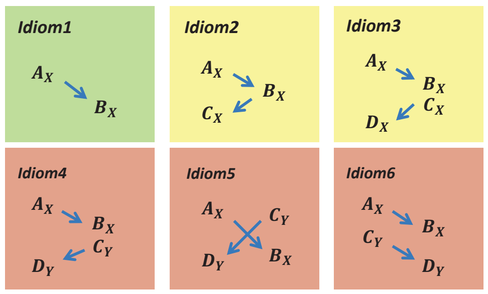
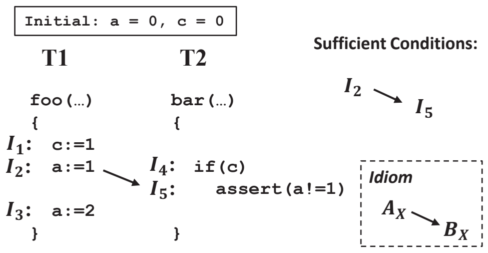


Figure 2. The canonical idioms for two inter-thread dependencies and two threads.


```
Initial: a = 0, c = 0

      T1                 T2
    foo(…)             bar(…)
    {                  {
I1:   c:=1
I2:   a:=1          I4:  if(c)
                    I5:    assert(a!=1)
I3:   a:=2
    }                  }
```

Figure 3. An idiom1 concurrency bug.

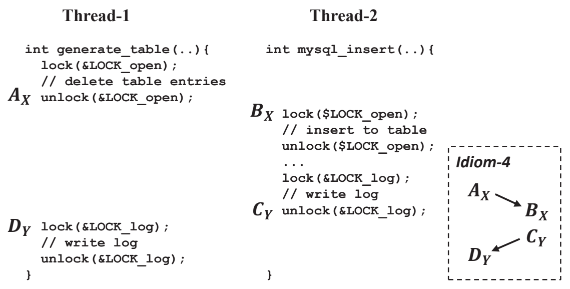
between the two memory accesses A_X and D_X, but there could be accesses to X between B_X and C_X.

Six idioms in Figure 2 can represent interleavings required to expose a majority of concurrency bugs: atomicity violations, including both single variable (idiom1, idiom2, idiom3) and multi-variable (idiom4, idiom5); typical deadlock bugs (idiom5), and generic order related concurrency bugs (idiom1, idiom6). These interleaving patterns are more general than the anomalous patterns used in prior studies to find specific classes of concurrency bugs [36, 38, 40].

2.2 Relation Between iRoots and Concurrency Bugs

The iRoot of a concurrency bug provides the *minimum* set of inter-thread dependencies and the associated memory or synchronization accesses, which if satisfied, can trigger that bug in an execution. Of course, higher order iRoots may also expose the same concurrency bug, but for the purpose of classifying concurrency bugs, we consider the iRoot that provides the *minimum* set of interleaving conditions.

Figure 3 shows an example of a concurrency bug. The idiom of the bug is shown in the dashed box. A and B represent static instructions in the program and X represents a memory location. The arrows denote inter-thread dependencies. The bug is triggered whenever the inter-thread dependency $I_2 \Rightarrow I_5$ is satisfied in an execution. Therefore, this is an idiom1 bug and its iRoot is $I_2 \Rightarrow I_5$. Note that there exists an inter-thread dependency $I_1 \Rightarrow I_4$ that must also be satisfied before the iRoot $I_2 \Rightarrow I_5$ can be exposed. This dependency affects the control flow of the thread $T2$ and determines whether $I5$ is executed or not. We refer to such conditions which must be satisfied in order to satisfy the idiom



```
Thread-1                                  Thread-2

  int generate_table(..){                 int mysql_insert(..){
    lock(&LOCK_open);
    // delete table entries
A_X unlock(&LOCK_open);
                                      B_X   lock($LOCK_open);
                                            // insert to table
                                            unlock($LOCK_open);
                                            ...
                                            lock(&LOCK_log);
                                            // write log
                                      C_Y   unlock(&LOCK_log);
D_Y lock(&LOCK_log);
    // write log
    unlock(&LOCK_log);
  }                                       }
```

Figure 4. A real idiom4 concurrency bug from MySQL.

Idiom1	Idiom2	Idiom3	Idiom4	Idiom5	Idiom6	Other
7	3	1	4	1	0	1

Table 1. Empirical study on 17 documented bugs.

conditions as *pre-conditions*. Also notice that $I_5 \Rightarrow I_3$ needs to be exposed before the bug can be triggered. However, this condition need not be part of the bug's iRoot ($I_2 \Rightarrow I_5$), because it is always implied by the bug's iRoot interleaving conditions.

Figure 4 shows a real concurrency bug in MySQL and its idiom. In this example, two critical sections in Thread-1 are expected to execute atomically, but the programmer did not enforce that constraint explicitly. The bug will be exposed when the critical sections in Thread-1 are intercepted with the critical section in Thread-2. The iRoot for this bug is of type idiom4 consisting of the two inter-thread dependencies between the lock and unlock operations. This example conveys an important observation that even if a concurrency bug is fairly complex involving many different variables and inter-thread dependencies, the iRoot of that bug (minimum set of interleaving conditions that need to be satisfied to trigger that bug) could be quite simple. Thus, by testing iRoots for a small set of idioms, we can hope to expose a significant fraction of concurrency bugs.

2.2.1 Empirical Analysis

To verify our hypothesis that achieving high coverage for our simple set of interleaving idioms could expose a significant fraction of concurrency bugs, we conducted an empirical study using 17 real world concurrency bugs from various programs including Apache, MySQL, and Memcached. Table 1 presents the results. Except one, the remaining 16 concurrency bugs can be characterized using one of our interleaving idioms. We could not represent one bug using any of our idioms (`Bug#10` in Table 2) because it was value dependent. We did not find any concurrency bug that can be classified as idiom6. Therefore, in this paper, we focus only on exposing iRoots for the first five idioms.

2.3 Coverage-Driven Testing Using Memoization

One of the disadvantages of random testing [3, 6] and systematic testing tools that expose all legal thread interleavings for a given test input [29] is that, they may expose the same thread interleaving (iRoot) again and again across different test inputs. In our coverage-driven testing method, we propose to memoize the thread interleavings (iRoots) exposed in a test run, and use it to target future tests to expose untested iRoots. In other words, if an iRoot has been already exposed during an earlier execution for some test input, Maple will not seek to expose the same iRoot again.

If a concurrency bug is dependent on values read or written by memory accesses, then exposing an iRoot once may not be enough. Extending interleaving idioms to include value conditions may address this problem. However, this would significantly enlarge the coverage domain and result in the same thread interleaving being unnecessarily tested multiple times for different values. We hypothesize that most concurrency bugs are such that whenever their erroneous interleavings are exposed they get triggered. We refer to this hypothesis as the value-independence hypothesis. Section 6 provides empirical evidence that supports this hypothesis. We also show that memoizing tested iRoots across different test inputs can drastically reduce testing time without significantly compromising Maple's ability in exposing concurrency bugs.

Maple seeks to achieve higher coverage by exposing as many different iRoots as possible during testing. Unlike coverage metrics such as basic block coverage, it is hard to estimate the total number of iRoots for a given program. However, number of exposed iRoots can be used as coverage metric for a saturation-based test adequacy [22, 39]. That is, a programmer can decide to stop testing at a point when additional tests are unlikely to expose new iRoots. We believe saturation-based testing approach is a practical solution for problems such as concurrent testing where estimating the coverage domain is intractable.

3. Online Profiling For Predicting iRoots

In this section, we discuss the design and implementation of Maple's *online* profiler. Given a program and a test input, the profiler predicts a set of candidate iRoots that can be tested for the given test input.

3.1 Notations and Terminology

As we discussed in Section 2.1, an iRoot for an idiom comprises of a set of inter-thread dependencies between memory accesses. A memory access could be either a data access or a synchronization access. For synchronization accesses, we only consider lock and unlock operations in this paper. A lock or an unlock operation is treated as a single access when we construct an iRoot, and all memory accesses executed within lock and unlock functions are ignored.

T_i (where i = 1,2,3,...) uniquely identifies a thread. A^i_X represents a dynamic memory access. The super script i uniquely identifies a dynamic access, but we usually omit it in our discussion to improve readability. If A^i_X is a data access, A stands for the static address of the memory instruc-

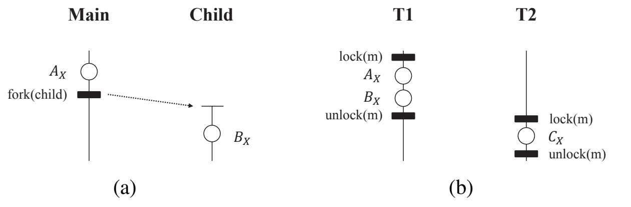


Figure 5. (a) Infeasible iRoots due to non-mutex happens-before relations. (b) Infeasible iRoots due to mutual exclusion.

tion and X refers to the memory location accessed. Two data accesses are said to conflict if both access the same memory location and at least one of them is a write. If A^i_X is a lock/unlock synchronization access, A stands for the address of the instruction that invoked the lock/unlock operation and X refers to the lock variable. Two synchronizations accesses conflict if one of them is a lock and the other is an unlock operation.

3.2 Naive Approach

We start with the simpler problem, which is predicting idiom1 iRoots for a given test input. One naive way to predict idiom1 iRoots is as follows. First, run the program once and obtain the set of accesses executed by each thread. Then, for any access A_X in thread T_c, if there exists a conflicting access B_X in another thread T_r, we predict two idiom1 iRoots: $A \Rightarrow B$ and $B \Rightarrow A$. For synchronization accesses, we predict $A \Rightarrow B$ only if A is an unlock operation and B is a lock operation. Obviously, this approach could produce many infeasible iRoots. an iRoot is said to be infeasible for a program if it can never manifest in any legal execution of that program. In the following sections, we discuss two major sources of inaccuracy in this naive algorithm and present our solutions.

