VERIFYING ATOMICITY OF COMPOSED CONCURRENT OPERATIONS

by

Ohad Shacham

under the supervision of Prof. Mooly Sagiv
and the consultation of Dr. Eran Yahav

A thesis submitted
for the degree of Doctor of Philosophy

Submitted to the Senate of Tel-Aviv University
August 2012
To my loved ones, Anita, Guy, and Tomer.
Abstract

Verifying Atomicity of Composed Concurrent Operations

Ohad Shacham
School of Computer Science
Tel-Aviv University

Modern programming languages such as Java and C# provide efficient concurrent library implementations that allow programmers to write scalable applications. Applications using the library often need to combine several library operations into a single atomic action. While the library interface can be extended to support certain common composed operations, the number of composed operations is in general unbounded, and programmers are often required to introduce their own composed operations. Unfortunately, composing library operations without hindering their performance is tricky and error-prone.

In this thesis we present a practical technique for automatically verifying the atomicity of composed concurrent operations. We conducted an empirical study and found that the vast majority of composed operations are based on concurrent maps. Therefore, our approach focuses on the common case of concurrent maps. The approach utilizes two techniques. The first is a bug hunting technique which performs modular testing of client code in an adversarial environment; we use a specification which is similar to non-commutativity to drastically reduce the number of executions explored to detect a bug. We rely on the fact that the linearization points for composed operation which updates the state is uniquely determined as the operation of the base collection which updates the state and show that this technique cannot miss violations. However, this technique alone, although very efficient in detecting atomicity violations, cannot prove their absence, due to the unbounded number of keys and values.

The second technique bounds the number of keys and values that must be explored. The main observation behind this approach is that many composed concurrent operations are data-independent. That is, the control-flow of the composed operation does not depend on specific input values. While verifying data-independence is undecidable in the general case, we provide simple sufficient conditions that can be used to establish a composed operation as data-independent. We show that for the common case of concurrent maps, data-independence reduces the hard problem of verifying linearizability into a verification problem that can be solved efficiently and has a bounded number of keys and values.
We have implemented our approach in a tool called COLT and evaluated it on all composed operations from 57 real-world applications (112 composed operations). COLT detected 59 linearizability violations in Apache Tomcat, Cassandra, MyFaces Trinidad, and other real-world applications and showed that many composed operations (51 out of 112) are data-independent. Out of these 51 data-independent operations, COLT automatically verified 31 of them as linearizable and the other 20 as having linearization bugs that could be repaired and then automatically verified. We examined the other 61 operations and showed that they are not linearizable, thus indicating that in most of the cases data independence does not limit the expressiveness of writing realistic linearizable composed operations.
Acknowledgements

First and foremost I offer my sincerest gratitude to my advisors, Mooly Sagiv and Eran Yahav. Their energy, inspiration, knowledge, and patience were crucial for the completion of this thesis.

I would like to thank Martin Vechev for his guidance during this thesis. His advice was always beneficial and working with him was a real privilege.

I would like to thank Alex Aiken for his guidance during this thesis and for the enjoyable visits at Stanford University.

I would like to thank Nathan Bronson for his guidance during this thesis.

I would like to thank John Field, Eran Yahav, and Martin Vechev for the enjoyable summer in IBM T.J. Watson Research Lab.

I would like to thank Guy Golan-Gueta, Omer Tripp, Ghila Castelnuovo, Roman Manevich, Greta Yorsh, Noam Rinetzky, and Ariel Jarovsky for many fruitful discussions that made this work better.

