Testing Atomicity of Composed Concurrent Operations

Ohad Shacham
Tel Aviv University
ohad.shacham@cs.tau.ac.il

Nathan Bronson
Stanford University
nbronson@cs.stanford.edu

Alex Aiken
Stanford University
aiken@cs.stanford.edu

Mooly Sagiv
Tel Aviv University
msagiv@post.tau.ac.il

Martin Vechev
ETH Zurich and IBM Research
martin.vechev@gmail.com

Eran Yahav [*]
Technion
yahave@cs.technion.ac.il

Abstract

We address the problem of testing atomicity of composed concurrent operations. Concurrent libraries help programmers exploit parallel hardware by providing scalable concurrent operations with the illusion that each operation is executed atomically. However, client code often needs to compose atomic operations in such a way that the resulting composite operation is also atomic while preserving scalability. We present a novel technique for testing the atomicity of client code composing scalable concurrent operations. The challenge in testing this kind of client code is that a bug may occur very rarely and only on a particular interleaving with a specific thread configuration. Our technique is based on modular testing of client code in the presence of an adversarial environment; we use commutativity specifications to drastically reduce the number of executions explored to detect a bug. We implemented our approach in a tool called COLT, and evaluated its effectiveness on a range of 51 real-world concurrent Java programs. Using COLT, we found 56 atomicity violations in Apache Tomcat, Cassandra, MyFaces Trinidad, and other applications.

Categories and Subject Descriptors D.2.5 [*Testing and Debugging*]; D.1.3 [*Concurrent Programming*]

General Terms Verification

Keywords concurrency, linearizability, testing, composed operations, collections

[*] Deloro Fellow

OOPSLA'11, October 22–27, 2011, Portland, Oregon, USA.
Copyright © 2011 ACM 978-1-4503-0940-0/11/10...$10.00

1. Introduction

Concurrent data structures are becoming critical components of many systems [27] but are notoriously hard to get right (e.g., [11]). To shield programmers from the complexity of concurrent data structures, modern languages hide their implementations in libraries (e.g., [1, 3, 19]). Data structures provided by the library usually guarantee that their operations are *atomic*. That is, every individual operation appears to take place instantaneously at some point between its invocation and its return, and the implementation details of the operation can be ignored by the programmer.

Custom Concurrent Data Structures While the library provides basic concurrent data structures, client code often needs specific concurrent data structures supporting additional operations. Such custom operations may combine several operations of the concurrent library. The main challenge when composing atomic operations is to guarantee that the composed operation is also atomic.

It is important to note that programmers usually compose the operations of underlying concurrent data structures without using a wrapping lock, because guaranteeing atomicity of the composed operation using a wrapping lock requires wrapping every other operation of the concurrent data structure with the same lock. Besides the major code modifications entailed by this approach, it severely limits concurrency and defeats the original purpose of using a concurrent data structure.

For example, Figure 1, taken from OpenJDK [4], implements an increment operation for a concurrent histogram (as a map of counters). The histogram is implemented as a subclass of `ConcurrentHashMap` and other map operations (e.g., `remove`, `get`, `put`) can be performed directly, and concurrently, on the map by client code. The increment operation does not use any locks (for the reasons mentioned above), and instead guarantees atomicity using an optimistic concurrency loop, using the (atomic) operations `putIfAbsent` and `replace` provided by the underlying

```
void inc(Class<?> key) {
 for (;;) {
  Integer i = get(key);
  if (i == null) {
   if (putIfAbsent(key, 1) == null) //@LP if succeeds
   return;
  } else {
   if (replace(key, i, i + 1)) //@LP if succeeds
   return;
  }
 }
}
```

Figure 1. A correct concurrent increment operation for a map of concurrent counters, implemented using ConcurrentHashMap operations (from OpenJDK [4]).

`ConcurrentHashMap`. While the code in Figure 1 does guarantee atomicity, we have found that many other open source clients combine atomic operations of underlying concurrent data structures in a non-atomic way.

Checking Atomicity Given the difficulty of writing composed atomic operations, it is desirable to provide programmers with an automatic technique for checking atomicity. Existing approaches to dynamic atomicity checking, such as Velodrome [15], identify violations of *conflict-serializability* [7] using an online computation of the happens-before relation over program executions.

Unfortunately, the conflict-serializability property is inappropriate for concurrent data structures since correct data structures often violate it. For example, the increment code of Figure 1 is not conflict-serializable. Indeed, many of the programs that we investigated are not conflict-serializable since they contain two memory accesses in the same method with an intervening write. (In Section 5, we show that for all of our benchmarks, correct methods would have been rejected by a conflict-serializability checker.) Therefore, instead of using conflict-serializability as the correctness criterion, which results in many false alarms, we focus on checking *linearizability* [21] of the concurrent data structure.

The Challenge Given an application, our goal is to test whether its composed concurrent operations might violate atomicity. In the rest of this paper, we refer to a method implementing the composed concurrent operation as the *method under test* (MUT).

Since atomicity violations often depend on specific thread configurations and schedules, very few program executions may exhibit the error. In fact, the biggest challenge for dynamic atomicity checking tools such as Velodrome [15] and Lineup [8] is identifying the thread configuration and schedule that expose an error. In our experience, exposing a single bug in TOMCAT using a non-modular approach took about one week of work and required us to manually write an input test and introduce scheduling bias.

COLT addresses the challenge of identifying thread configurations and scheduling by: (i) modularly checking that composed operations are linearizable w.r.t. an open environment relative to the collection; (ii) guiding execution by interleaving non-commutative collection operations.

Modular Checking of Linearizability Rather than checking the application as a whole, COLT tests it under an open environment with respect to collection operations. The open environment over-approximates any possible manipulation of the environment on the collection used by the MUT. This allows us to test the MUT in an adversarial environment rather than trying to reproduce adversarial testing conditions within the original application.

However, since COLT covers interleaving that may not occur for the MUT in the context of its application, it might produce false alarms in cases where application-specific invariants create a restricted environment in which the MUT is actually atomic. Fortunately, we found that such situations are rare, and on real programs COLT has a very low false alarm rate. Intuitively, this happy situation arises because operations can usually be executed in an arbitrary order without violating linearizabilty. Furthermore, checking the code for linearizability against all possible environments guarantees that the code is robust to future extensions of the client. Indeed, our experience has been that programmers fix linearization bugs discovered by COLT even if such bugs are unlikely or impossible in the current application, because they are concerned that future, apparently unrelated changes might trigger these latent problems.

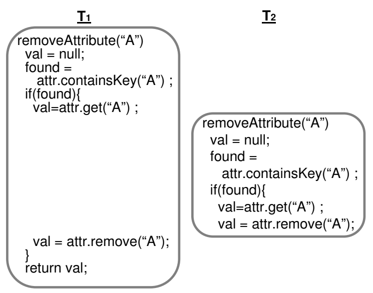
Adversarial Execution Guided by Non-Commutativity Our goal is to test the atomicity of the MUT. We would like to avoid exploring executions that result in collection values already observed by the MUT in earlier executions. Indeed, COLT aims for every newly explored execution path to yield a new collection result for the MUT. This can be viewed as a kind of partial order reduction (e.g., [18]) where commutativity is checked at the level of atomic collection operations.

Main Contributions The main contributions of this paper are:

- We present an approach for modular testing of linearizability for composed concurrent operations.
- We demonstrate our approach in the context of Java concurrent collections. We have implemented a dynamic tool called COLT that checks the atomicity of composed concurrent collection operations.
- We use collection non-commutativity specifications to direct exploration. This significantly increases the bug hunting capabilities of our tool, as confirmed by our experimental results at Section 5. We show that this optimization cannot filter out linearizability violations.
- We show that COLT is effective in detecting real bugs while maintaining a very low rate of false alarms. Using COLT, we were able to identify 56 linearizability violations in Apache Tomcat, Cassandra, MyFaces Trinidad,

```
1  Attribute removeAttribute(String name) {
2      Attribute val = null;
3      synchronized(attr) {
4          found = attr.containsKey(name) ;
5          if (found) {
6              val = attr.get(name);
7              attr.remove(name);
8          }
9      }
10     return val;
11 }
```

(a)



```
T1
removeAttribute("A")
  val = null;
  found =
    attr.containsKey("A") ;
  if(found){
    val=attr.get("A") ;



    val = attr.remove("A");
  }
  return val;
```

```
T2
removeAttribute("A")
  val = null;
  found =
    attr.containsKey("A") ;
  if(found){
    val=attr.get("A") ;
    val = attr.remove("A");
```

```
Attribute removeAttribute(String name) {
    Attribute val
      = attr.get(name) ; //@LP val==null
    if (val != null) {
        val = attr.remove(name); //@LP
    }
    return val;
}
```

(c)

(b)

Figure 2. (a) An erroneous code from Apache Tomcat [2] version 5 where `attr` is a `HashMap`. In version 6 the `attr` implementation was changed to a `ConcurrentHashMap` and `synchronized(attr)` from line 3 was removed. (b) is an execution that shows an atomicity violation in `removeAttribute` at Tomcat version 6. (c) is a fixed linearizable version of `removeAttribute`.