3.3 Non-Mutex Happens Before Analysis

The profiler predicts iRoots based on a few profiled executions for a given test input. The predicted iRoots may not appear in any of the profiled executions, but they are predicted to be realizable in some other legal executions. We should avoid predicting iRoots that can never manifest in any of the legal executions.

We observe that some of the happens-before relations tend to remain the same in all of the legal executions. Therefore, these happens-before relations can be used to filter out infeasible iRoots predicted in the naive approach. For example, in Figure 5(a), an access A_X is executed before the main thread forks the child thread. B_X is executed in the child thread. Assuming that A_X and B_X are conflicting, the naive approach will predict two idiom1 iRoots: $A \Rightarrow B$ and $B \Rightarrow A$. However, it is trivial to observe that in any legal execution, A_X executes before B_X because of the fork call. As a result, we should not predict the iRoot $B \Rightarrow A$ as a candidate to test.

We improve the accuracy of our profiler by exploiting the observation that *non-mutex happens-before* relations mostly remain the same across different executions for a given input. A non-mutex happens-before relation is due to any synchronization operation other than a lock/unlock. Happens-before relations due to locks tend to change across executions, because the order in which locks are acquired could easily change. On the contrary, we find that non-mutex happens-before relations (e.g. fork-join, barrier and signal-wait) are more likely to remain constant across different executions. Therefore, the profiler predicts an iRoot only if it does not violate the non-mutex happens-before relations in at least one of the profiled executions. For the program in Figure 5(a), B_X cannot happen before A_X any of the executions according to the non-mutex happens-before relation due to fork. As a result, the profiler will not predict $B \Rightarrow A$ as a candidate iRoot to test. Though effective in pruning infeasible iRoots, this analysis is not sound because some non-mutex happens-before relations are not guaranteed to remain constant across different executions for an input.

3.4 Mutual Exclusion Analysis

Mutual exclusion constraints imposed by locks could also prevent naively predicted iRoots from manifesting in any of the alternate executions. For example, in Figure 5(b), all the accesses (A_X, B_X and C_X) are protected by the same lock m. Assume that these accesses conflict with each other. The naive approach would predict $A \Rightarrow C$ (and $C \Rightarrow B$) to be a candidate iRoot to test. Clearly, $A \Rightarrow C$ (and $C \Rightarrow B$) is not feasible because of the mutual exclusion constraint imposed by the lock m.

To further improve its accuracy, the profiler is augmented with a *mutual exclusion analysis* phase to filter those infeasible iRoots that are caused by the mutual exclusion constraints. To achieve this, the profiler needs to collect two types of information for each access A_X. One is the lockset information which contains the set of locks that are held by $Thd(A_X)$ when executing A_X. The other is the critical section information which specifies whether A_X is the first or the last access to X in the critical section that contain A_X.

We now use an example to illustrate how these two types of information can be used to filter infeasible iRoots caused by the mutual exclusion constraints. Consider the example in Figure 5(b). The profiler needs to decide whether iRoot $A \Rightarrow C$ is feasible. It first checks the locksets of both accesses: A_X and C_X. If the locksets are disjoint, the profiler will immediately predict the iRoot to be feasible. If not, the profiler will go to the next step. In this example, A_X and C_X have the same lockset $\{m\}$. Therefore, the profiler proceeds to the next step. In the second step, for each common lock (in our example its m), the profiler checks whether the mutual exclusion constraint imposed by the common lock will prevent the iRoot from manifesting. It checks whether A_X is the last access to X in the critical section that is guarded by the common lock m, and whether C_X is the first access to X in the critical section that is guarded by the common lock m. If either of them is not true, the profiler will predict that the iRoot is infeasible. In our example, since B_X is the last access to X in the critical section that is guarded by the common lock m, the iRoot $A \Rightarrow C$ is predicted to be infeasible. This analysis is also not sound since control flow differences between executions could affect our analysis, but it works well in practice.

3.5 Online Profiling Algorithm

Our profiler predicts candidate iRoots to test for a particular idiom using an online mechanism that we describe in detail in this section. An online algorithm avoids the need to collect large traces.

3.5.1 Baseline Algorithm

The profiler monitors every memory access. For each object, the profiler maintains an *access history* for each thread. We use $AH_X(T_i)$ to denote the access history for object X and thread T_i. Each access A_X in the access history $AH_X(T_i)$ is ordered by the execution order of T_i, and is associated with a *vector clock* and an *annotated lockset*. The vector clock, denoted as $VC(A_X)$, is used to perform the non-mutex happens-before analysis. It is the same as that used in many of the dynamic data race detectors, except that here we consider non-mutex happens-before relations. The annotated lockset, denoted as $AnnoLS(A_X)$, is used to perform the mutual exclusion analysis. It consists of a set of locks, each of which is annotated with a sequence number and two bits. The sequence number is used to uniquely identify each critical section guarded by the lock, and the two bits indicate whether the access is the first or the last access in the corresponding critical section. We say that two annotated locksets are disjoint if no common lock is found between the two sets. Both the vector clock and the annotated lockset are recorded when $Thd(A_X)$ is executing A_X.

When an access A_X is being executed by T_c, the profiler checks the access histories from all other threads on object X (i.e. $AH_X(T_r), T_r \neq T_c$). If there exists a conflicting access B_X in $AH_X(T_r)$, the profiler will predict the iRoot $B \Rightarrow A$ if the following conditions are true: (1) B_X does not happen after A_X by checking $VC(B_X)$ and $VC(A_X)$ (*the non-mutex happens-before check*). (2) Either $AnnoLS(A_X)$ and $AnnoLS(B_X)$ are disjoint, or for each common lock m held by A_X and B_X, A_X is the first access to X in the corresponding critical section guarded by m and B_X is the last access to X in the corresponding critical section guarded by m (*the mutual exclusion check*). Similarly, the profiler will also predict the iRoot $A \Rightarrow B$ according to the above rules.

To make the profiling algorithm online, we need to deal with several issues. One issue is that when A_X executes, some access, say C_X, has not been performed yet. As a result, C_X will not be in any access history. However, the profiler will still correctly predict iRoot $A \Rightarrow C$ and iRoot $C \Rightarrow A$ at the time C_X is executed if they are feasible. Another issue with the online algorithm is that when executes A_X, the profiler cannot precisely compute the annotated lockset $AnnoLS(A_X)$ required by the mutual exclusion analysis. The reason is because it does not know whether the access A_X will be the last access in the current critical section or not. We solve this issue by delaying predicting iRoots for A_X until either of the following events happens: (1) another access to X is reached by $Thd(A_X)$. (2) X is about to be deallocated (e.g. free()). (3) $Thd(A_X)$ is about to exit. The insight here is that the profiler can precisely compute the annotated lockset for A_X if any of the above events happens.

3.5.2 Optimizations

We have designed a few optimizations to make the online algorithm practical for large applications.

(1) Condensing access histories. We find that it is not necessary to store two accesses in an access history if these two accesses are *identical*. Two accesses from the same thread are said to be identical if they are originated from the same static instruction, and have the same vector clock and annotated lockset. The reason is because iRoots only consider static instructions (rather than dynamic events). Therefore, an access will not be added to an access history if an identical access in the access history can be found. This optimization is sound and can not only save space, but save time as well since each time when predicting iRoots, the number of accesses in the access histories that need to be examined reduces.

(2) Caching prediction results. To predict iRoots for an access A_X, the profiler needs to scan the access histories from all other threads on object X, which could be a time consuming operation. We call this operation a *full scan*. We observe that it is not always needed to perform a full scan. If A_X does not cause the access history to be updated (i.e. it can be condensed according to the previous optimization), the profiler can safely skip the full scan as no new iRoot will be predicted even if a full scan is performed.

(3) Removing useless access history entries. We observe that it is not necessary to keep all access histories from the beginning of the program execution. If an access in the access history can no longer be part of any potentially predicted iRoot, we can safely remove it from the access history.

(4) Monitoring only shared instructions. Maintaining access histories for each memory location is very expensive. Clearly, the profiler does not need to maintain access histories for thread private locations. We perform an online analysis that runs concurrently with the profiler and detects the instructions that can access shared locations. The profiler uses this information to create access histories only for the locations accessed by these shared instructions. We omit the

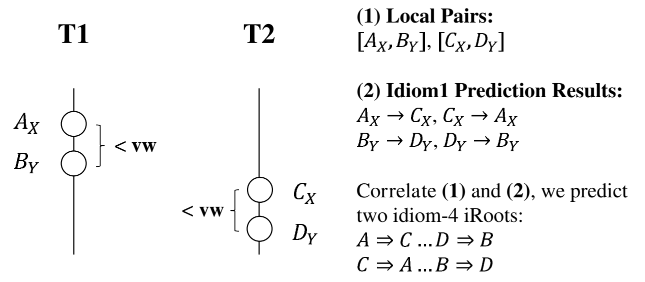

T1
T2
A_X
B_Y
< vw
< vw
C_X
D_Y
(1) Local Pairs:
$[A_X, B_Y]$, $[C_X, D_Y]$
(2) Idiom1 Prediction Results:
$A_X \rightarrow C_X$, $C_X \rightarrow A_X$
$B_Y \rightarrow D_Y$, $D_Y \rightarrow B_Y$
Correlate (1) and (2), we predict two idiom-4 iRoots:
$A \Rightarrow C \ldots D \Rightarrow B$
$C \Rightarrow A \ldots B \Rightarrow D$

Figure 6. Predicting iRoots for compound idioms.

details of the shared instruction analysis due to space constraint.

3.6 Predicting iRoots for Compound Idioms

To predict iRoots for compound idioms, we designed an algorithm that leverages the idiom1 prediction results. The approach is generic to all compound idioms defined in Section 2. The algorithm is divided into two parts: identifying local pairs, and correlating with idiom1 prediction results. In this section, we discuss these two parts in detail.