I would also like to thank Karen Yorav for reviewing a few chapters of this thesis.
# Contents

## 1 Introduction

1.1 Custom Concurrent Data Structures .................................................. 1

1.2 Checking Linearizability ................................................................. 2
  1.2.1 Modular Checking of Linearizability .............................................. 3
  1.2.2 Adversarial Execution Guided by Influence Specification ................. 4
  1.2.3 Pruning Violating Executions .................................................... 4
  1.2.4 Verifying Linearizability of Data-Independent Composed Concurrent Operations 5

1.3 Evaluation ....................................................................................... 5

1.4 Thesis Contributions ......................................................................... 6

## 2 Motivation

2.1 Testing Linearizability of Composed Concurrent Operations ................... 9
  2.1.1 Apache Tomcat Motivating Example .............................................. 9
  2.1.2 Composed Operations Extraction ................................................. 12
  2.1.3 Modular Checking of Linearizability .............................................. 12
  2.1.4 Adversarial Execution Guided by Influence .................................... 12
  2.1.5 Pruning Violating Executions ..................................................... 13

2.2 Verifying Linearizability of Data-Independent Composed Concurrent Operations .. 14
  2.2.1 Apache ServiceMix Motivating Example ...................................... 14
  2.2.2 Data Independent Composed Operations ...................................... 15

## 3 Preliminaries

3.1 Language ....................................................................................... 17

3.2 Concurrent Collection and Composed Operations .................................. 18

3.3 Histories ....................................................................................... 18

3.4 Linearizability ............................................................................... 21

3.5 Checking Linearizability ................................................................... 21
## 4 Reducing the Number of Threads

## 5 Thread-Centric Linearizability Testing

5.1 Influence Guided Environment .............................................. 31
5.2 Exploration Procedure .......................................................... 32
  5.2.1 Transition Relation .......................................................... 32
  5.2.2 Traces and Exploration Procedure ...................................... 33
5.3 Example .................................................................................. 34
5.4 Pruning Violating Traces ......................................................... 35
  5.4.1 Non-Encapsulated Composed Operations ............................... 35
  5.4.2 Type State Based Interface ............................................... 37

## 6 Thread-Centric Correctness Guarantees

6.1 Influence History .................................................................. 39
6.2 Traces and Influence Histories ................................................ 43
  6.2.1 History Generation ............................................................ 44
  6.2.2 Trace and History Correlation .......................................... 44
6.3 Testing vs. Verification ........................................................... 46

## 7 Data-Independent Operations

7.1 Data Independence ................................................................ 47
  7.1.1 Rule Based Data Independence ......................................... 48
  7.1.2 Example ........................................................................... 53
7.2 Composed Operations Classes .............................................. 55
  7.2.1 Singleton Collection Method ............................................ 55
  7.2.2 Fixed Collection Method ................................................... 55
  7.2.3 Value Collection Method .................................................. 57
  7.2.4 Example of a Data-Dependent Method ............................... 57