Adobe BlazeDS, as well as in other real life applications. Some of these violations expose real bugs, while other violations represent potential bugs in future client implementations.

2. Motivation

Figure 2a shows a composed operation taken from Apache Tomcat [2] version 5. In this version `attr` is a sequential `HashMap`. The atomicity of the composed operation is guaranteed by wrapping the composed operation with `synchronized(attr)` block. The operation maintains the invariant that `removeAttribute` returns either the value it removes from `attr` or `null`.

In Tomcat version 6, the developers decided to utilize fine-grain concurrency, therefore, `attr`'s implementation was changed from a `HashMap` to a `ConcurrentHashMap` and consequently the programmers decided to remove the `synchronized(attr)` blocks from the program as well as from line 3 of Figure 2a. This caused a bug which was identified by COLT.

Hard to cover rarely-executed traces Figure 2b shows an execution of `removeAttribute` revealing the invariant violation in the composed operation in Tomcat version 6. In this execution trace, thread T_1 returns a value different than `null` even though it does not remove a value from `attr`, which breaks the invariant. This violation occurs due to the `remove("A")` operation by thread T_2 occurring between the `get("A")` and the `remove("A")` of thread T_1. In order to reveal the violation, it is not enough that `remove` occurs at a specific point in the trace, but also that the collection operations all use same key "A". This composed operation is not linearizable and COLT successfully detects the violation. We reported this bug to Tomcat's developers as well as the fix in Figure 2c and they acknowledged the violation and accepted the fix.

While COLT discovers this violation, tools that run the whole program are unlikely to succeed — in particular, the above violation can only be triggered with a specific choice of keys.

2.1 The Main Ideas in our Solution

We now briefly describe the main ideas in our tool.

Composed Operations Extraction COLT includes a semi-automatic technique that significantly helps the user in extracting composed operations out of whole applications. A simple static analysis identifies methods that may invoke multiple collection operations. These methods are returned to the user for further review. In some cases the static analysis identifies a composed operation inside a larger method and the user must manually extract and generate the MUT.

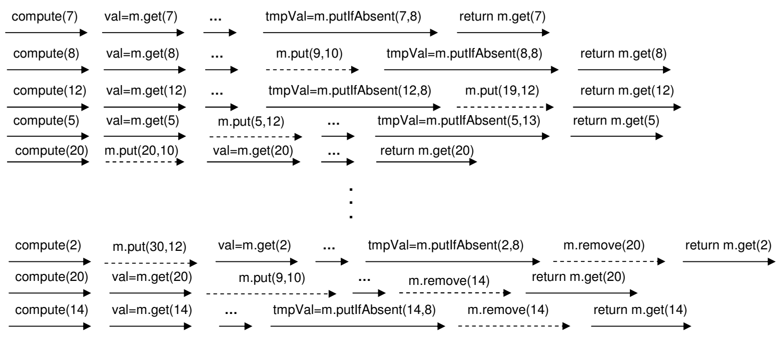
Modular Checking of Linearizability COLT checks linearizability [21] in a modular way by invoking one MUT at a time in an environment that performs arbitrary collection operations concurrently.

Adversarial Execution Guided by Non-commutativity Due to the rare nature of many atomicity violations, a dynamic tool for detecting atomicity violations must use some form of focused exploration to be effective. In COLT, we use collection semantics to reduce the space of explored interleavings, without filtering out interleavings leading to linearizability violations.

As a first step, we assume that the underlying collection implementation is linearizable and thus the internal representation of the collection can be abstracted away and collection operations can be assumed to execute atomically. As done in [6], this allows us to consider interleavings at the level of collection operations, without considering interleavings of their internal operations. However, the key insight that we use is that before and after each collection operation op_1 executed by the MUT, the environment chooses a collection operation op_2 to execute that does not commute with

```
1 V compute(K k) {
2   V val = m.get(k);
3   if (val == null) {
4     val = calculateVal(k);
5     m.putIfAbsent(k, val);
6   }
7   return m.get(k);
8 }
```

(a)


compute(7) val=m.get(7) ... tmpVal=m.putIfAbsent(7,8) return m.get(7)
compute(8) val=m.get(8) ... m.put(9,10) tmpVal=m.putIfAbsent(8,8) return m.get(8)
compute(12) val=m.get(12) ... tmpVal=m.putIfAbsent(12,8) m.put(19,12) return m.get(12)
compute(5) val=m.get(5) m.put(5,12) ... tmpVal=m.putIfAbsent(5,13) return m.get(5)
compute(20) m.put(20,10) val=m.get(20) ... return m.get(20)
compute(2) m.put(30,12) val=m.get(2) ... tmpVal=m.putIfAbsent(2,8) m.remove(20) return m.get(2)
compute(20) val=m.get(20) m.put(9,10) ... m.remove(14) return m.get(20)
compute(14) val=m.get(14) ... tmpVal=m.putIfAbsent(14,8) m.remove(14) return m.get(14)

(b)

Figure 3. (a) Non linearizable example, capturing bugs from `Adobe BlazeDS` and others; (b) sample executions of `compute` concurrently with a client running arbitrary collection operations. Solid and dashed edges represent operations by `compute` and the client respectively.

op_1. Note that scheduling operations only before every collection operation of the MUT is insufficient, as this strategy can omit interleavings that lead to linearizability violations (see Section Section 4.5).

Figure 3a shows an example of a non-linearizable method inspired by bugs from `Adobe BlazeDS`, `Vo Urp` and `Ehcache-spring-annotations`. The procedure `compute(K k)` uses an underling concurrent collection to memoize the value computed by `calcuateVal(k)`. When the value for a given key is cached in the collection it is returned immediately; when the value for a given key is not available, it is computed and inserted into the collection.

Figure 3b shows sample executions of `compute(K k)` running concurrently with a general client that performs arbitrary collection operations with arbitrary arguments. Operations of `compute(K k)` are shown with solid arrows, operations of the general client are depicted with dashed arrows. Out of the 8 sample executions in the figure, only 1 exposes the atomicity violation; in practice a smaller fraction of executions reveal the atomicity violation. As we show in Section 5, a random search of the space of executions fails to find even a single violation in practice.

To trigger an atomicity violation in a MUT, particular collection operations, with a particular key, must interleave between the collection operations of the MUT. The set of client operations that can potentially trigger an atomicity violation in a point of execution in the MUT can be characterized as operations that do not commute with the collection operation in the MUT. In the example, the executions that reveal the atomicity violation are those in which a client operation affects the result of the `m.get(k)` in line 7 of `compute(K k)`. This suggests that an adversarial client should focus on scheduling an operation that does not commute with `m.get(k)` right before scheduling this MUT operation. An example of an operation that does not commute with `m.get(k)` is `m.remove(k)`, as shown in the last execution of Figure 3b.

3. Preliminaries

In this section we define linearizability [21] and linearization points, show two ways for checking linearizability, and finally define a notion of commutativity [29].

3.1 Linearizability

Linearizability [21] is defined with respect to a sequential specification (pre/post conditions). A concurrent object is linearizable if each execution of its operations is equivalent to a legal sequential execution in which the order between non-overlapping operations is preserved.