3.6.1 Identifying Local Pairs

A local pair, as suggested by its name, is a pair of accesses from the same thread. During a profiling execution, if the profiler finds two accesses A_X and B_Y (X may or may equal to Y) such that A_X and B_Y are from the same thread, A_X is executed before B_Y, and the number of dynamic instructions executed between A_X and B_Y is less than a pre-set threshold vw (vw stands for vulnerability window and is specified in the idiom definition), it will record a local pair $[A_X, B_Y]$. For example, Figure 6 shows a profiling execution. Accesses A_X and B_Y in T_1 are executed first, followed by C_X and D_Y in T_2. The profiler records two local pairs from this profiling execution: $[A_X, B_Y]$ and $[C_X, D_Y]$. To collect local pairs, the profiler uses a rolling window for each thread to keep track of the recent accesses.

3.6.2 Correlating with Idiom1 Prediction Results

To predict iRoots for compound idioms, we propose to leverage the idiom1 prediction results. We use an example to illustrate how to correlate local pairs with idiom1 prediction results to predict compound idiom iRoots. Consider the example shown in Figure 6. As mentioned, the profiler identifies two local pairs: $[A_X, B_Y]$ and $[C_X, D_Y]$. Meanwhile, the profiler also records the idiom1 prediction results. For instance, A_X and C_X can produce two idiom1 iRoots $A \Rightarrow C$ and $C \Rightarrow A$ according to the idiom1 prediction algorithm, therefore the profiler records both $A_X \rightarrow C_X$ and $C_X \rightarrow A_X$ in the idiom1 prediction results [1]. Similarly, the profiler records $B_Y \rightarrow D_Y$ and $D_Y \rightarrow B_Y$. Now, consider the first local pair $[A_X, B_Y]$. According to the predicted idiom1 results, C_X can potentially depend on A_X, and B_Y can potentially depends on D_Y. As a result, the profiler predicts an idiom4 iRoot $A \Rightarrow C...D \Rightarrow B$ (assume $X \neq Y$). Similarly, for another local pair $[C_X, D_Y]$, the profiler predicts another idiom4 iRoot $C \Rightarrow A...B \Rightarrow D$. Currently, the profiler performs the correlation part at the end of each profiling execution. Similar optimization technique is used to condense local pairs, that is if two local pairs from the same thread have both their accesses identical, the profiler just records one of them.

[1] Notice that the idiom1 prediction results are only useful for the current profiling execution, and will be discarded once the execution finishes. They are different from the predicted idiom1 iRoots which last across executions. They contain more information than idiom1 iRoots do.

4. Actively Testing Predicted iRoots

In this section, we discuss the design and implementation of Maple's active scheduler. Maple's profiler predicts a set of iRoots that can be realized in an execution using a test input. The goal of Maple's active scheduler is to validate the prediction by orchestrating the thread schedule to realize the predicted iRoots in an actual execution for the test input.

4.1 A Naive Approach

Suppose that we want to expose an idiom1 candidate iRoot $A \Rightarrow B$. The static instructions A and B are called *candidate instructions*. In a test run, there might be multiple dynamic accesses associated with a single candidate instruction. We still use A_X to denote a dynamic accesses to object X by the candidate instruction A. The naive approach works as follows. Whenever a candidate instruction (say A_X) is reached by a thread (say T_1), the active scheduler delays the execution of T_1. During the delay, if another thread (say T_2) reaches the other candidate instruction (say B_X), then the iRoot $A \Rightarrow B$ is exposed by executing A_X first and then executing B_X (as shown in Figure 7(a)).

This approach is used in several prior studies (e.g. [36]). While it is simple, it can lead to several issues, including deadlocks (also referred as thrashing in [20]). Consider the example in Figure 7(b). Suppose that T_1 reaches A_X first. The active scheduler, in this case, delays the execution of T_1, waiting for the other candidate instruction to be reached in T_2. T_2 is blocked when calling the barrier function, leading to a deadlock because no thread can make forward progress at that state. One way to mitigate this issue is to make use of timeout. In the example, if a timeout is introduced for each delay, T_1 will eventually be woken up when the timeout has expired. However, as discussed in the following sections, this is not enough to address most of the issues.

4.2 Non-preemptive and Strict Priority Scheduler

There are two problems with a timeout-based approach. First, it is sensitive to the underlying environment, hence fragile [19]. For instance, the timeout should be set to a larger value when running the program on a slower machine. Second, determining how long the timeout should be is not straightforward. A large timeout is detrimental to per-

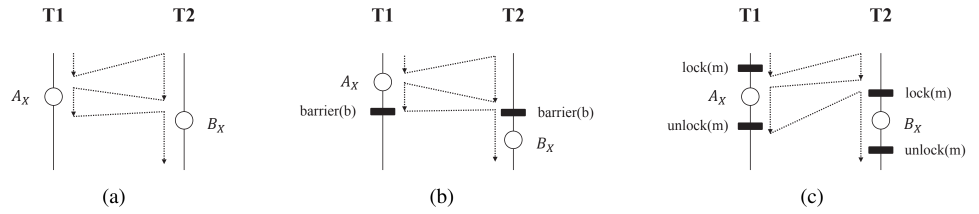


Figure 7. (a) The ideal situation for exposing an idiom1 iRoot $A \Rightarrow B$. (b) The naive approach could deadlock when exposing an idiom1 iRoot $A \Rightarrow B$. (c) The situation in which the watch mode is turned on for exposing an idiom1 iRoot $A \Rightarrow B$.

formance due to the longer delays, while a shorter timeout could cause unnecessary give ups.

An alternative to timeout is to monitor the status of each thread (blocked or not) by instrumenting every synchronization operations and blocking system calls (e.g. [20, 35, 38]). For example, in Figure 7(b), if the active scheduler keeps track of the status of each thread, it should know that T_2 is blocked after it calls the barrier function. Thus, T_1 will be woken up immediately since no other thread in the system can make forward progress at that state.

Our approach eliminates the need for monitoring thread status. The main idea is to leverage the non-preemptive and strict priorities provided by the underlying operating system (OS). All threads are forced to run on a single processor. Each thread is assigned a non-preemptive strict priority. Under this scenario, a lower priority thread never gets executed if there exists a non-blocked higher priority thread. In Linux, the real-time priorities are actually non-preemptive strict priorities [2]. By using non-preemptive strict priorities, the deadlocks will be automatically detected and resolved by the underlying OS since it knows the status of each thread. Let us consider the example in Figure 7(b). Initially, T_1 has a priority $P_{init}(T_1)$ and T_2 has a priority $P_{init}(T_2)$. Suppose that $P_{init}(T_1) > P_{init}(T_2)$. Therefore, T_1 executes first. When T_1 reaches A_X, the active scheduler changes the priority of T_1 to P_{low} such that $P_{low} < P_{init}(T_2)$. Due to the nature of the non-preemptive strict priorities, T_1 is preempted by T_2 immediately after the priority change. When T_2 calls the barrier function, it is blocked. At this moment, T_1 becomes the only non-blocked thread, and resumes execution immediately after T_2 is blocked. The deadlock situation is naturally resolved. Note that the active scheduler only needs to monitor the instructions involved in the iRoot being exposed, thus limiting the runtime overhead.

4.3 Complementary Schedules

Another problem with the approach discussed in Section 4.1 is that it does not have a mechanism to control the order in which threads get executed. Assume that we want to expose the idiom1 iRoot $A \Rightarrow B$ in the example of Figure 7(c), where A_X is in T_1 and B_X in T_2, respectively. Because both A_X and B_X are protected by the same lock m, if B_X is reached by T_2 first, the iRoot will not be exposed. The delay introduced before B_X will not help because T_1 will never be able to reach A_X due to the fact that T_2 is still holding the lock m. In order to expose this iRoot, A_X must be reached by T_1 first. However, the naive approach (Section 4.1) cannot guarantee this as it does not have a mechanism to control the order in which the threads get executed.

We address this issue using a technique called *complementary schedules*. The idea is to use two test runs on each candidate iRoot. Each newly created thread T_i, including the main thread, is assigned with an initial priority $P_{init}(T_i)$. In the first test run, the initial priorities are assigned from high to low, following the order of thread creation. To be more precise, we have $P_{init}(T_i) > P_{init}(T_j)$ (T_i has a higher priority than T_j) if thread T_i is created earlier than T_j. In the second test run, initial priorities are assigned from low to high, following the order of threads creation. In order words, $P_{init}(T_i) < P_{init}(T_j)$ if thread T_i is created earlier than T_j. Using this technique, we increase the likelihood that, in one of the two test runs, A_X will be reached first in the example shown in Figure 7(c).

4.4 Watch Mode Optimization

A problem with the naive approach is that it can unnecessarily give up exposing some iRoot in certain cases. Consider the example in Figure 7(c). We are still interested in exposing the idiom1 iRoot $A \Rightarrow B$. If T_1 reaches A_X first, the naive approach gives up exposing the iRoot for A_X right after the timeout. However, giving up is not necessary here because it is still possible that B_X could execute later without any access to X in between A_X and B_X.

We use a mechanism called *watch mode* to exposes an iRoot in such case. In watch mode, every memory access is monitored. Consider again the example in Figure 7(c). When T_1 reaches A_X first and sets its priority to P_{low} (P_{low} is lower than any initially assigned priorities), T_2 gets control and executes, but is blocked when trying to acquire the lock m. As mentioned above, T_1 resumes immediately after T_2 is blocked and executes A_X. At this moment, instead of giving up exposing iRoot for A_X, the active scheduler enters the watch mode and monitor every memory access. The active scheduler still keeps the priority of T_1 to P_{low}. Once the lock m is released, T_1 is preempted by T_2 because T_2 has a higher priority than T_1. Shortly after, T_2 reaches B_X and no access

[2] More specifically, schedule policy SCHED_FIFO.

to X is found in between A_X and B_X. Therefore, the iRoot $A \Rightarrow B$ is exposed.