## 8 Experimental Evaluation

8.1 Implementation ..................................................................... 59
8.2 Applications ........................................................................... 60
8.3 Results .................................................................................... 62
  8.3.1 Linearizability Violations .................................................. 65
  8.3.2 Benchmark Classification ................................................ 68
8.4 Conflict Serializability vs. Linearizability ................................ 71
8.5 A Recurring Example: Memoization ..................................... 71
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.5.1</td>
<td>Advanced Example</td>
<td>73</td>
</tr>
<tr>
<td>8.5.2</td>
<td>Benchmark Distribution</td>
<td>74</td>
</tr>
<tr>
<td>8.6</td>
<td>Reasons for Success</td>
<td>74</td>
</tr>
<tr>
<td>8.6.1</td>
<td>Source of Bugs</td>
<td>74</td>
</tr>
<tr>
<td>8.6.2</td>
<td>Bug Characteristics</td>
<td>74</td>
</tr>
<tr>
<td>9</td>
<td>Related Work</td>
<td>77</td>
</tr>
<tr>
<td>9.1</td>
<td>Dynamic Atomicity Checking</td>
<td>77</td>
</tr>
<tr>
<td>9.2</td>
<td>Static Checking and Verification</td>
<td>78</td>
</tr>
<tr>
<td>9.3</td>
<td>Effective Techniques for State Space Reduction</td>
<td>79</td>
</tr>
<tr>
<td>9.3.1</td>
<td>Partial Order Reduction</td>
<td>79</td>
</tr>
<tr>
<td>9.3.2</td>
<td>Thread-Centric Approach</td>
<td>79</td>
</tr>
<tr>
<td>10</td>
<td>Future Work</td>
<td>81</td>
</tr>
<tr>
<td>11</td>
<td>Conclusion</td>
<td>83</td>
</tr>
<tr>
<td></td>
<td>Bibliography</td>
<td>84</td>
</tr>
<tr>
<td>A</td>
<td>Chameleon: Adaptive Selection of Collections</td>
<td>91</td>
</tr>
<tr>
<td>A.1</td>
<td>Introduction</td>
<td>91</td>
</tr>
<tr>
<td>A.1.1</td>
<td>Main Contributions</td>
<td>92</td>
</tr>
<tr>
<td>A.2</td>
<td>Overview</td>
<td>93</td>
</tr>
<tr>
<td>A.2.1</td>
<td>Motivating Example</td>
<td>93</td>
</tr>
<tr>
<td>A.2.2</td>
<td>Tradeoffs in Collection Implementations</td>
<td>97</td>
</tr>
<tr>
<td>A.2.3</td>
<td>Possible Solutions for Low Utilization</td>
<td>98</td>
</tr>
<tr>
<td>A.3</td>
<td>Automated Collection Selection</td>
<td>99</td>
</tr>
<tr>
<td>A.3.1</td>
<td>Optimal Selection of Collection Implementations</td>
<td>99</td>
</tr>
<tr>
<td>A.3.2</td>
<td>Semantic Collections Profiling</td>
<td>100</td>
</tr>
<tr>
<td>A.3.3</td>
<td>Rule Engine</td>
<td>102</td>
</tr>
<tr>
<td>A.4</td>
<td>Implementation</td>
<td>105</td>
</tr>
<tr>
<td>A.4.1</td>
<td>Design Choices</td>
<td>106</td>
</tr>
<tr>
<td>A.4.2</td>
<td>Library Architecture</td>
<td>106</td>
</tr>
<tr>
<td>A.4.3</td>
<td>VM Support</td>
<td>109</td>
</tr>
<tr>
<td>A.4.4</td>
<td>Discussion</td>
<td>111</td>
</tr>
<tr>
<td>A.5</td>
<td>Experimental Results</td>
<td>111</td>
</tr>
<tr>
<td>A.5.1</td>
<td>Benchmarks</td>
<td>111</td>
</tr>
<tr>
<td>A.5.2</td>
<td>Methodology</td>
<td>112</td>
</tr>
</tbody>
</table>
List of Tables

3.1 Sequential specification of a map. $M : \mathbb{N} \rightarrow \mathbb{N}$ denotes the content of the map. 19

5.1 Influence specification for a map. For simplicity, we assume that $v_1$ and $v_2$ are non-null. 32

5.2 Sequential specification of a map where our technique may miss all linearizability violations. $M : \mathbb{N} \rightarrow \mathbb{N}$ denotes the content of the map and $cnt$ denotes an integer initialized by 0. 38

7.1 Sample composed operations and their specifications. $compute$ is a data-independent composed operation and $Replace$ is a data-dependent operation. $M : \mathbb{N} \rightarrow \mathbb{N}$ denotes the content of the map. 50

8.1 Applications used for experiments 60

A.1 Heap and trace statistics for each execution. Information is aggregated per allocation context. 101

A.2 Example of built-in rules. 104

A.3 Statistics gathered on every garbage collection cycle for each allocation context 110
List of Figures

1.1 A correct concurrent increment operation for a map of concurrent counters, implemented using ConcurrentHashMap operations (from OpenJDK [15]). .......................... 2

1.2 Linearizable version of a composed operation taken from Apache Tomcat [4]. This operation is not supported by the enriched Java library [7]. ...................... 3

2.1 (a) Erroneous code from Apache Tomcat [4] 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. ................................. 10

2.2 (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. ................................................................. 11

2.3 A linearizable data-independent composed operation taken from Apache ServiceMix [3] 14

3.1 A syntax of a core language for composed operations .................. 17

3.2 Histories for the custom map builds by getLock from Figure 2.3. ....... 20

4.1 An example with two writes. (a) shows a method twoWrites that writes value 4 twice to the same key and returns it. (b) shows a non-linearizable history involving twoWrites. ................................................................. 25

5.1 An example of a non-linearizable method inspired by bugs from Adobe BlazeDS, Vo Urp and Ehcache-spring-annotations. ............................... 35
5.2 A trace of the doubleNonLin method from Figure 5.1 showing a linearizability violation. The violation is revealed by the different return value (val) at the end of the sequential trace. A state is represented by a box, a linearization point is represented by a black box, a state of the sequential run is represented by a circle, an arrow represents a transition, and a dashed arrow represents an environment transition together with its operation.