An operation *op* is a pair of an invocation event and a response event. An invocation event is a triple $(tid, op, args)$ where tid is the thread identifier, *op* is the operation identifier, and $args$ are the arguments. Similarly, a response event is a triple (tid, op, val) where tid and *op* are as just defined, and val is the value returned from the operation. For an operation *op*, we denote its invocation by $inv(op)$ and its response by $res(op)$. A *history* is a sequence containing invoke and response events. A *complete invocation* of an operation *op* is a history with two events where the first event is $inv(op)$ and the second event is $res(op)$ A *complete history* is a history without pending invocation events (that is, all invocation events have a matching response event). A *sequential history* is one in which each invocation is a complete invocation. A *thread subhistory*, $h|tid$ is the subsequence of all events in h with thread id tid. A sequential history h is *legal* if it belongs to the sequential specification. Two histories

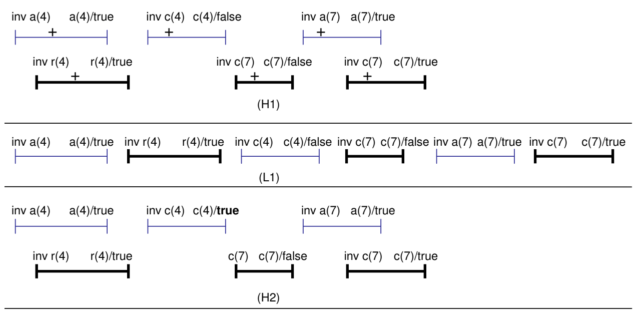


Figure 4. Concurrent histories and possible sequential histories corresponding to them.

h_1, h_2 are *equivalent* when for every tid, $h_1|tid = h_2|tid$. An operation op_1 precedes op_2 in h, and write $op_1 <_h op_2$, if $res(op_1)$ appears before $inv(op_2)$ in h. A history h is linearizable when there exists an equivalent legal sequential history s, called a linearization, such that for every two operations op_1, op_2, if $op_1 <_h op_2$ then $op_1 <_s op_2$. That is, s is equivalent to h, and respects the global ordering of non-overlapping operations in h.

Example Next we illustrate the concepts of linearizability and histories on a concurrent set data structure with methods `add`, `remove`, and `contains`.

Figure 4 shows two concurrent histories H_1 and H_2, and a sequential history L_1. All histories involve two threads invoking operations on a shared concurrent set. In the figure, we abbreviate names of operations, and use `a`,`r`, and `c`, for `add`, `remove`, and `contains`, respectively. We use `inv op(x)` to denote the invocation of an operation `op` with an argument value `x`, and `op/val` to denote the response `op` with return value `val`.

Consider the history H_1. For now, ignore the + symbols. In this history, `add(4)` is overlapping with operation `remove(4)`, and `add(7)` overlaps `contains(7)`. The history H_1 is linearizable. We can find an equivalent sequential history that preserves the global order of non-overlapping operations. The history L_1 is a possible *linearization* of H_1 (in general, a concurrent history may have multiple linearizations).

In contrast, the history H_2 is non-linearizable. This is because `remove(4)` returns $true$ (removal succeeded), and `contains(4)` that appears *after* `remove(4)` in H_2 also returns $true$.

Linearizability provides the illusion that for each operation *op*, there exists a point between its invocation $inv(op)$ and response $res(op)$ in the history h where *op* appears to take effect instantaneously. This point is typically referred to as the *linearization point* $lp(op)$ of the operation *op*. Given a concurrent history h, the (total) ordering between these points induces a linearization marked as $lin(h)$. We refer to $lin(h)$ as the *reference history* of h.

Example Consider the history H_1 of Figure 4, the + symbols in the figure denote the linearization point in each operation. The relative ordering between these points determines the order between overlapping operations, and therefore determines a unique linearization of H_1, shown as L_1.

3.2 Checking Linearizability

There are two alternative ways to check linearizability [28]: (i) *automatic linearization*—explore all permutations of a concurrent execution to find a valid linearization; (ii) *linearization points*—build a linearization ($lin(h)$) on-the-fly during a concurrent execution, using linearization points.

The first technique is fully automatic and checks whether there exists a linearization for the concurrent execution. The second technique requires either user-provided linearization points or uses a heuristic for guessing linearization points. This technique checks whether the concurrent execution is equivalent to a specific legal sequential execution defined by the linearization points.

3.3 Commutativity

Following [29], we say that an operation op_1 commutes with an operation op_2 with respect to a complete history $base$, if s_1 and s_4 are complete invocations of op_1, s_2 and s_3 are complete invocations of op_2, and *base*$\cdot s_1 \cdot s_3$ and *base*$\cdot s_2 \cdot s_4$ are equivalent histories. Conversely, if *base* $\cdot$ s_1 $\cdot$ s_3 and *base* $\cdot$ s_2 $\cdot$ s_4 are not equivalent histories then op_1 and op_2 does not commute with respect to $base$.

Example An example of non-commutative operations is `r(4)` and `c(4)` in history L_1 from Figure 4. These operations are non-commutative with respect to the complete history `a(4)`, because if the order between the two operations changes and `c(4)` is executed before `r(4)` then `c(4)` would instead return `true`.

4. Commutativity-Guided Checking

COLT aims to check linearizability in a modular fashion by checking for violations of a composed concurrent operation. In this section we present our commutativity-based checking of linearizability.

4.1 General Setting

COLT exploits the fact that collection operations are atomic and encapsulated (the state of the collection is only accessed through methods of the collection), which allows COLT to only explore interleavings between the (atomic) operations, without having to interleave the implementations of the operations. Such histories, consisting of atomic operations, can be generated by a single thread.

Therefore, to test linearizability, we generate histories via only two threads: one thread that runs the MUT, and another (adversarial) thread that runs arbitrary collection operations. The general technique used by COLT's environment contains three components:

Operation	remove(k)	get(k)	putIfAbsent(k, v_2)
remove(k)	get(k) $\neq$ null	get(k) $\neq$ null	true
get(k)	get(k) $\neq$ null	false	get(k) = null
putIfAbsent(k, v_1)	true	get(k) = null	get(k) = null

Figure 5. Non-commutativity Specification for Map. For simplicity, we assume that v_1 and v_2 are non-null.

- MUT CLIENT, which implements the thread running the MUT (MUT THREAD) and is responsible for generating inputs for the MUT.
- ENV CLIENT, which implements the thread running arbitrary collection operations (ENV THREAD) and is responsible for choosing collection operations together with their input arguments.
- SCHEDULER, which is responsible for scheduling the ENV THREAD and the MUT THREAD.

Main Question The main question in building COLT is:

How should we define the behavior of MUT CLIENT*,* ENV CLIENT*, and* SCHEDULER *to effectively find bugs?*

4.2 Naive Approach

The naive approach is to use randomness. Here, the MUT CLIENT would randomly select input values for the MUT, the ENV CLIENT would randomly select collection operations together with their input arguments, and the SCHEDULER would randomly schedule between the MUT THREAD and the ENV THREAD.

This approach is ineffective, as the space of executions using random values is huge, and only a few of these choices are likely to lead to executions that contain violations. Indeed, as shown in Section 5, the naive approach does not work well; the use of random choices fails to expose even a single violation in realistic examples after hours of execution. Note that even if we consider a systematic exploration of possible schedules (via say context-bounding [25]), we will still not find many problematic behaviors unless we know the precise values with which to exercise these schedules.

4.3 Commutativity Guided Scheduling

The main insight behind our approach is to leverage the commutativity of collection operations in order to define the behaviors of ENV CLIENT and SCHEDULER. With our approach, we do not produce histories that we know are certainly linearizable. This enables us to significantly prune the massive search space of possible executions and focus on schedules that can trigger violations.

Commutativity Specifications A non-commutativity specification for some of the Map operations is shown in Table 5. Such specifications are easy to obtain from the specification of Map's methods. The way to make use of this table is as follows: i) select an operation op_1 from a column; ii) select an operation op_2 from a row; iii) pick a complete history $base$ such that the state of the collection at the end of $base$ satisfies the condition in the box. Then, by the definition of commutativity, $op_1(k)$ and $op_2(k)$ do not commute from $base$. In the case where the box is *false*, it means the operations always commute from any complete history $base$, i.e, they are never non-commutative. Conversely, when the box is *true*, the operations never commute. That is, regardless of which complete history $base$ is selected, the operations never commute with respect to $base$.