In the same example, during the watch mode, it is likely that T_1 reaches an instruction – no matter whether it is a candidate instruction or not – that accesses X as A_X does. In such case, the active scheduler is not able to expose iRoot for A_X because T_1 already has the lowest priority at that moment. It just ends the watch mode and gives up exposing iRoot for A_X. If the access to X (not instruction B) is from a thread other than T_1 (say T_3), the active scheduler sets the priority of T_3 to P_{low}. The intuition here is that some other threads may make progress and reach the other candidate instruction B. However, if the conflicting access is eventually executed by T_3 in spite of its lowest priority, the active scheduler ends the watch mode and gives up.

The watch mode can be implemented efficiently using debug registers or by leveraging the selective instrumentation mechanism in PIN [28]. We implement the second approach. For compound idioms, the execution under watch mode is usually short given that we have distance constraints in the idiom definitions. For idiom1, the selective instrumentation mechanism in PIN can affect performance depending on how long the active scheduler spends in watch mode. The overhead is discussed in Section 6.

4.5 Candidate Arbitration

There might exist multiple dynamic accesses that correspond to the same candidate instruction during the execution. In many cases, the active scheduler has to decide which of these accesses belongs to the candidate iRoot to expose. For example, while the active scheduler is exposing iRoot for A_X (seeking candidate instruction B in other threads), it is possible that another thread also reaches the candidate instruction A, or another thread reaches candidate instruction B, but it happens to access a location other than X (say B_Y). In either one of these situations, the active scheduler has two choices: either continue to expose the iRoot $A \Rightarrow B$ for A_X, or give up on that attempt and seek to expose the iRoot for latter access of A or for B_Y. We decided to make these choices random with equal probability. We choose not to use a fixed policy because it could cause some feasible iRoot to become impossible to expose. We save the random seed for reproduction purpose.

We aware that the random arbitration algorithm we use may cause a later access exponentially unlikely to be used as part of a candidate iRoot. This will become an issue when we do want to expose an iRoot for a later access (e.g. only the iRoot that uses this access will lead to a bug). Nevertheless, we are still able to expose all the bugs we have analyzed using this strategy (Section 6.2). This may be because many of the bugs we analyzed manifest early in their executions, or not many dynamic accesses exist for the candidate instructions in the iRoots that expose the bugs. Even if that, we believe this is an important problem and we plan to address it in our future work. Currently, we can think of two possible ways. First, we can associate more information with each candidate instruction such as its calling context so that some irrelevant accesses that use the same instruction but different contexts will be filtered out. Second, we can devise a more sophisticated arbitration algorithm using more test runs.

4.6 Dealing with Asynchronous External Events

Some applications depend on asynchronous external events such as network and asynchronous signals. These events are usually difficult to deal with in the active scheduler because it has no control on when these events are delivered. Consider the example in Figure 8 where T_2 has a higher priority initially. When T_2 reaches B_X, its priority is changed to P_{low}, at which point T_1 is scheduled. If T_1 is blocked when calling the function `sigwait` (e.g. because the signal might not have been delivered yet), since all threads except T_2 are blocked in the system at that time, T_2 has to execute B_X in spite of its lowest priority; thus giving up the exposition of iRoot $A \Rightarrow B$ for B_X. We observe that if the asynchronous signal in this example is delivered earlier, the active scheduler might be able to expose the iRoot.

To solve this problem, the active scheduler introduces extra time delay where it is about to give up, hoping that the potential external event will arrive during that period. For instance, in the example of Figure 8, when the active scheduler is about to give up by executing B_X in T_2 after T_1 has been blocked, a time delay is injected right before B_X is executed. During this period, if the asynchronous signal is delivered, the active scheduler can successfully expose the iRoot $A \Rightarrow B$.

For applications that do not depend on asynchronous external events, there is no need for the active scheduler to inject extra time delay. We detect whether an application depends on asynchronous external events by monitoring system calls and signals during profiling. Even if an application does depend on asynchronous external events, this might not be true for all the iRoots. During profiling, we mark each candidate iRoot with a flag indicating whether this iRoot depends on asynchronous external events or not. The active scheduler uses this flag to decide whether to inject time delay or not. Finally, to ensure forward progress, we set a timeout for each delay. The timeout value can be optimized according to the application and the input.

4.7 Compound Idioms

A compound idiom iRoot is composed of multiple idiom1 iRoots. Our general policy for exposing compound idiom iRoots is to expose each of the idiom1 iRoot one at a time. Each of the idiom1 iRoot is exposed as described before, but the algorithm for exposing compound idiom iRoots needs to address two more issues that we describe next.

First, the active scheduler always enters the watch mode after the first candidate instruction in a compound iRoot is executed. To understand why, consider the example in

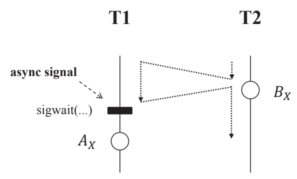


Figure 8. Problem with asynchronous external events.

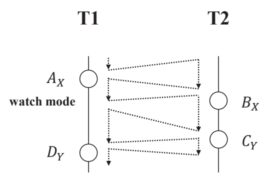


Figure 9. Expose a compound idiom iRoot $A \Rightarrow B ... C \Rightarrow D$.

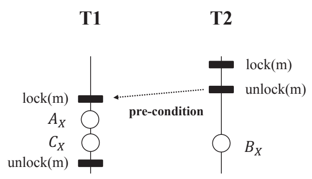


Figure 10. A pre-condition exists when trying to expose iRoot $A \Rightarrow B$.

Figure 9. The goal is to expose an idiom4 iRoot $A \Rightarrow B ... C \Rightarrow D$. According to the idiom4 definition, we need to make sure that there is no other access to the same locations that A and D access in between them. Therefore, after A_X is executed in this example, we enter the watch mode. If there is any access to location X before D_Y is reached, the active scheduler stops trying to expose the iRoot for A_X because this violates the idiom definition. One complex aspect of this implementation is that, before D_Y is reached, we do not know which location it is going to access. We solve this problem by recording the set of locations that are accessed by T_1 after A_X is executed. This set is checked when D_Y is reached, to verify that none of the addresses touched conflicts with the address accessed by D_Y. In addition, the active scheduler exits the watch mode and gives up exposing iRoot for A_X if the number of dynamic instructions executed by T_1 after A_X exceeds the pre-defined threshold specified in the idiom definition.

Second, the arbitration is biased after the first part of the compound idiom iRoot is exposed. In the example of Figure 9, after the first part of the iRoot ($A \Rightarrow B$) is exposed, if T_2 reaches A_Z (an access to Z by candidate instruction A), we have two choices: (1) ignore A_Z and continue looking for the second part of the iRoot; (2) expose the iRoot for A_Z and discard the already exposed first part. In such a case, we select the first choice with a higher probability.

Finally, exposing iRoots for idiom5 is slightly different from other compound idioms. To expose an idiom5 iRoot $A \Rightarrow B ... C \Rightarrow D$, our strategy is to let two different threads reach A and C, respectively; then execute them, and seek B and D in the corresponding threads.

4.8 Limitations

One limitation of the current active scheduler is that it cannot handle pre-conditions. The pre-conditions for an iRoot is the necessary conditions that need to satisfied to expose the iRoot. For example, in Figure 10, there exists a pre-condition (from `unlock` to `lock`) that needs to be satisfied so that the iRoot $A \Rightarrow B$ can be exposed. Currently, our active scheduler has no knowledge of these pre-conditions; therefore it cannot enforce them. The complementary schedules might alleviate this problem to some extent. To fully address this problem, one possible solution would be to automatically derive pre-conditions for a given iRoot [21]. We leave this as a future work.

5. Memoization of iRoots

Past work on systematic testing and active testing ignore the information about the interleavings tested from previous test runs with different inputs. We believe that this is crucial in reducing the number of interleavings that need to be tested for a given program input. Therefore, we propose a memoization module in our active testing infrastructure. The memoization module is composed of a database of tested interleavings and a database of fail-to-test interleavings for each interleaving idiom as shown in Figure 1. This module is used to avoid testing the same interleaving again and again across different test inputs.

The candidate interleavings are pruned out depending on whether previous attempts were made to test them. If a candidate interleaving was tested before (i.e. it has been exposed by the active scheduler), it is filtered out by consulting the tested interleavings database. This optimization is sound if the bugs we are targeting are not value dependent. Also, if several attempts were made in the past to test a candidate interleaving and the active testing system failed to produce a legal execution exposing the desired interleaving, this candidate interleaving is filtered out using the fail-to-test interleavings database, which stores all such failed to expose candidate interleavings. This allows us to avoid trying to expose thread interleavings that can never manifest. Unlike memoization of tested iRoots, this is an unsound optimization even if a bug is not value dependent. However, the number of times a candidate interleaving is actively tested is configurable, and a programmer can choose to set it to a very high value if soundness is a concern.

6. Evaluation

This section evaluates Maple and discusses our experiences in using the tool for testing real world applications. We first describe the configurations of Maple we used in our experiments (Section 6.1). Then, we compare Maple with other general concurrency testing techniques in two different usage scenarios (Section 6.2 and Section 6.3), and show how memoization can be useful in both of these scenarios (Section 6.2.2 and Section 6.3.2). Finally, we discuss the efficiency and effectiveness of Maple (Section 6.4).

6.1 Maple Configuration

Maple is built to be highly configurable. We now describe the default configurations of Maple in our experiments.