5.3 Non-encapsulated example with a global variable update.

5.4 A composed operation built using the interface from Table 5.2.

6.1 Influence and non-influence histories for the custom map builds by getLock from Figure 2.3.

7.1 Histories for the custom map built by the data-independent composed operation getLock from Figure 2.3.

7.2 A possible linearizable fix for the example from Figure 5.1. This examples is an SCM.

7.3 The traces generated by our algorithm, for doubleLin method from Figure 7.2. A state is represented by a box, a linearization point is represented by a black box, a state of the sequential run is represented by a circle, an arrow represents a transition, and a dashed arrow represents an environment transition together with its operation.

7.4 An example, taken from AutoAndroid [5] which is an FCM and not an SCM.

7.5 VCM example, taken from OpenJDK [15].

7.6 A data-dependent method example, taken from fleXive [8].

8.1 COLT overview

8.2 Classification of 112 benchmark methods into SCM, VCM, FCM, and data dependent methods, and classification into linearizable and non-linearizable.

8.3 Methods, out of the 112, that are either linearizable or can be fixed to be linearizable in each one of our benchmarks.

8.4 The data-dependent getInstance method taken from fleXive, fixed to be linearizable.

8.5 Composed operations distribution per result. As the graph shows, 42 composed operations are non-linearizable in their current applications and 17 composed operations are non-linearizable only in an open environment.

8.6 Linearizability violations for each benchmark.

8.7 Classification of our 112 data-dependent and data-independent composed operations according to the rules they falsify or satisfy.

8.8 A non-linearizable composed operation from Granite. The result of the operation depends on the global variables _proxy and _noProxy.
8.9 The desired functionality of a function \texttt{compute(K k)} that memoizes the result of \texttt{calculateVal(k)} (note that Java does not actually have an atomic keyword) .

8.10 Implementation types of \texttt{compute}. Conditional linearization points for each linearizable implementation are marked with \texttt{@LP[condition]}. 

8.11 The distribution of our 112 composed operations of the types shown in Figure 8.10. Type (i) is missing because it is not a composed operation.

A.1 CHAMELEON overview

A.2 Percentage of live data consumed by collections in \texttt{tvla} running on a benchmark. The “X” axis shows the GC cycle and the “Y” axis shows the percentage of the live data.

A.3 Combined results for top 4 allocation contexts in \texttt{tvla}.

A.4 Simple language for implementation selection rules.

A.5 Library architecture. Shaded fields are updated by the VM.

A.6 Minimal heap size required to run the benchmark after applying fixes suggested by CHAMELEON, shown as percentage of the original minimal heap size.

A.7 Running times of the benchmarks after applying fixes suggested by CHAMELEON, shown as percentage of the original running time. Running times were obtained by running each benchmark with its corresponding original minimal-heap size.

A.8 Percentage of collections in original version of bloat
Chapter 1

Introduction

Concurrent data structures are now critical components of many systems [67] but are notoriously hard to get right (e.g., [35]). To shield programmers from the complexity of concurrent data structures, modern languages hide their implementations in libraries (e.g., [1, 11, 44]). Data structures provided by the library usually provide guarantee that their operations are linearizable. That is, a concurrent object operation has the same effect as if it were performed sequentially; Linearizability is the weakest notion that guarantees observational equivalence [38].

1.1 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 library operations. The main challenge when composing linearizable operations is to guarantee that the composed operation is also linearizable.