General Approach It is possible to augment ENV CLIENT and SCHEDULER to be commutativity-aware. That is, ENV CLIENT selects an operation with such keys and values so that the operation does not commute with any of the preceding operations performed by MUT THREAD or with the operation that is about to be performed by the MUT THREAD. The SCHEDULER is responsible for scheduling the ENV THREAD before and after each collection operation performed by the MUT THREAD.

The intuition behind our approach is conceptually simple: if the ENV THREAD operation commutes with all of the MUT THREAD's operations, then we can shift the ENV THREAD's operation to not interleave with the MUT THREAD. That is, we can always produce a linearization of the concurrent history by simply moving all of ENV THREAD's operations before or after the operations in MUT THREAD. Indeed, producing concurrent histories that we know can always be linearized is not useful for finding linearizability violations.

In addition, we also augment the SCHEDULER so that underlying collection operations are always performed atomically. For instance, in Figure 3a, an operation such as `remove()` will be allowed to preempt `compute()`, yet we always schedule `remove()` to execute atomically, without preemption. This is because the underlying collection (excluding MUT) is already linearizable and due to the collection encapsulation the MUT always accesses the collection state through the underlying methods (i.e., `get()`) and hence the MUT cannot violate the linearizability of `remove()`.

Implemented Approach In this work, we implemented a simpler version of the more general approach. Rather than checking if a scheduled operation commutes with *all* preceding operations performed by MUT THREAD, we only check whether the operation does not commute with the operation just performed by MUT THREAD or about to be performed by MUT THREAD. Although this approach may miss linearizability violations, as we show later, it is still a very effective approach for discovering errors in practice.

4.4 One Thread Implementation

Our approach uses two threads ENV THREAD and MUT THREAD to detect linearizability violations. Using the (non) commutativity specification of the map, we know that ENV THREAD may run operation *op* only before or after a map op-

```
void ENV(operation, args, map) {
  if (*) {
    (nonCommutativeOp, args) =
      getNonComOp(operation, args, map);

    map.nonCommutativeOp(args);
  }
}
```

(a)

```
V compute(K k) {
 ENV(‘‘get’’, k, m);
 val = m.get(k);
 ENV(‘‘get’’, k, m);
 if (val == null) {
   val = calculateVal(k);
   ENV(‘‘putIfAbsent’’, k, val, m);
   m.putIfAbsent(k, val);
   ENV(‘‘putIfAbsent’’, k, val, m);
 }
 ENV(‘‘get’’, k, m);
 val = m.get(k);
 ENV(‘‘get’’, k, m);
 return val;
}
```

(b)

Figure 6. One thread instrumentation done by COLT for function `compute` from Figure 3a. (a) shows function `ENV` imitating ENV THREAD and (b) shows the instrumentation of `compute` done by COLT.

eration *op*′ of MUT THREAD S.T. *op* and *op*′ do not commute. Therefore, we can implement both logical threads using a single actual thread as follows: given a MUT M we instrument M to include ENV THREAD operations before and after map operations of the MUT THREAD.

Figure 6a shows a method `ENV`, which takes an operation (`operation`), its arguments (`args`), and a `map`. `ENV` imitates ENV THREAD by make a non-deterministic choice whether to run an operation (given by `getNonComOp`) non-commutative to `operation(args)` at `map`. Method `getNonComOp` takes an `operation`, its arguments (`args`), and a `map`. This function returns one operation non commutative to `operation(args)` at `map`.

Figure 6b shows the instrumentation done by COLT for function `compute` from Figure 3a. As Figure 6b shows, `ENV` is called before and after every operation of `m` done by `compute`. This instrumented program uses one thread and imitates an environment thread running in parallel to `compute`.

4.5 Commutativity Guided Scheduling: An Example

Let us illustrate how our approach works on a simple example. Suppose that we would like to check the linearizability of the `compute()` operation from Figure 3a using a user-provided linearization point. The execution that we will construct next is shown in Figure 7.

The example shows a concurrent execution of the MUT THREAD and the ENV THREAD together with the concurrent history of the concurrent execution. In addition, the example shows the REFERENCE THREAD representing the sequential run together with its corresponding sequential history.

The execution starts when the MUT CLIENT selects an arbitrary key, say 7, for `compute()` and the SCHEDULER invoked the MUT THREAD to perform its operations up to but not including `putIfAbsent(7, 14)`. Next, the SCHEDULER, using a non-commutativity specification, decides to run the ENV THREAD. The ENV THREAD is invoked, and ENV CLIENT consults Table 5. To achieve non-commutativity with the next operation of `compute()`, i.e., `putIfAbsent(7, 14)`, ENV CLIENT must also use the same key, i.e., key 7 (otherwise, if the keys are different, the operations will commute). From Table 5, we observe that `putIfAbsent(7, v)` does not commute with `putIfAbsent(7, 14)` when key 7 is not in the map (for *any* value of v). Therefore, ENV CLIENT selects the operation `putIfAbsent(7,12)` for the ENV THREAD (the value 12 for v is picked randomly).

Since `putIfAbsent(7,12)` executes atomically, this is a linearization point for `putIfAbsent(7,12)`, and hence we execute the REFERENCE THREAD that executes operation `putIfAbsent(7,12)` on the reference map. This adds the first operation to the sequential history, and the results of the first operation `putIfAbsent(7,12)` on the sequential and concurrent history are compared. In this case, the results are the same, i.e., $null$, and therefore, a violation of linearizability is not yet detected.

Next, the SCHEDULER lets the MUT THREAD run and the operation `putIfAbsent(7, 14)` operation is performed. Assuming that `putIfAbsent(7,14)` is a linearization point of `compute(7)`, the reference implementation of `compute` is executed atomically and the `compute` operation (marked as c) is added to the sequential history with its result 12. Later, `compute(7)`'s result in the sequential history will be compared to `compute(7)`'s result in the concurrent history (when it will be available).

At this point, once again, the ENV THREAD is scheduled before `get(7)`. Again, ENV CLIENT consults Table 5, and it observes that `remove(7)` does not commute with `get(7)` as key 7 is in the map at this point. Therefore, ENV CLIENT performs `remove(7)` after `putIfAbsent(7,14)` and before `get(7)`. As `remove(7)` is executed atomically, this is a linearization point for `remove(7)` and hence we execute the REFERENCE THREAD that executes `remove(7)` on the reference map. This adds an operation to the sequential history (marked as r), and the results of the operation `remove(7)` on the sequential and concurrent history are compared. In this case, the results are the same, i.e., 12, and therefore, a violation of linearizability is not yet detected.

However, when the `get(7)` operation executes by the MUT THREAD and subsequently `compute(7)` returns, it will return the value $null$ and the response and the result of `compute(7)` is added to the concurrent history. When we compare the result of `compute(7)` on the concurrent and

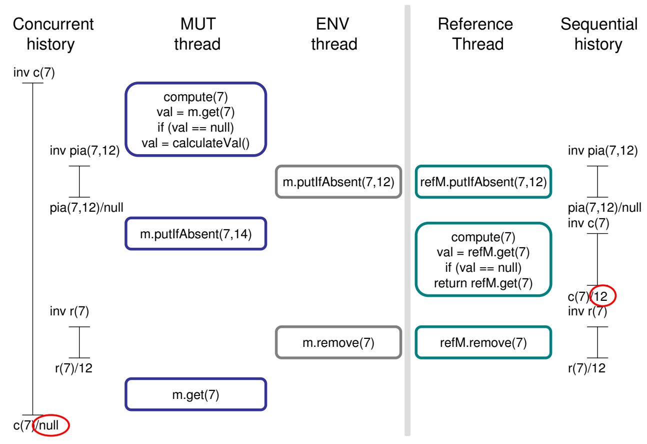


Figure 7. An example execution of our technique, testing method `compute()` from Figure 3a.

```
1 void add(K k) {
2   val = m.get(k); // @LP (val != null)
3   if (val == null) {
4     m.put(k, 2); // @LP
5   }
6 }
```

Figure 8. An example showing that restricting ENV THREAD to perform operations only before map operation of the MUT can miss linearizability violation.

sequential histories, we obtain different results, i.e., *null* vs. 12, triggering a linearizability violation.