In the *Profiling Phase*, the program is profiled using the best randomized testing technique (explained later in Section 6.3.1) a few number of times. For each profile run, the profiler observes what iRoots are exposed and predicts candidate iRoots to test. The profiling phase stops when no new candidate iRoot is found for N consecutive profile runs (we use an empirical value $N = 3$ throughout our experiments). Unless otherwise noted, Maple observes and predicts all iRoots in the program by default, including those iRoots from libraries such as Glibc. We believe this is necessary because we do find bugs that are related to the library code (e.g. `Bug#11` and `Bug#13` in Table 2).

In the *Testing Phase*, candidate iRoots are exposed using the active scheduler. Currently, the active scheduler tests all candidate iRoots starting from idiom1 and then proceeds to test iRoots for other idioms in the order of their complexity (one to five). For each idiom, the active scheduler always chooses to test those iRoots that are from the application code first. More sophisticated ranking mechanism may exist, but we leave that to future work. Each candidate iRoot will be attempted at most twice, as mentioned in Section 4.3.

6.2 Usage Scenario 1: Exposing Bugs with Bug Triggering Inputs

One scenario that Maple can be useful is when a programmer or a user accidentally exposed a non-deterministic bug for some input, but is unable to reproduce the failed execution. In that case, the programmer can use Maple with the bug triggering input to quickly expose the buggy interleaving. Once the buggy interleaving is found, Maple can also reproduce it faithfully by replaying the same schedule choices, which can be very useful during the debugging process.

To evaluate the ability of Maple in exposing bugs in such scenarios, we choose 13 buggy application with their corresponding bug triggering inputs (shown in Table 2). Among these benchmarks, 4 (`Bug#1` to `Bug#4`) are synthetic bugs studied in [27], 1 (`Bug#5`) is a code snippet extracted from a real buggy program, and 8 (`Bug#6` to `Bug#13`) are real bugs from real executions.

We want to know whether Maple is able to expose these bugs and how fast it can expose these bugs when comparing to other general concurrency testing tools. We compare Maple with two random testing techniques, `PCT` and `PCTLarge`. `PCT` [3] is a recently proposed random testing technique that provides probabilistic guarantee in exposing concurrency bugs. In `PCT`, threads are randomly assigned a non-preemptive strict priority (similar to that used in the active scheduler of Maple); during execution, `PCT` changes the priority of the currently running thread to lowest at random points d times. The authors state that most of the concurrency bugs can be exposed with a small value of d. In our experiment, we choose to use $d = 3$. `PCTLarge` is a variation of `PCT` that we proposed. It has the same algorithm as that in `PCT` except that it uses non-strict priorities instead of strict priorities. For instance, in Linux, we use nice values to serve as non-strict priorities. Higher priority threads will have more time quantum than lower priority threads. Interestingly, we found that `PCTLarge` usually performs better than `PCT`. More details are provided in Section 6.3.1.

For each bug, we run it repeatedly using its bug triggering input until the bug is triggered. Each time, a different testing technique is used. We compare the time needed by each testing technique to expose the bug. For Maple, we assume *no* previously built memoization database is available. The effect of memoization is discussed in Section 6.2.2. Table 2 shows the results. As shown in the table, Maple can expose all 13 bugs, including 3 previously unknown bugs (`Bug#11` to `Bug#13`). In contrast, `PCT` and `PCTLarge` can only expose 7 and 11 bugs respectively before timeout (24 hours) in reached. Moreover, Maple can expose all the real bugs much faster than `PCT` and `PCTLarge`. Maple uses more time to expose `Bug#5` than `PCT` and `PCTLarge`. This is because `Bug#5` is an idiom4 bug and a lot of time is spent testing irrelevant idiom1, idiom2 and idiom3 iRoots according to our current ranking mechanism. We found that `PCT` or `PCTLarge` expose bugs faster than Maple on some applications with small execution lengths (e.g. `Bug#3`). This is expected because the smaller the execution length, the higher the probability to expose the bug, but Maple has to pay a high cost for analysis. Nevertheless, the random techniques do not scale for long execution lengths (e.g. `Bug#8`). `Bug#10` does not have an idiom because it is value dependent.

6.2.1 Experiences in Finding Unknown Bugs

We found three previously unknown bugs. `Bug#11` was accidentally found when testing `Bug#9`, a documented bug in Aget. We observed a situation where the program hangs when testing `Bug#9`. We looked at the iRoot that caused the hang and tried the same iRoot again with the same random seed. In addition, we attached a tracer to the active scheduler to record the execution trace. The deadlock happened again in less than 5 runs [3]. With the help of the trace, we eventually found the root cause of this bug. The thread that handles signals is asynchronously canceled when holding an i/o lock (in `printf`), causing a deadlock in the main thread when it tries to acquire the same lock.

`Bug#12` is an intermittent bug in an CNC-based application that manifests as an assertion failure. CNC was developed by Intel and stands for Concurrent Collections. The particular application we examined is a server-client application that builds on Intel Thread Building Blocks (TBB) to synchronize threads. This bug was provided to us by a developer at Intel who could not expose it even after attaching a software deterministic record and replay tool to it [37]. Maple was able to expose the assertion failure in about 400 test runs, much faster than the two random testing techniques. However, because we do not have access to the

[3] This is because we cannot faithfully replay some non-deterministic external events which are part of the program inputs.

ID	App	Type	Idiom	Maple					PCT [3]		PCTLarge		# NonStack Mem Ops	# Thds	Native Time
				# Profile	Profile Time	# Test	Test Time	Total Time	# Runs	Time	# Runs	Time			
1	LogProcSweep	S	Idiom1	11	16.5	1	0.6	**17.1**	98511	**86400(TO)**	10169	**8188.6**	3.3K	3	0.1
2	StringBuffer	S	Idiom1	8	12.0	1	0.8	**12.8**	40	**56.4**	61	**49.1**	2.4K	2	0.1
3	CircularList	S	Idiom3	6	9.5	1	1.0	**10.6**	6	**9.1**	18	**14.6**	3.3K	3	0.1
4	BankAccount	S	Idiom1	6	9.0	1	0.9	**10.0**	12	**17.4**	44	**35.4**	3.6K	3	0.1
5	MySQL-LogMiss	E	Idiom4	8	13.2	100	120.8	**133.9**	18	**29.0**	15	**13.6**	4.9K	3	0.1
6	Pbzip2	R-K	Idiom1	8	151.9	2	3.2	**155.1**	26933	**86400(TO)**	3336	**6144.1**	32.1M	3	0.1
7	Apache #25520	R-K	Idiom1	36	580.7	93	1544.2	**2124.9**	3485	**31688.0**	12951	**86400(TO)**	218.5K	5	3.6
8	MySQL #791	R-K	Idiom1	10	436.5	3975	43097.6	**43534.1**	11754	**86400(TO)**	10574	**81887.2**	1.8M	13	4.4
9	Aget #2	R-K	Idiom4	9	148.1	11	29.2	**177.4**	152	**355.0**	335	**619.5**	32.0K	3	0.1
10	Memcached #127	R-K	N/A	41	304.6	4	11.3	**316.0**	1010	**3635.1**	306	**782.5**	89.5K	4	1.2
11	Aget #1	R-U	Idiom1	9	74.7	18	123.9	**198.6**	32075	**86400(TO)**	47636	**86400(TO)**	529.5K	3	0.1
12	CNC	R-U	Idiom1*	6	50.6	403	4163.8	**4214.4**	11063	**86400(TO)**	10012	**49046.8**	209.6K	3	1.1
13	Glibc	R-U	Idiom1*	30	1120.4	20	36.6	**1157.0**	39560	**86400(TO)**	16147	**34349.1**	28.5M	4	0.1

Table 2. Bug exposing capability given bug triggering inputs. All the time reported in the table are in seconds. **TO** stands for timeout (24 hours). In the *type* column, **S** stands for synthetic bugs, **E** stands for extracted bugs, **R-K** stands for real bugs which are known, and **R-U** stands for real bugs which are unknown. * The root cause of Bug#12 and bug Bug#13 have not been confirmed yet. They are exposed when attempting idiom1 iRoots.

source code, we could not help the programmer understand the root cause of the bug using iRoot.

The Glibc bug (Bug#13) was also accidentally discovered when testing Bug#6 on a machine with glibc-2.5. It manifested as an assertion failure from the free function. We could reproduce the buggy interleaving using the same iRoot and the same random seed. The bug never showed up when a newer version of glibc was used. Since the memory management code in glibc is quite complex, the root cause has not been confirmed yet.

6.2.2 Memoization Help Expose Bugs Faster

We aware that applying memoization may affect the bug exposing capability of Maple. For example, if an iRoot cannot be exposed under some inputs, it does not mean that it is not feasible under other inputs. Since we put a limit on the total number of test runs on any iRoot in our current settings, the corresponding iRoot that leads to the bug might not be attempted when the bug triggering input is used, causing the bug to be missed. In order to see how memoization can affect the bug exposing capability, we evaluate 4 real bugs from Table 2 (Bug#7 to Bug#10). Other real bugs are not chosen either because the bugs can be exposed using any input (Bug#6, Bug#11 and Bug#13), or no other input is available (Bug#12). We first test the 4 bugs using inputs that do not trigger the bug to build the memoization databases. Then, we test the bugs using the bug triggering inputs. Table 3 shows the results. We can see that all the 4 bugs can be exposed when memoization is applied. More importantly, the time required to expose each bug also reduces drastically. For instance, we save about 94% of the testing time for Bug#8. In fact, for the server application bugs, we save can a lot of testing time by memoizing those iRoots that are related to server start and server shutdown, clearly showing the benefit of memoization.