It is important to note that programmers usually compose the operations of underlying concurrent data structures without using a wrapping lock, because guaranteeing an atomic behavior 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 1.1, taken from OpenJDK [15], implements an increment operation for a concurrent histogram (as a map of counters). The histogram is implemented in UncaughtExceptions which is a subclass of class ConcurrentHashMap; other map operations (e.g., remove, get, put) can be performed directly, and concurrently, on the histogram by client code. The increment operation does not use any locks (for the reasons mentioned above), and instead guarantees atomicity by means of an optimistic concurrency loop, using the (atomic) operations putIfAbsent and replace provided by the underlying ConcurrentHashMap. While the code in Figure 1.1 does
Figure 1.1: A correct concurrent increment operation for a map of concurrent counters, implemented using ConcurrentHashMap operations (from OpenJDK [15]).

Guaranteeing atomicity, we have found that many other open source clients combine atomic operations of underlying concurrent data structures in a non-atomic way. Figure 2.1a in Chapter 2 shows such an example.

1.2 Checking Linearizability

Given the difficulty of writing composed atomic operations, it is desirable to provide programmers with an automatic technique for checking linearizability.

In general, automatically checking that a concurrent object is linearizable is a difficult task: it is known that the problem is EXPSPACE-hard even for finite systems [18]. In contrast, other popular correctness criteria, such as conflict-serializability, can be checked more efficiently [40]. Unfortunately, such conditions are too restrictive and reject virtually all real-world concurrent objects in use, as these objects are not conflict-serializable. For example, the increment code of Figure 1.1 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 Chapter 8, we show that for all of our benchmarks, correct methods would have been rejected by a conflict-serializability checker.)

Instead of verifying the linearizability of composed operations in client code, one can extend the library interface to support additional operations. Indeed, the technique presented in this thesis [64, 65] influenced the interface of Java concurrent collections, which is being enriched to avoid some of
1.2. Checking Linearizability

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

Figure 1.2: Linearizable version of a composed operation taken from Apache Tomcat [4]. This operation is not supported by the enriched Java library [7].

these errors [7, 13]. While the library interface can be extended to support certain common composed operations, the number of composed operations is in general unbounded, and programmers are often requires to introduce their own composed operations. For example, Figure 1.2 shows a linearizable version of a composed operation taken from Apache Tomcat [4] that is not supported by the enriched library. Our tool COLT presented in this thesis verified that this method is indeed linearizable.

The Challenge Given an application, our goal is to verify that its composed concurrent operations are linearizable or to show that they might violate linearizability.

Since linearizability violations often depend on specific thread configurations and schedules, very few program executions might exhibit the violation. 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) restricting execution by interleaving only influence collection operations, where influence is a property stronger than non-commutativity.

Verifying linearizability is undecidable in general and required an unbounded number of keys and values. COLT addresses this challenge by: (i) identifying a class of data-independent operations. (ii) showing that for the common case of concurrent maps, data-independence reduces the hard problem of verifying linearizability into a verification problem that can be solved efficiently with a bounded number of keys and values.

1.2.1 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 composed operations. This allows us to check the lin-
linearizability of the composed operations in an adversarial environment rather than trying to reproduce adversarial testing conditions within the original application.

However, since COLT covers interleavings that may not occur for the composed operation in the context of its application, it might produce false alarms in cases where application-specific invariants create a restricted environment in which the composed operation is actually linearizable. Fortunately, we found that such situations are rare, and on real programs COLT has a very low false alarm rate. Intuition tells us that this situation arises because operations can usually be executed in an arbitrary order without violating linearizability. Furthermore, checking the code for linearizability against all possible environments guarantees that the client 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 code modifications might trigger these latent problems.

1.2.2 Adversarial Execution Guided by Influence Specification

Our goal is to check the linearizability of a given composed operation. We would like to avoid exploring executions that result in collection values already observed by the composed operation in earlier executions. COLT aims for every newly explored execution path to yield a new collection result for the composed operation. This can be viewed as a kind of partial order reduction (e.g., [43]) where non-commutativity is checked at the level of atomic collection operations.