Scheduling environment operations only before MUT operations Note that scheduling operations only before every collection operation of the MUT can sometimes lead to missing linearizability violations. Figure 8 shows a non-linearizable function `add`, where restricting ENV THREAD to perform operations only before `m` operations performed by `add` would miss all non-linearizable executions of `add`. Function `add` receives an input key `k` and in case that `k` is not already in `m` adds `k` to `m` with the value 2. In this function the user provided two conditional linearization points. One at line 2, in case that $val \neq null$ and the second one at line 4.

Figure 9 shows an execution of `add` that reveals the linearizability violation. All the violating executions of `add` are similar to the one shown at Figure 9. In this execution, the linearizability violation of `add` occurs due to the `get` operation that occurs after the last operation of `add`. Therefore, restricting the environment to perform operations only before `m` operations performed by `add` would prune this violating execution of `add`.

5. Experimental Evaluation

In this section we explain COLT's implementation and demonstrate the effectiveness of COLT by evaluating it on a range

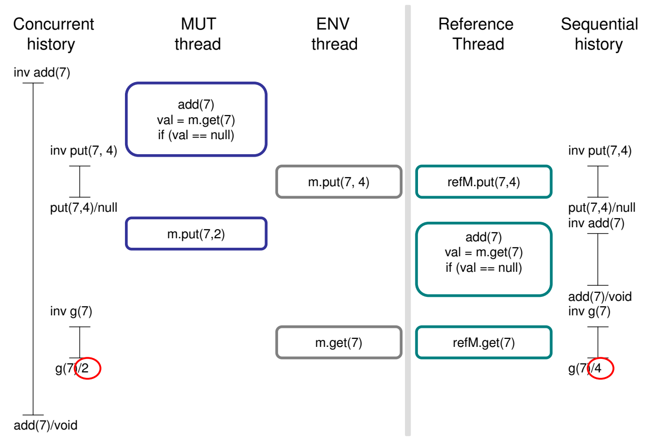


Figure 9. An example execution of our technique, testing method `add(K k)` from Figure 8. Note that the linearizability violation revealed by the different return value of the `get` method occurs after the last operation of `add`.

of real-world applications. Using our approach we found a number of errors, substantiating our hypothesis that programmers often make incorrect assumptions when using concurrent collections. Together with the error, we reported a suggested fix for that error to the development team. In many cases, our fixes were accepted by the development team and incorporated in the application. All of our experiments were carried out using 64-bit Linux with 8GB of RAM running on a dual-core, hyper-threaded 2.4Ghz AMD Opteron.

5.1 Implementation

An outline of COLT's implementation is shown in Figure 10. The programmer provides a multithreaded program to the `MUT Extractor`. The `MUT Extractor` uses a simple static analysis that identifies methods that may invoke multiple collection operations. These methods are returned to the user for review. In some cases the `MUT Extractor` identifies a specialized concurrent data structure operation inside a large method and the user needs to manually extract and generate the MUT (which occurred in 19% of our examples). The user writes a driver that generates keys and values for the MUT and for the collection. In addition, the user provides a non-commutative driver that takes an input object and returns a different object. This driver is used by the non-commutative aware adversarial environment. In order to help the user, we integrated into COLT simple and generic drivers for primitive and simple types such as Integers. These drivers, together with linearization points and the MUT, are given to the Bytecode instrumentor. The Bytecode instrumentor instruments the MUT and generates our linearizability checker using the library's non-commutativity specification. Then, the instrumented MUT is repeatedly executed until either a linearizability violation is found or a time bound is exceeded.

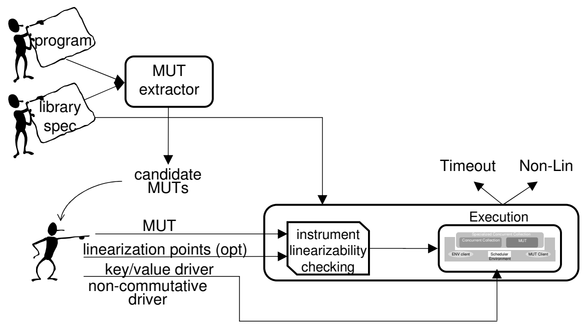


Figure 10. COLT overview

5.1.1 Checking Linearizability

COLT has three modes for checking linearizability and in general does not rely on user-specified linearization points. The first mode enumerates linearizations, the second enumerates candidate linearization points, and the third uses heuristics for guessing linearization points.

In our experiments, the third mode worked sufficiently well and we never resort to the more expensive modes or manually provided linearization points.

5.2 Applications

The 51 real-world applications shown in Table 1 were selected because their code is available and they make use of Java's concurrent collections. In many cases concurrent collections were introduced to address observed scalability problems, replacing `Collections.synchronizedMap` or manual locking of a sequential map.

Each of the applications contains one or more methods that were extracted and tested by COLT. These methods were extracted using COLT's `MUT Extractor` (discussed in Section 5.1). which identifies methods that may invoke multiple collection operations. In 18 out of the 95 extracted methods, COLT identified a specialized concurrent data structure operation inside a large method and we manually extracted and generated the MUT.

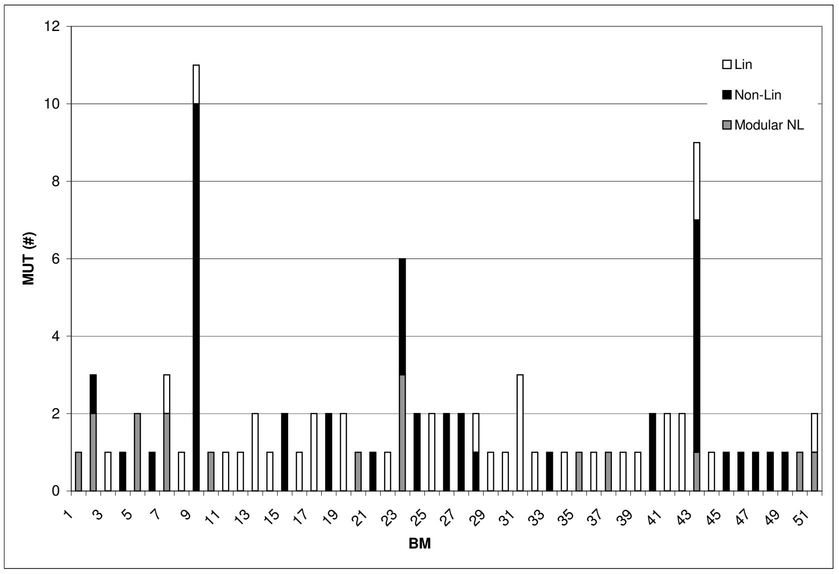
5.3 Results

We tested 95 methods in 51 applications. As Figure 11b shows 36 (38%) of these methods were linearizable in an open environment (and hence also linearizable in their program environment). Another 17 (18%) of these methods were not linearizable in an open environment, but were safe when implicit (and unchecked) application invariants were taken into account. Finally, 42 (44%) of these methods had atomicity violations that could be triggered in the current application.

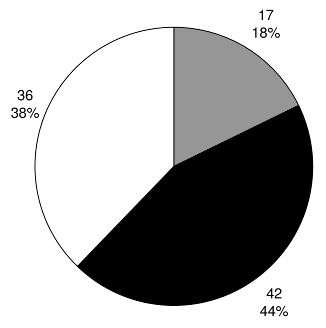
Figure 11a shows the results reported by COLT. The 'X' axis shows the application number from Table 1 and the 'Y' axis shows the number of MUTs checked for each application. "Non-Lin" shows the methods in our experiments that are not linearizable in an open environment as well as in the