ID	App.	Memo	# Profile	Profile Time	# Test	Test Time	Total Time
7	Apache #25520	No	36	580.7	93	1544.2	2124.9
		Yes	22	357.6	2	18.3	375.8
8	MySQL #791	No	10	436.5	3975	43097.6	43534.1
		Yes	8	362.9	162	1953.6	2316.5
9	Aget #2	No	9	148.1	11	29.2	177.4
		Yes	6	100.5	8	21.8	122.3
10	Memcached #127	No	41	304.6	4	11.3	316.0
		Yes	36	272.6	5	12.1	284.8

Table 3. Memoization help expose bugs more quickly. All the time reported in the table are in seconds.

6.3 Usage Scenario 2: Coverage-Driven Testing

Another usage scenario that Maple can be helpful is when a programmer has a test input and wants to explore as many interleavings as possible for that input within the time budget. In this scenario, Maple can be used to cover more interleavings quickly. Also, memoization can prevent the programmer from testing the same interleaving multiple times, which helps reduce testing time.

6.3.1 Maple Achieves Higher Coverage Faster

The first question we want to address is whether Maple can cover more interleavings faster than other testing techniques. To quantify the coverage on interleavings, we use *iRoot coverage* as the coverage metric in our experiments. The iRoot coverage is measured using a tuple of numbers, each of which is the number of exposed iRoots for one idiom. For example, the following iRoot coverage $(1, 2, 5, 100, 50)$ means that the test has successfully exposed 1 idiom1 iRoot, 2 idiom2 iRoots, 5 idiom3 iRoots, 100 idiom4 iRoots and 50 idiom5 iRoots. We have implemented a tool, called *observer*, in our dynamic analysis framework to measure the iRoot coverage. The same observer is also reused in the profiler to observe exposed iRoots during profile runs so as to avoid testing these iRoots again during the test phase.

We compare it with 4 other testing techniques: PCT, PCTLarge, RandDelay and CHESS. PCT and PCTLarge have already been introduced in Section 6.2. RandDelay injects random time delay at random points during the execution.

The number of points in which a delay is introduced is proportional to the execution length (one per 50K non stack memory accesses). The program is run on multi-core processors when `RandDelay` is used. `CHESS` [32] is a systematic testing tool. For a given program and a given input, it tries to explore all possible thread interleavings that have few preemptions. It was originally developed for Windows programs. We implemented it in our framework for Linux. Currently, it works for programs that use POSIX threads for synchronization. It employs the sleep-set based partial order reduction technique described in [30], and uses a fair scheduler discussed in [31]. We use a preemption bound of 2 throughout our experiments as suggested in [29] [4]. To handle a program that has data races, we run a dynamic data race detector first to find all racy memory accesses in the program, and then inform the `CHESS` scheduler so that it can explore different orderings of these racy memory accesses.

We use seven bug-free multi-threaded applications in this experiments, among which (`fft` and `radix`) are scientific applications from Splash2 [46], (`pfscan`, `pbzip2`, `aget`) are utility programs, and (`memcached` and `apache`) are server applications. For scientific and utility programs, we use random inputs (e.g. random number of thread, random files and directories, random URLs, etc.). For `memcached`, we wrote our own test cases which perform commonly used operations such as set/get keys and incr/decr keys. For `apache`, we use SURGE [1] to generate URLs and use `httperf` to generate parallel requests. Notice that when testing server applications, each test run consists of starting the server, issuing the requests, and stopping the server. This process is automated through scripting.

To compare Maple with these tools, we attach the same observer to each tool to collect the iRoot coverage after each test run. The current implementation of `CHESS` cannot identify the low level synchronization operations used in Glibc. Though we can treat those unrecognizable synchronization operations as racy memory accesses and still run `CHESS` on it, we believe this approach is unfair to `CHESS` as the number schedules to explore will increase unnecessarily comparing to the case in which we can recognize those synchronization operations. As a result, we decide to only consider iRoots from application code and ignore library code in this experiment to ensure a fair comparison.

Figure 11 shows the iRoot coverage achieved by these tools using the same amount of time as Maple does. We run Maple till its completion. For `apache`, as we are not able to run Maple till completion due to its scale, we test it for 6 hours. The observer overhead is excluded from the testing time. Y-axis is normalized to the iRoot coverage achieved by Maple. We are not able to run `CHESS` on `aget`, `memcached` and `apache` because in these applications, some non-deterministic events (e.g. network package arrival) are not controllable by `CHESS` [5]. From the results shown in Figure 11, we find that Maple gains iRoot coverage faster than all the tools we have analyzed. On average, it achieves about 15% more coverage than the second best tool in our experiment. Also, we find `CHESS` only achieves about 60% of the iRoot coverage achieved by Maple using the same amount of time as Maple does. Be aware that the results shown in Figure 11 do not mean that Maple is able to explore more interleavings than random testing tools and systematic testing tools. In fact, `CHESS` explores a different interleaving in each test run. The results shown here convey a message that if we believe iRoot coverage is a good coverage metric for concurrent testing, a specially engineered tool like Maple is able to achieve iRoot coverage faster than a more general testing tool such as `CHESS` and `PCT`.

We also notice that `PCTLarge` performs better than other random testing techniques like `PCT` and `RandDelay`. We believe the reason is because `PCTLarge` has more context switches than others. As a result, we choose to use `PCTLarge` to randomize the profile runs in Maple.

Figure 12 shows the rate of increase in iRoot coverage using different testing tools. The X-axis is the number of test runs and the Y-axis is the total number of iRoots exposed so far. We only show results for those applications on which we are able to run `CHESS`. We run `CHESS` till its completion in this experiment. The results in Figure 12 further justify the fact that Maple is able to gain iRoot coverage faster than random testing tools and systematic testing tools. Also, we notice an interesting fact that `CHESS` experiences a slow start in gaining iRoot coverage. We believe this is due to the use of depth-first search strategy in `CHESS`. A best-first search strategy may alleviate this problem at the cost of storing more states [5].

6.3.2 Memoization Help Reduce Testing Time

The next question we want to address is how much testing time we can save when memoization is applied under this usage scenario. To do that, for each bug free application, we test it using 8 different inputs ($input_i, i \in [1, 8]$). When testing with $input_{i+1}$, we compared the testing time between the following two methods: (1) without memoization database; (2) using the memoization database built from $input_1$ to $input_i$. We choose to memoize both the exposed iRoots and the fail-to-expose iRoots (we set the threshold to 6, i.e. each iRoot will not be attempted more than 6 test runs). For this experiment, we only test for idiom1 iRoots due to time constraints. Figure 13 shows the results. The Y-axis represents the testing time of the method that uses memoization (normalized to the testing time without memoization). The line plotted in red shows the average of the applications we analyzed. We observe that, with memoization, the testing time reduces gradually when more and more

[4] In fact, 13 out of 14 bugs studied in [29] are exposed with a preemption bound of 2. Using a preemption bound larger than 2 will drastically increase the number of test runs and exceed our time budget.

[5] Such programs are called non closed programs.

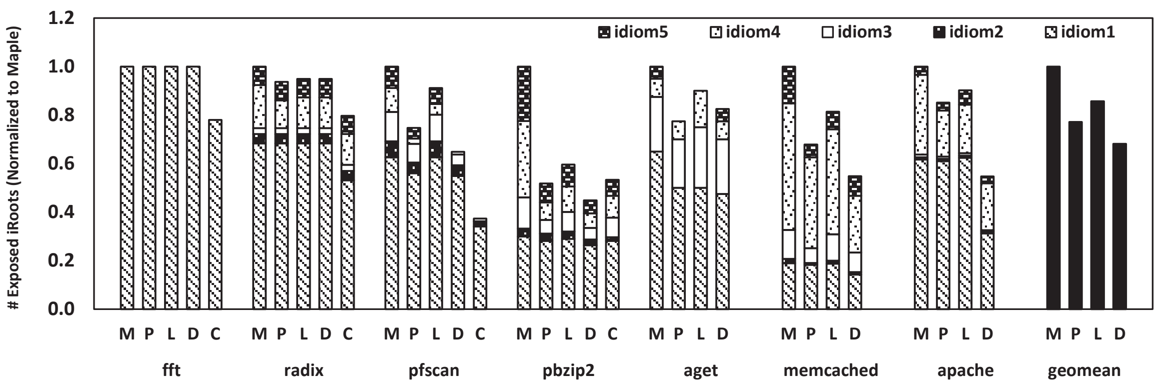


Figure 11. Comparison with different testing methods using the same amount of time. **M** stands for Maple, **P** stands for `PCT`, **L** stands for `PCTLarge`, **D** stands for `RandDelay`, and **C** stands for `CHESS`.

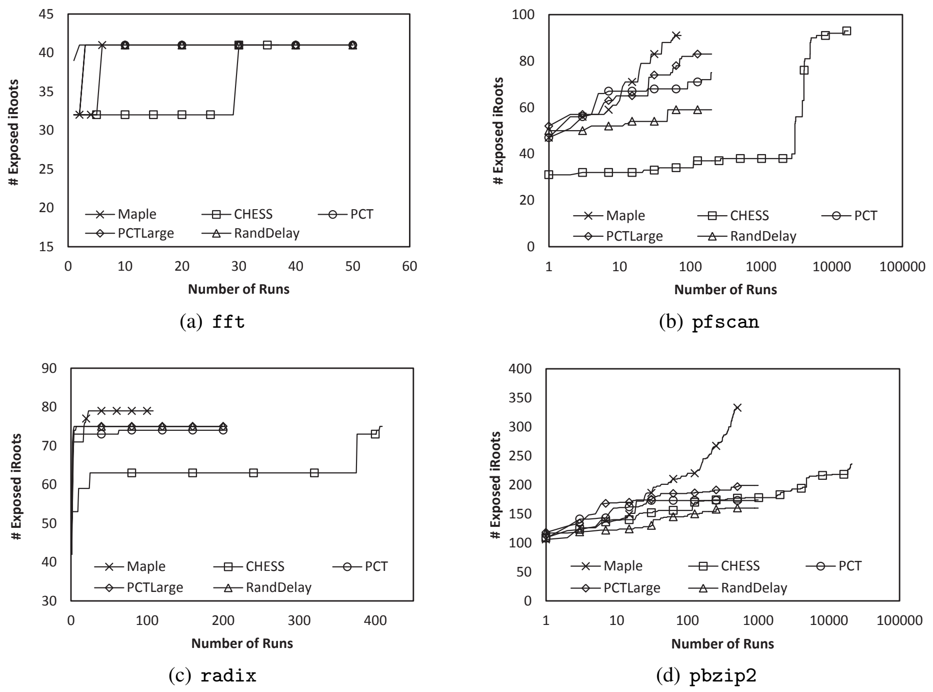


Figure 12. Comparison with different testing methods. X-axis is the number of test runs, and Y-axis is the total number of iRoots exposed.

inputs are tested. For $input_8$, the average saving on testing time is about 90%. This clearly shows the benefit of memoization in reducing testing time.