Using an adversarial execution guided by influence specification make COLT an efficient tool for finding linearizability violations of composed concurrent operations. Moreover, using the influence specification bounds the number of base operations by the number of base operations performed by the composed operation during an execution. However, the number of executions that need to be explored remains unbounded, and in general COLT cannot verify the linearizability of a given composed operation.

1.2.3 Pruning Violating Executions

We conducted an empirical study using code search engines [9, 14] and found that composed operations that use concurrent maps are most common. The empirical study was done by searching for Java’s concurrent collection types, such as ConcurrentHashMap and PriorityBlockingQueue. Moreover, we augmented the search by utilizing Java concurrent collections unique tokens, such as replace and putIfAbsent, which significantly helped identifying composed operations. From the above reason, we show that for the common case of concurrent maps, our technique does not prune all violating executions. The proof relies on the fact that the linearization points for operations of custom collections
which updates the state (via base collection operations) is uniquely determined as the operation of the base collection which updates the state. This result is a consequence of encapsulation of the underlying collection. We also show that a composed operation with two updating operations is non-linearizable.

Even though our testing approach does not prune all violating executions, it cannot prove the linearizability of a composed operation. This results due to the unbounded number of keys and values that must be explored.

1.2.4 Verifying Linearizability of Data-Independent Composed Concurrent Operations

The main observation behind our verification approach is that many composed concurrent operations are data-independent. That is, the control-flow of the composed operation does not depend on specific input values. While verifying data-independence is undecidable in the general case, we provide simple sufficient conditions that can be used to establish that a composed operation is data-independent. The main idea behind these rules is to restrict the inputs to the composed operation, and the control flow inside the composed operation, to guarantee that it treats all keys uniformly. Influenced by the results of our empirical study, in which all the discovered composed operations use concurrent maps, we show that for the common case, data-independence reduces the hard problem of verifying linearizability into a verification problem that can be solved efficiently with a bounded number of keys and values. Using data-independence, we can verify linearizability by applying our testing approach with at most one key and two values.

1.3 Evaluation

We use a simple static analysis tool to extract composed operations from applications. Our linearizability verifier is implemented in Promela, and verifies the extracted operations using the SPIN model checker [49]; our linearizability bug hunter is implemented using bytecode instrumentation.

We extracted all 112 composed collection operations from 57 applications and checked which of them satisfy the data-independence requirement (it is straightforward to implement a frontend that checks these syntactic restrictions).

We found that 61 (54%) of these operations are data-dependent and cannot be proven correct using our technique. However, a close inspection revealed that these 61 methods are in fact non-linearizable! Of the 51 methods that can be handled by our technique, 50 are data-independent and 1 depends on fixed input keys.

Spin checked the extracted models for these 51 operations in less than a second each, verifying 31 methods as correct and identifying a linearizability violation in the remaining 20 methods. Then, we manually fixed the linearizability violations in these 20 methods. After the repair, our tool successfully proved all of the repaired methods linearizable. For the data-independent operations, we ran our testing
tool and values and found 39 additional linearizability violations in less than a second each. These violations are in addition to the 20 violations identified by Spin.

1.4 Thesis Contributions

The main contributions of this thesis are:

- We define the problem of verifying linearizability for composed concurrent operations.
- We present an approach for modular testing and verification of linearizability for composed concurrent operations.
- We use collection influence specifications to direct exploration. This significantly increases the bug hunting capabilities of our testing tool, as confirmed by our experimental results in Chapter 8.
- We prove that the linearization points for composed operation built using the interface from Table 3.1 which updates the state (via base collection operations) is uniquely determined as the operation of the base collection which updates the state.
- We prove the soundness of our approach showing that it can find a violation for any non-linearizable composed operation built using the interface from Table 3.1.
- We define a notion of data-independence for composed operations. This notion is inspired by Wolper [75].
- We define a set of restrictions that can be easily checked and guarantee that a composed operation is data-independent.
- We conduct a thorough empirical study showing that many realistic composed operations are indeed data-independent and all the data-dependent composed operations we identified are non-linearizable.
- We prove that for data-independent composed operations linearizability, can be checked using a single key and two values. In particular, for programs with finite local states, verifying linearizability becomes decidable.
- We demonstrate our approach in the context of Java concurrent collections. We implemented a hybrid dynamic and static based tool called COLT that checks the linearizability of composed concurrent collection operations.
- We show that in practice COLT is effective in detecting real bugs while maintaining a very low rate of false alarms. Using COLT, we were able to identify 59 linearizability violations in Apache
1.4. Thesis Contributions

Tomcat, Apache Cassandra, Apache MyFaces Trinidad, Adobe BlazeDS, as well as in other real-life applications. Some of these violations expose real bugs, while others represent potential bugs in future client implementations.