#	Program	LOC	Description
1	Adaptive Planning	1,103,453	Automated budgeting tool
2	Adobe BlazeDS	180,822	Server-based Java remoting
3	Amf-serializer	4,553	AMF3 messages serializaton
4	Annsor	1,430	runtime annotation processor
5	Apache Cassandra	54,470	Distributed Database
6	Apache Derby	618,352	Relational database
7	Apache MyFaces Trinidad	201,130	JSF framework
8	Apache Struts	110,710	Java web applications framework
9	Apache Tomcat	165,266	Java Servlet and Server Pages
10	Apache Wicket	142,968	Web application framework
11	ApacheCXF	311,285	Services Framework
12	Autoandroid	19,764	Tools for automating android projects
13	Beanlib	42,693	Java Bean library
14	Carbonado	53,455	Java abstraction layer
15	CBB	16,934	Concurrent Building Blocks
16	cometdim	5,571	A web IM project
17	Direct Web Remoting	26,094	Ajax for Java
18	dyuproject	26,593	Java REST framework
19	Ehcache Annotations for Spring	3,184	Automatic integration of Ehcache in spring projects
20	Ektorp	6,261	Java API for CouchDB
21	EntityFS	79,820	OO file system API
22	eXo	13,298	Portal
23	fleXive	910,780	Java EE 5 content repository
24	GlassFish	260,461	JavaServer faces
25	Granite	28,932	Data services
26	gridkit	8,746	Kit of data grid tool and libs
27	GWTEventService	17,113	Remote event listening for GWT
28	Hazelcast	59,139	Data grid for Java
29	ifw2	54,888	Web application framework
30	JBoss AOP	1,013,073	Aspect oriented framework
31	Jetty	64,039	Java HTTP servlet server
32	Jexin	11,024	functional testing platform
33	JRipples	148,473	Program analysis
34	JSefa	27,208	Object serialization library
35	Jtell	5,402	Event collaboration library
36	keyczar	4,720	Cryptography Toolkit
37	memcache-client	4,884	Memcache client for Java
38	OpenEJB	191,918	Server
39	OpenJDK	1,634,818	JDK 7
40	P-GRADE	1,154,884	P-GRADE Grid Portal
41	Project Tammi	163,913	Java development framework
42	Project Track	5,160	Example application
43	RESTEasy	81,586	Java REST framework
44	Retrotranslator	27,315	Automatic compatibility tool
45	RoofTop	2,036,614	network/systems monitoring (NSM) tool
46	Tersus	165,160	Visual Programming Platform
47	torque-spring	2,526	Torque support classes
48	Vo Urp	24,996	UML data models translator
49	WebMill	57,161	CMS portal
50	Xbird	196,893	XQuery processor and XML db
51	Yasca	326,502	Program analysis tool

Table 1. Applications used for experiments



(a)



(b)

Figure 11. Benchmark results. (a) shows the MUT results for each benchmark. (b) shows the MUT distribution per results. As the graph shows, 42 MUTs are non linearizable in their current applications, 17 MUTs are non linearizable only in COLT's open environment, and 36 are linearizable in an open environment.

client environment. "Modular NL" shows the methods in our experiments that are not linearizable in an open environment but are linearizable in the client's current environment.

For example, application #9 is Apache Tomcat and has 10 non-linearizable methods in an open environment as well as the client environment and 1 linearizable method in any environment. For all of the non linearizable methods("Non-Lin" and "Modular NL"), COLT reported an interleaving in which the MUT is not atomic. In cases where we observed the method to be linearizable under a restricted environment (e.g., no `remove` operations), we confirmed that re-running COLT under an appropriately restricted adversary no longer reports the same violation.

We manually inspected all of the methods where COLT timed out without reporting a violation after 10 hours. We managed to construct a linearizability proof for each one of them, showing that: (i) these methods are indeed linearizable and COLT did not miss an error; and (ii) no false alarms were reported by COLT on these linearizable methods. These methods are presented in Figure 11a as "Lin".

Naive Adversary Testing with a naive adversary failed to find a single violation within a timeout of 10 hours per method, while COLT reported a violation for all of these non-linearizable methods in less than a second.

Confirmed Violations We reported all violations of linearizability (even those that were only present in an open environment) to the project developers. For each violation we included the interleaving and a suggested fix. Interestingly, in some cases, even though the tool reported errors that cannot occur in the program environment, the development team still decided to adopt our suggested fixes to make the code more robust to future program modifications.

Apache Cassandra has two methods that implement an optimistic concurrent algorithm on top of a concurrent collection. COLT reports a violation for both methods. The reason for the violation is that this algorithm may throw a null pointer exception when running concurrently with a remove operation. We reported this violation to the development team and it turned out that under the program environment, remove operations are allowed only on local copies of the collection. Therefore, in the current program environment the method is linearizable. However, the project lead decided to adopt our suggested fixes in order to make the method linearizable in any future evolution of the program.

Apache MyFaces Trinidad, **Ektrop**, **Hazelcast**, **Grid-Kit**, **GWTeventservice**, and **DYProject** use modules that try to atomically add and return a value from a collection or return a value if one is already in the collection for a given key. COLT reports a violation for all of these methods. For **Apache MyFaces Trinidad**, we reported the potential violation, but the developers had fixed this problem before we submitted the report. For **Ektrop**, the developers decided to keep our remarks in case they opt to make program modifications in the future. For **Hazelcast**, the developers acknowledged the violation and replied that the code is being re-factored. For **GWTeventservice**, the developers acknowledged the violations and adopted our fixes. For **GridKit**, the developers reported that the violation is not feasible in their environment. However, they still decided to adopt our fixes as a preventive measure. For **DYProject**, the developers acknowledged the violations and adopted our fixes.

In **Apache Tomcat** COLT found violations in 10 of the 11 methods checked. Two of the reported violations were approved as violations by the Tomcat development team. Most of the violations were caused by switching the implementation from a `HashMap` to a `ConcurrentHashMap` while removing the lock that guarded the `HashMap` in the collection client. Another type of violation occurs in `FastHttpDateFormat` and is caused by switching from a `synchronizedMap` to a `ConcurrentHashMap` without changing the client code. In this case, violations were introduced because the implementation of `synchronizedMap` guards each operation by the object's lock while the implementation of `ConcurrentHashMap` has internal locks which are different than the object lock. Therefore, two of the methods are built by a set of collection operations guarded by the collection's lock. These methods were linearizable under `synchronizedMap` and became non-linearizable under `ConcurrentHashMap`.

For the rest of the violated methods we reported the violations but have not yet heard from the development teams.

5.4 Conflict Serializability vs. Linearizability

None of the application methods we tested are conflict serializable [15]. The conflict serializability checker reports violation on all of our methods, whether or not the method behaves atomically. For the methods that required repair, none of the repairs were conflict serializable.

5.5 A Recurring Example: Memoization

A recurring operation in our benchmarks was memoization, in which a concurrent map was used to store the results of an expensive computation. The authors of `ConcurrentHashMap` explicitly avoided including this functionality, because the optimal design depends on the hit rate of the memoization table, the cost of the computation, the relative importance of latency versus throughput, potential interference between duplicate work, the possibility that the computation might fail, etc.

Consider a function `compute(K k)` that memoizes the result of `calculateVal(k)`. The desired functionality (note that Java does not actually have an `atomic` keyword) is:

```
V compute(K k) {
  atomic {
    V v = m.get(k);
    if (v == null) {
      v = computeVal(k);
      m.put(k, v);
    }
    return v;
  }
}
```

Figure 12 shows some of the implementations that we encountered for `compute`, including the buggy version from Figure 3a.

The procedure `compute(K k)` uses an underlying concurrent collection to memoize the value computed by function `calculateVal(k)`. When the value for a given key is cached in the collection it is returned immediately; when the value for a given key is not available, it is computed and inserted into the collection.

Figure 12(i) shows a linearizable concurrent implementation of `compute`. This operation is linearizable because it has only one collection operation, `putIfAbsent`, therefore, the linearizability guarantee is provided by the collection library. Even though this implementation is correct and linearizable, in most cases, programmers avoid writing the `compute` operation in this way because the internal implementation of `putIfAbsent` acquires a lock and this can be avoided in several cases. Another reason is that often `calculateData` can be time consuming or cause a side effect, and hence should not be executed more than once.