6.4 Characteristics of Maple

In the following, we discuss the characteristics of Maple in terms of its efficiency (Section 6.4.1) and effectiveness (Section 6.4.2).

6.4.1 Performance Overhead

Table 4 shows the average performance overhead of the profiler and the active scheduler for each application. We also include the Pinbase overhead, which is the overhead of PIN

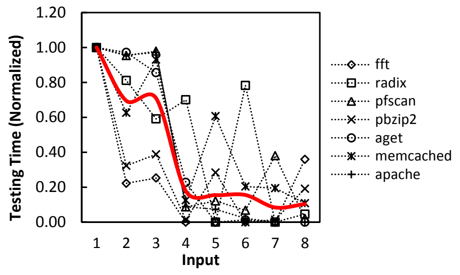


Figure 13. Memoization saves testing time. Y axis is normalized to the execution time without memoization.

App.	Pinbase	Profiler	Active Scheduler
fft	6.9X	30.9X	16.3X
radix	6.9X	67.7X	17.8X
pfscan	8.3X	31.9X	27.7X
pbzip2	9.5X	183.3X	45.4X
aget	13.7X	34.4X	98.8X
memcached	2.1X	4.8X	4.1X
apache	1.7X	6.2X	6.0X
mysql	1.6X	15.7X	2.5X

Table 4. Runtime overhead of Maple comparing to native execution time.

itself without any instrumentation. All the numbers shown in Table 4 are normalized to the native execution time. The overhead of the profiler varies depending on the applications. The overhead ranges from 5X (I/O bound applications) to 200X (memory intensive applications), and on average is at about 50X. The overhead of the active scheduler also varies, ranging from 3X to 100X. The average overhead is about 30X. We identify two major factors that contribute to the overhead of the active scheduler. One is due to the extra time delay that we introduce to solve the asynchronous external events problem. The other is because the candidate instructions of some infeasible iRoots are reached so frequently. We believe we still have room to improve the performance of the active scheduler.

6.4.2 Effectiveness of the Active Scheduler

App.	Idiom1	Idiom2	Idiom3	Idiom4	Idiom5
fft	87.5%	100.0%	40.0%	36.0%	25.0%
radix	66.7%	0.0%	0.0%	16.9%	5.6%
pfscan	13.3%	6.7%	5.6%	4.8%	6.4%
pbzip2	23.4%	23.1%	8.0%	5.6%	12.8%
aget	13.6%	3.6%	7.0%	2.7%	8.6%
memcached	7.8%	1.4%	8.4%	2.7%	7.1%
apache	6.0%*	1.0%*	0.0%*	7.0%*	4.0%*
mysql	5.0%*	0.0%*	1.0%*	1.0%*	2.0%*

Table 5. The success rate of the active scheduler (# successfully exposed iRoots/ # total predicted iRoots). For `apache` and `mysql`, we experimented with 100 randomly selected candidate iRoots.

Finally, we discuss how effective the active scheduler is in exposing iRoots. For each application and each idiom, we collect the success rate of the active scheduler (# successfully exposed iRoots/ # total predicted iRoots). We run Maple till its completion except for `apache` and `mysql` which exceed our time budget. For these two applications, we randomly sample 100 candidate iRoots for each idiom and report the success rate. On average, the active scheduler achieves about 28% success rate on idiom1, 17% on idiom2, 9% on idiom3, 10% on idiom4 and 9% on idiom5. We realize that the success rate for the active scheduler is not satisfactory. We identify three major reasons: (1) the profiler algorithm in not accurate in the sense that it cannot detect user customized happens before relations, producing many infeasible iRoots; (2) the active scheduler cannot deal with pre-conditions which might exist for some iRoots. (3) no dynamic information being associated with each iRoot combining with the fact that the currently candidate arbitration mechanism is not sophisticated enough causes some iRoots unlikely to be exposed. Nonetheless, Maple succeeds at exposing concurrency bugs faster than the state of the art randomization techniques, as previously demonstrated. We plan to further improve the accuracy of the profiler and the active scheduler in future.

7. Related Work

This section places our work in the context of other testing methodologies and bug detection tools.

7.1 Coverage Driven Concurrent Testing

There have been a few studies on coverage metrics for concurrent programs [2, 22, 24, 39, 41, 47]. Taylor et al. [41] presented a family of coverage criteria for concurrent Ada programs. All-du-path [47] is a coverage metric for concurrent programs that is based on definition-use pairs. Sherman et al. [39] discusses a few coverage metrics based on synchronizations and inter-thread dependencies. However, unlike Maple, none of these work discusses a synergistic set of testing tools that can help programmers achieve high coverage for the proposed coverage metric and analyze its effectiveness in exposing concurrency bugs.

7.2 Active Testing

Recently several active testing methods have been proposed for concurrent software testing [17, 20, 23, 35, 36, 38, 40]. A typical active testing tool has two phases: a prediction phase and a validation phase. In the prediction phase, these tools use approximate bug detectors to predict potentially buggy thread interleavings in a program. In the validation phase, an active scheduler would try to exercise a suspicious buggy interleaving in a real execution to verify whether it is really a bug or merely a false positive.

In the prediction phase, these tools use either static or dynamic analysis techniques to predict certain types of concurrency bugs in a program such as data races [38], atomicity violations [35, 36, 40], atomic-set serializability violations [17, 23], and deadlocks [20]. The interleaving patterns of these tools represent erroneous interleaving patterns and target certain types of concurrency bugs. Unlike these tools, our interleaving idioms definitions are more general and are used to define a coverage metric. An iRoot is not an anomalous thread interleaving, but just a coverage element that needs to be exposed in an execution during testing. Maple's goal is to achieve a high coverage for our interleaving idioms, which we believe is more general for testing purpose than these bug driven active testing tools. Also, during testing, we memoize interleavings that have been already covered and avoid testing the same interleavings again across different inputs. To the best of our knowledge, none of the previous active testing tools use memoization across different inputs.

We elaborate more on specific differences between our idioms and the bug patterns used in prior studies. Race-

Fuzzer [38] targets data races. The interleaving pattern of data races is different from idiom1. One obvious difference is that idiom1 not only capture inter-thread dependencies that are racy, but also capture non-racy dependencies. Therefore, idiom1 can capture those race free order violation bugs which RaceFuzzer cannot. AssetFuzzer [23] and PECON [17] detects atomic-set serializability violations [42] for Java programs. The interleaving patterns of atomic set serializability violations are also different from our idioms. The main reason is because these patterns are defined over atomic sets and units of work which do not exist and cannot be easily identified in C/C++ programs. DeadlockFuzzer [20] detects deadlocks. Our idiom5 only captures a subset of the deadlocks patterns that can be captured by DeadlockFuzzer, which involve only two threads.

Maple's prediction algorithm is similar to that in [17, 23, 36, 40] except that Maple uses an online algorithm while all these tools use trace based offline algorithms. We believe an online algorithm is more practical in that it does not need to collect execution traces which can potentially be very large for long running programs.

There are two common ways in which validation is performed. One way is to precisely compute an alternate schedule from the observed schedule to expose the bug and then enforce it [17, 21, 40]. However, this requires expense analysis, which is not suitable if the goal is to achieve interleaving coverage like ours. The other approach is to use heuristics, usually best effort methods, to expose predicted interleavings [20, 23, 35, 36, 38]. Maple's active scheduler falls into the second category. We believe Maple is better due to the following reasons: first, Maple uses a novel idea by leveraging the non-preemptive strict priorities provided by the underlying OS, eliminating the need of monitoring all synchronization operations and blocking system calls that are required by those previous systems [23, 35, 38], thus is simpler and less expensive; Second, Maple is more sophisticated in the sense that many of the issues that Maple handles are not handled or not handled well by those tools. For example, RaceFuzzer, CTrigger and AtomFuzzer does not solve the deadlock problem (also referred as thrashing [20]) very well. Also, none of them handle asynchronous external events.

7.3 Stress Testing and Random Testing

Stress testing is still widely used in software industry today. A parallel program is subjected to extreme scenarios during test runs hoping to expose buggy interleavings. This method is clearly inadequate since naively executing a program again and again over an input tends to unnecessarily test similar thread interleavings. A few techniques have been proposed to improve the stress-testing. The main idea is to *randomize* the thread interleavings so that different thread interleavings will be exercised in different test runs. These techniques mainly differ in the way that they randomize the thread interleavings. For example, ConTest [6] injects random delays at synchronization points. PCT [3] assigns random priority to each thread and change priorities at random points during an execution. However, all of these random testing techniques suffer a common problem: the probability of exposing a rare interleaving that can trigger a concurrency bug is very low given that the interleaving space is so huge. Comparing to these random testing techniques, our technique has a much higher probability in exposing concurrency errors due to the following two reasons: first, the idioms guides our tool to test those interleavings that are more likely to cause concurrency errors; second, we have memoization which would guide us to test untested interleavings first. Our results clearly demonstrated the ability of Maple in exposing untested interleavings and bugs faster than the best randomization techniques.