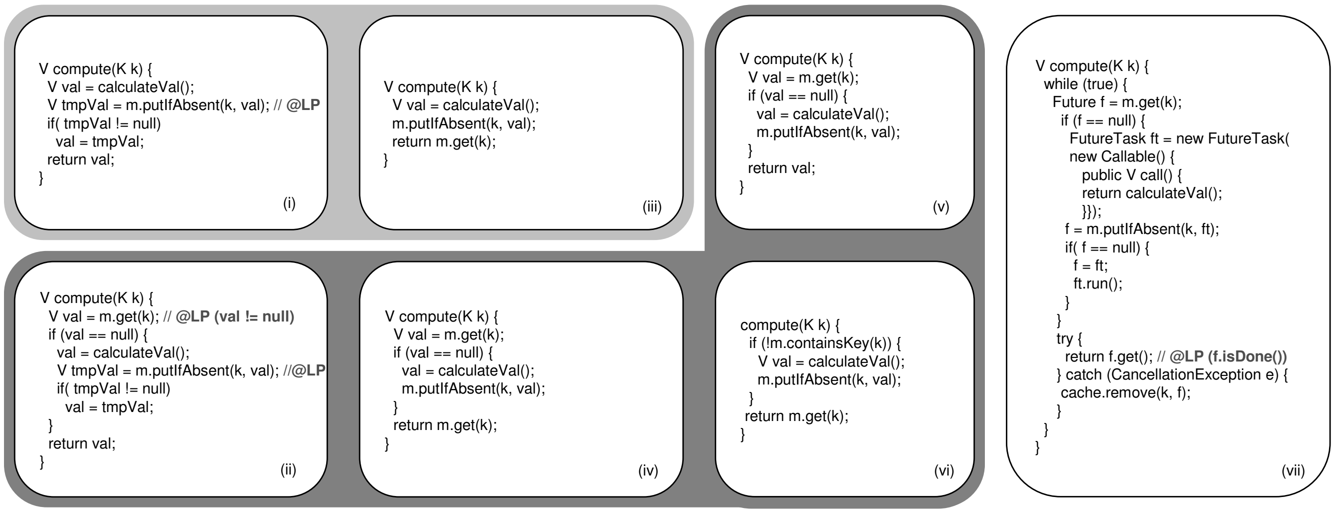
The implementation in Figure 12(ii) is mostly used to avoid lock acquisition when `k` is already inside the collection. This implementation is linearizable and has two conditional linearization points marked by `@LP [condition]`. The first conditional linearization point occurs when the `get` operation returns a value different than `null`. In this case, the value returned by `get` is returned without any additional access to `m`. The second linearization point occurs at the `putIfAbsent`. In cases where `putIfAbsent` succeeds in updating `m` a `null` value is returned by the `putIfAbsent` and the updated value is returned. Otherwise, `putIfAbsent` returns the value from `m` and this value is returned by `compute`.

The implementations in Figure 12(iv), Figure 12(v), and Figure 12(vi) are common non linearizable implementations of Figure 12(ii) and the implementation in Figure 12(iii) is a common non-linearizable implementation of Figure 12(i).

Figure 7 displays an interleaving that reveals the non-linearizability of the implementation in Figure 12(iv): the `remove(7)` operation at line 10 causes `compute(7)` to return a `null` result. A similar interleaving can reveal the non-linearizability of Figure 12(iii) and Figure 12(vi) where a `remove(k)` operation occurs between the `putIfAbsent(k)` and the `get(k)`.

An interleaving that reveals the non-linearizability of Figure 12(v) occurs when `put(k,*)` with the same key `k` is executed between `if (val == null)` and `putIfAbsent` of `compute`. In this case the `putIfAbsent` fails and the value returned by `compute` is not the value corresponding to `k` in the map `m`.

Advanced Example When the `calculateVal` function is either time consuming or causes a side effect the code in Figure 12(vii) is used. This implementation uses Java's `Future` construct to guarantee that only one execution of



```
V compute(K k) {
  V val = calculateVal();
  V tmpVal = m.putIfAbsent(k, val); // @LP
  if( tmpVal != null)
    val = tmpVal;
  return val;
}
                                    (i)
```

```
V compute(K k) {
  V val = calculateVal();
  m.putIfAbsent(k, val);
  return m.get(k);
}
                                    (iii)
```

```
V compute(K k) {
  V val = m.get(k);
  if (val == null) {
    val = calculateVal();
    m.putIfAbsent(k, val);
  }
  return val;
}
                                    (v)
```

```
V compute(K k) {
  while (true) {
    Future f = m.get(k);
      if (f == null) {
        FutureTask ft = new FutureTask(
        new Callable() {
           public V call() {
           return calculateVal();
           }});
       f = m.putIfAbsent(k, ft);
       if( f == null) {
         f = ft;
         ft.run();
       }
     }
     try {
       return f.get(); // @LP (f.isDone())
     } catch (CancellationException e) {
      cache.remove(k, f);
     }
   }
}
                                    (vii)
```

```
V compute(K k) {
  V val = m.get(k); // @LP (val != null)
  if (val == null) {
    val = calculateVal();
    V tmpVal = m.putIfAbsent(k, val); //@LP
    if( tmpVal != null)
      val = tmpVal;
  }
  return val;
}
                                    (ii)
```

```
V compute(K k) {
  V val = m.get(k);
  if (val == null) {
    val = calculateVal();
    m.putIfAbsent(k, val);
  }
  return m.get(k);
}
                                    (iv)
```

```
compute(K k) {
  if (!m.containsKey(k)) {
    V val = calculateVal();
    m.putIfAbsent(k, val);
  }
 return m.get(k);
}
                                    (vi)
```

Figure 12. Implementation types of compute, conditional linearization points for each linearizable implementation are marked with @LP [condition].

`calculateData` is done for each added key. This way only one thread is responsible for the calculation while the others may block until the calculation completes. If `Future` is canceled, the `Future` is removed from the map and a new memoization iteration continues until a `Future` terminates successfully and its value is returned. A linearization point for this implementation occurs when `f.get()` returns successfully (marked by @LP (f.isDone())) due to the fact that the `f` calculation terminated successfully. The reason this is a linearization point is that there might be a case where another thread canceled `f`'s execution and then the `compute` execution should continue.

Even though the implementation in Figure 12(vii) is linearizable, COLT reports a violation of linearizability. The problem is that COLT is not aware of the `Future` semantics and should treat a `Future` value that has not terminated successfully as if its corresponding key is missing from the map. Augmenting COLT with this information solves the problem and COLT no longer reports a linearizability violation in this case.

In this example, the `Future` computation does not itself perform collection operations and thus no special handling of the scheduling of futures is required, even if the Future is added to the collection (and such examples exist in practice). In general COLT warns the user if a Future performs collection operations, but we have seen no such examples in practice.

Benchmark Distribution Figure 13 shows the distribution of the different kinds of composed concurrent operations (as given in Figure 13) in our benchmark.

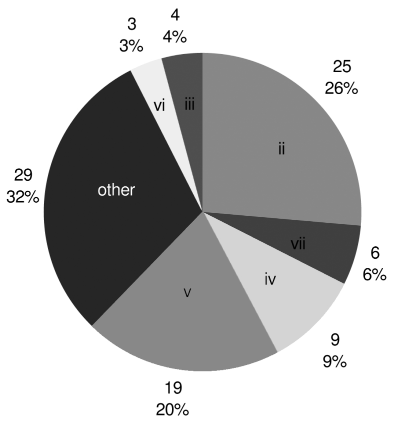


Figure 13. Distribution of our 95 MUTs of the types shown at Figure 12. Type (i) is missing because it is not a composed operation.

Overall 68% of the MUTs we checked were memoization examples. The most common bug pattern (20%) was the implementation in Figure 12(v) followed by the implementation in Figure 12(iv) (9%).

5.6 Reasons for Success

Source of Bugs most of the MUTs were written originally using a concurrent collection. However, in some cases, the MUT was modified while changing the collection implementation from sequential to concurrent. Refactoring the code in most cases resulted in a linearizability violation. A clear example is **Apache Tomcat** where 10 out of 11 MUTs were refactored in a non-linearizable manner.

Bugs Characteristics Observing the implementations in Figure 12 reveals that these implementations behave uniformly for each given key. This characteristic occurs in all our checked MUTs and implies that if the MUT is not linearizable then for each key there exists an interleaving revealing the MUT's non-linearizability.

The fact that programmers tend to write buggy specialized concurrent collections and that their corresponding MUTs are uniform significantly eases bug detection in real code. The uniformity characteristic implies that bugs can be detected using any input key so long as the adversary performs the right operation with the right interleaving. Our non-commutativity aware adversarial environment easily detects these non-linearizable interleavings.