7.4 Systematic Testing

An alternative to stress testing is systematic testing [13, 16, 32, 44] which tries to explore all possible thread interleavings for each test input. Even with partial order reduction techniques [10, 12], the number of thread interleavings to test for a given input is still huge. Therefore, a few heuristics have been proposed to further reduce the testing time at the cost of missing potential concurrency errors. CHESS [32] bounds the number of preemptions in each test run. HaPSet [44] records observed PSet [48] (which are essentially idiom1 iRoots) during testing and guides systematic search towards those interleavings that can produce new PSet dependencies. However, even if these heuristics are used, these tools still suffer from scalability problem, especially for programs that have long execution length. Furthermore, these tools do not have a way to remember tested interleavings across different inputs, unlike Maple. Finally, these systematic testing tools usually require a closed unit testing environment which is in fact not easy to realize in practice. In contrast, a completely closed unit testing environment is not strictly required while using Maple. However, such tools do have one distinct advantage over Maple in that they can provide certain guarantees to find a concurrency bug in a program for a given input.

7.5 Test Input Generation

Test input generation is a testing technique that can be used to achieve high code coverage [4, 14, 15, 34]. For a given program, their goal is to generate test inputs so that testing the program with the generated test inputs can cover most of the code in the program. In contrast, the goal of Maple is to orchestrate thread interleavings for a given test input to cover more thread interleavings. Thus, Maple complements test input generators and aids in achieving higher iRoot coverage.

7.6 Bug Detection Tools

Concurrency bug detection tools can be divided into two categories: static bug detection tools and dynamic bug detection tools. Static concurrency bug detection tools [7, 11, 25, 33, 43] analyze programs statically and predict potential concurrency bugs. Most of the static bug detection tools produce large volume of false positives, thus preventing them from being widely used by programmers. MUVI [25] uses static source analysis to predict multi-variable atomicity violations. It can complement Maple's profiler for finding iRoots of complex idioms, but unlike Maple, it does not use active scheduler for exposing predicted erroneous interleavings.

Dynamic bug detection tools [8, 9, 26, 45, 49] usually require bugs to manifest during monitored runs and do not actively seek to expose incorrect interleavings. Maple could complement dynamic bug detection tools by producing new thread interleavings faster.

8. Conclusion

Maple is a new coverage-driven approach to test multi-threaded programs. To this end, we discussed a coverage metric based on a generic set of interleaving idioms. We discussed a profile-based predictor that determines the set of untested thread interleavings that can be exposed for a given input, and an active scheduler to effectively expose them. A key advantage of our approach over random and systematic testing tools is that we avoid testing the same thread interleavings across different test inputs. Our experience in using Maple to test real-world software shows that Maple can trigger bugs faster by exposing more untested interleavings in a shorter period of time than conventional methods.

Acknowledgments

We thank the anonymous reviewers for comments that improved this paper. This work is supported by the National Science Foundation under the award CCF-0916770 and an Intel grant.

References

[1] P. Barford and M. Crovella. Generating representative web workloads for network and server performance evaluation. In *SIGMETRICS*, pages 151–160, 1998.

[2] A. Bron, E. Farchi, Y. Magid, Y. Nir, and S. Ur. Applications of synchronization coverage. In *PPOPP*, pages 206–212, 2005.

[3] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *ASPLOS*, pages 167–178, 2010.

[4] C. Cadar, D. Dunbar, and D. R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224, 2008.

[5] K. E. Coons, S. Burckhardt, and M. Musuvathi. Gambit: effective unit testing for concurrency libraries. In *PPOPP*, pages 15–24, 2010.

[6] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur. Multithreaded java program test generation. *IBM Systems Journal*, 41(1):111–125, 2002.

[7] D. R. Engler and K. Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *SOSP*, pages 237–252, 2003.

[8] C. Flanagan and S. N. Freund. Fasttrack: efficient and precise dynamic race detection. *Commun. ACM*, 53(11):93–101, 2010.

[9] C. Flanagan, S. N. Freund, and J. Yi. Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. *SIGPLAN Not.*, 43(6):293–303, 2008.

[10] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *POPL*, pages 110–121, 2005.

[11] C. Flanagan and S. Qadeer. A type and effect system for atomicity. *SIGPLAN Not.*, 38(5):338–349, 2003.

[12] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer, 1996.

[13] P. Godefroid. Model checking for programming languages using verisoft. In *POPL*, pages 174–186, 1997.

[14] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *PLDI*, pages 213–223, 2005.

[15] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *NDSS*, 2008.

[16] K. Havelund and T. Pressburger. Model checking java programs using java pathfinder. *STTT*, 2(4):366–381, 2000.

[17] J. Huang and C. Zhang. Persuasive prediction of concurrency access anomalies. In *ISSTA*, pages 144–154, 2011.

[18] D. Jackson and C. Damon. Elements of style: Analyzing a software design feature with a counterexample detector. *IEEE Trans. Software Eng.*, 22(7):484–495, 1996.

[19] V. Jagannath, M. Gligoric, D. Jin, Q. Luo, G. Rosu, and D. Marinov. Improved multithreaded unit testing. In *SIGSOFT FSE*, pages 223–233, 2011.

[20] P. Joshi, C.-S. Park, K. Sen, and M. Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *PLDI*, pages 110–120, 2009.

[21] V. Kahlon and C. Wang. Universal causality graphs: A precise happens-before model for detecting bugs in concurrent programs. In *CAV*, pages 434–449, 2010.

[22] B. Krena, Z. Letko, and T. Vojnar. Coverage metrics for saturation-based and search-based testing of concurrent software. In *RV*, 2011.

[23] Z. Lai, S.-C. Cheung, and W. K. Chan. Detecting atomic-set serializability violations in multithreaded programs through active randomized testing. In *ICSE (1)*, pages 235–244, 2010.

[24] S. Lu, W. Jiang, and Y. Zhou. A study of interleaving coverage criteria. In *ESEC/SIGSOFT FSE*, pages 533–536, 2007.

[25] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou. Muvi: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *SOSP*, pages 103–116, 2007.

[26] S. Lu, J. Tucek, F. Qin, and Y. Zhou. Avio: detecting atomicity violations via access interleaving invariants. In *ASPLOS*, pages 37–48, 2006.

[27] B. Lucia, J. Devietti, K. Strauss, and L. Ceze. Atom-aid: Detecting and surviving atomicity violations. In *ISCA*, pages 277–288, 2008.

[28] C.-K. Luk, R. S. Cohn, R. Muth, H. Patil, A. Klauser, P. G. Lowney, S. Wallace, V. J. Reddi, and K. M. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, pages 190–200, 2005.

[29] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI*, pages 446–455, 2007.

[30] M. Musuvathi and S. Qadeer. Partial-order reduction for context-bounded state exploration. Technical Report MSR-TR-2007-12, Microsoft Research, 2007.

[31] M. Musuvathi and S. Qadeer. Fair stateless model checking. In *PLDI*, pages 362–371, 2008.

[32] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI*, pages 267–280, 2008.

[33] M. Naik, C.-S. Park, K. Sen, and D. Gay. Effective static deadlock detection. In *ICSE*, pages 386–396, 2009.

[34] C. Pacheco and M. D. Ernst. Randoop: feedback-directed random testing for java. In *OOPSLA Companion*, pages 815–816, 2007.

[35] C.-S. Park and K. Sen. Randomized active atomicity violation detection in concurrent programs. In *SIGSOFT FSE*, pages 135–145, 2008.

[36] S. Park, S. Lu, and Y. Zhou. Ctrigger: exposing atomicity violation bugs from their hiding places. In *ASPLOS*, pages 25–36, 2009.

[37] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie. Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs. In *CGO*, pages 2–11, 2010.

[38] K. Sen. Race directed random testing of concurrent programs. In *PLDI*, pages 11–21, 2008.

[39] E. Sherman, M. B. Dwyer, and S. G. Elbaum. Saturation-based testing of concurrent programs. In *ESEC/SIGSOFT FSE*, pages 53–62, 2009.

[40] F. Sorrentino, A. Farzan, and P. Madhusudan. Penelope: weaving threads to expose atomicity violations. In *SIGSOFT FSE*, pages 37–46, 2010.

[41] R. N. Taylor, D. L. Levine, and C. D. Kelly. Structural testing of concurrent programs. *IEEE Trans. Software Eng.*, 18(3):206–215, 1992.

[42] M. Vaziri, F. Tip, and J. Dolby. Associating synchronization constraints with data in an object-oriented language. In *POPL*, pages 334–345, 2006.

[43] J. W. Voung, R. Jhala, and S. Lerner. Relay: static race detection on millions of lines of code. In *ESEC/SIGSOFT FSE*, pages 205–214, 2007.

[44] C. Wang, M. Said, and A. Gupta. Coverage guided systematic concurrency testing. In *ICSE*, pages 221–230, 2011.

[45] D. Weeratunge, X. Zhang, and S. Jagannathan. Analyzing multicore dumps to facilitate concurrency bug reproduction. In *ASPLOS*, pages 155–166, 2010.

[46] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: Characterization and methodological considerations. In *ISCA*, pages 24–36, 1995.

[47] C.-S. D. Yang, A. L. Souter, and L. L. Pollock. All-du-path coverage for parallel programs. In *ISSTA*, pages 153–162, 1998.

[48] J. Yu and S. Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. In *ISCA*, pages 325–336, 2009.

[49] W. Zhang, J. Lim, R. Olichandran, J. Scherpelz, G. Jin, S. Lu, and T. W. Reps. Conseq: detecting concurrency bugs through sequential errors. In *ASPLOS*, pages 251–264, 2011.

[50] W. Zhang, C. Sun, and S. Lu. Conmem: detecting severe concurrency bugs through an effect-oriented approach. In *ASPLOS*, pages 179–192, 2010.