6. Related Work

Partial Order Reduction Partial order reduction techniques, such as [18], use commutativity of individual memory operations to filter out execution paths that lead to the same global state. Indeed, our technique is a partial order reduction that uses a similar observation. However, our technique works on an encapsulated ADT and leverages commutativity at the level of the ADT, rather than the level of memory operations. A traditional partial order reduction technique working at the level of memory operations would not be able to leverage the commutativity specification at the level of ADT operations and would instead require exposing the implementation of ADT operations and checking whether individual memory operations within them commute. In contrast, our approach treats ADT operations as atomic operations and leverages commutativity of ADT operations to significantly increase the bug hunting capabilities of our tool.

Dynamic Atomicity Checking Dynamic atomicity checkers such as [14, 15] check for violations of conflict serializability. As noted earlier, conflict-serializability is inappropriate for concurrent data structures. Therefore, in this work we check for violations of linearizability. Vyrd [12] is a dynamic checking tool that checks a property similar to linearizability. Vyrd required manual annotation of linearization points. Line-Up [8] is a dynamic linearizability checker that enumerates schedules. The formal result of [13] implies that we need not generate schedules where linearizable operations are executed non-atomically. This can reduce the number of interleavings that need to be explored. This insight has also been discussed and made use of in the preemption sealing work of [6].

In contrast with COLT, all of these existing tools are non-modular and not directed using non-commutativity. As shown in Section 5, when commutativity information is not utilized, the ability to detect collection-related atomicity violations remains low. In fact, the poor results of the random adversary in Section 5 are obtained when underlying collection operations are considered atomic.

GAMBIT [10] is a unit testing tool for concurrent libraries built on top of the CHESS tool [25]. GAMBIT uses prioritized search of a stateless model checker using heuristics, bug patterns, and user-provided information. Even though GAMBIT and COLT are unit testing tools for concurrent libraries, COLT is working on a specialized concurrent data structure library built on top of an already verified and well defined concurrent collection. COLT leverages this fact together with non-commutativity to guide the scheduling to quickly reveal bugs. From the above reasons GAMBIT is unlikely to detect the collection-related atomicity violations.

An active testing technique for checking conflict serializabilty is presented in [26]. The technique uses bug patterns to control the scheduler directing the execution to error-prone execution paths. Even if the technique is adjusted to check linearizability, it is not modular and not directed using non-commutativity. Therefore, its ability to detect collection-related violations would likely be low.

Static Checking and Verification Some approaches [17, 24] do static verification and use the atomicity proof to simplify the correctness proofs of multithreaded programs. Our work also uses the linearizability proof of the collection library to prune non-violating paths. However, our technique is different than these works because we perform modular testing and check only methods. In addition, we use collection non-commutativity specifications to direct exploration. Moreover, we are checking the linearizability of the methods. Other work such as [28] focuses on model checking individual collections. Our work operates at a higher level, as a client of already verified collections and leverages the specifications of the underlying collections to reduce the search space. Shape analysis tools, such as [5], prove the linearizability of heap manipulating concurrent programs with a bounded number of threads. These tool are verifiers and may produce false alarms due to the overapproximating abstractions they employed. Our work is different than these works because we use dynamic analysis.

The idea of using a general client over-approximating the thread environment is common in modular verification. Previous work represented the environment as invariants [22] or relations [23] on the shared state. This idea has also been used early on for automatic compositional verification [9]. In addition, this approach has led to the notion of thread-modular verification for model checking systems with infinitely-many threads [16] and has also been applied to the domain of heap-manipulating programs with coarse-grained concurrency [20]. COLT uses the idea of an adversarial environment for dynamic checking, leveraging non-commutativity information to effectively reveal atomicity violations.

7. Conclusions

We have presented a modular and effective technique for testing composed concurrent operations. Our technique uses a specialized adversary that leverages commutativity information of the underlying collections to guide execution towards linearizability violations. We implemented our technique in a tool called COLT and showed its effectiveness by detecting 56 previously unknown linearizability violations in 51 popular open-source applications such as Apache Tomcat.

Acknowledgments

We thank Karen Yorav for reading a preliminary version of this paper and making valuable remarks. This research was partially supported by The Israeli Science Foundation (grant no. 965/10), NSF Grant CCF-0702681, and a gift from IBM.

References

[1] Amino concurrent building blocks. http://amino-cbbs.sourceforge.net/.

[2] Apache tomcat. http://tomcat.apache.org/.

[3] Intel thread building blocks. http://www.threadingbuildingblocks.org/.

[4] openjdk. http://hg.openjdk.java.net/jdk7/jaxp/jdk.

[5] AMIT, D., RINETZKY, N., REPS, T. W., SAGIV, M., AND YAHAV, E. Comparison under abstraction for verifying linearizability. In *CAV* (2007), pp. 477–490.

[6] BALL, T., BURCKHARDT, S., COONS, K. E., MUSUVATHI, M., AND QADEER, S. Preemption sealing for efficient concurrency testing. In *TACAS* (2010), pp. 420–434.

[7] BERNSTEIN, P. A., HADZILACOS, V., AND GOODMAN, N. *Concurrency Control and Recovery in Database Systems*. 1987.

[8] BURCKHARDT, S., DERN, C., MUSUVATHI, M., AND TAN, R. Line-up: a complete and automatic linearizability checker. In *PLDI* (2010), pp. 330–340.

[9] CLARKE, JR., E. Synthesis of resource invariants for concurrent programs. *TOPLAS 2*, 3 (1980), 338–358.

[10] COONS, K. E., BURCKHARDT, S., AND MUSUVATHI, M. GAMBIT: effective unit testing for concurrency libraries. In *PPOPP* (2010), pp. 15–24.

[11] DOHERTY, S., DETLEFS, D. L., GROVES, L., FLOOD, C. H., LUCHANGCO, V., MARTIN, P. A., MOIR, M., SHAVIT, N., AND GUY L. STEELE, J. DCAS is not a silver bullet for nonblocking algorithm design. In *SPAA* (2004), pp. 216–224.

[12] ELMAS, T., TASIRAN, S., AND QADEER, S. Vyrd: verifying concurrent programs by runtime refinement-violation detection. In *PLDI* (2005), pp. 27–37.

[13] FILIPOVIĆ, I., O'HEARN, P., RINETZKY, N., AND YANG, H. Abstraction for concurrent objects. In *ESOP* (2009), pp. 252–266.

[14] FLANAGAN, C., AND FREUND, S. N. Atomizer: A dynamic atomicity checker for multithreaded programs. In *POPL* (2004), pp. 256–267.

[15] FLANAGAN, C., FREUND, S. N., AND YI, J. Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. In *PLDI* (2008), pp. 293–303.

[16] FLANAGAN, C., AND QADEER, S. Thread-modular model checking. In *SPIN* (2003), pp. 213–224.

[17] FLANAGAN, C., AND QADEER, S. A type and effect systrm for atomicity. In *PLDI* (2003), pp. 338–349.

[18] GODEFROID, P. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.

[19] GOETZ, B., PEIERLS, T., BLOCH, J., BOWBEER, J., HOLMES, D., AND LEA, D. *Java Concurrency in Practice*. Addison Wesley, 2006.

[20] GOTSMAN, A., BERDINE, J., COOK, B., AND SAGIV, M. Thread-modular shape analysis. In *PLDI* (2007), pp. 266–277.

[21] HERLIHY, M. P., AND WING, J. M. Linearizability: a correctness condition for concurrent objects. *TOPLAS 12*, 3 (1990).

[22] HOARE, C. A. R. *Towards a theory of parallel programming*. 1972.

[23] JONES, C. B. *Specification and design of (parallel) programs*. 1983.

[24] MUSUVATHI, M., AND QADEER, S. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI* (2007), pp. 446–455.

[25] MUSUVATHI, M., QADEER, S., BALL, T., BASLER, G., NAINAR, P. A., AND NEAMTIU, I. Finding and reproducing heisenbugs in concurrent programs. In *OSDI* (2008), pp. 267–280.

[26] PARK, C.-S., AND SEN, K. Randomized active atomicity violation detection in concurrent programs. In *SIGSOFT* (2008), pp. 135–145.

[27] SHAVIT, N. Data structures in the multicore age. *Commun. ACM 54* (March 2011), 76–84.

[28] VECHEV, M., YAHAV, E., AND YORSH, G. Experience with model checking linearizability. In *SPIN* (2009), pp. 261–278.

[29] WEIHL, W. E. Commutativity-based concurrency control for abstract data types. *IEEE Trans. Computers 37*, 12 (1988), 1488–1505.