- We show that in practice COLT is effective in proving linearizability for real-life data-independent composed concurrent operation by proving the linearizability of 30 methods (in a second each).

In addition to our work on verifying atomicity of composed concurrent operations, we present, in Appendix A, another tool that works on collections. The tool called CHAMELEON [66], automatically selects the appropriate collection implementations for a given application. CHAMELEON uses what we call semantic profiling together with a set of collection selection rules to make an informed choice about collection implementation. This approach is markedly different from existing profiling tools where the user is forced to manually filter massive amounts of irrelevant data, typically offline, in order to make an educated guess. Consequently, using CHAMELEON significantly reduces the profiling time because CHAMELEON directs the user to specific allocation contexts where objects allocated at these contexts caused bloat. We presented this work in PLDI 2009 [66] as well as in the PLDI 2009 ACM Student Research Competition and won the 2nd place.
CHAPTER 1. INTRODUCTION
Chapter 2

Motivation

2.1 Testing Linearizability of Composed Concurrent Operations

2.1.1 Apache Tomcat Motivating Example

Figure 2.1a shows a composed operation taken from Apache Tomcat [4] version 5. In this version `attr` is a sequential `HashMap`. The atomicity of the composed operation is guaranteed by wrapping the composed operation with a `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, as a consequence, the programmers decided to remove the `synchronized(attr)` blocks from the program as well as from line 3 of Figure 2.1a. This caused a bug which was identified by COLT.

Hard to cover, rarely-executed traces Figure 2.1b 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; but also that the collection operations must all use same key “A” as well. This composed operation is not linearizable and COLT successfully detects the violation. When we reported this bug to Tomcat’s developers as well as the fix in Figure 2.1c, 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.

We now briefly describe the main ideas behind how COLT tests for this violations.
1. **getAttribute** 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)

1. **getAttribute** removeAttribute(String name) {
2.     Attribute val = attr.get(name); // @LP val == null
3.     if (val != null) {
4.         val = attr.remove(name); // @LP
5.     }
6.    return val;
7. }
8. }

(b)

(c)

Figure 2.1: (a) Erroneous code from Apache Tomcat [4] 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.
2.1. Testing Linearizability of Composed Concurrent Operations

```java
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)

Figure 2.2: (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.
2.1.2 Composed Operations Extraction

COLT includes a semi-automatic technique that greatly assists 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 programmer writes a composed operation inside a large method instead of allocating a new method for the composed operation. In this case, the static analysis identifies the large method and the user must manually extract and generate the composed operation.

2.1.3 Modular Checking of Linearizability

COLT checks linearizability [47] in a modular way by invoking one composed operation at a time in an environment that performs arbitrary collection operations concurrently.

2.1.4 Adversarial Execution Guided by Influence

Due to the rarity of many linearizability violations, a dynamic tool for detecting them 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 all those that lead to linearizability violations. As a first step, we assume that the underlying collection implementation is linearizable; thus, the internal representation of the collection can be abstracted away and collection operations can be assumed to execute atomically. As in [21], this allows us to only consider interleavings at the level of collection operations, without considering interleavings of their internal operations. However, the key insight that we use is different: before and after each collection operation $op_1$ executed by the composed operation, the environment chooses a collection operation $op_2$ that either influences or is influenced by $op_1$. Note that in general scheduling operations before and after every collection operation of the composed operation is insufficient, as this strategy can omit interleavings that lead to linearizability violations. In Section 5.4 we show examples where scheduling operations before and after every collection operation of the composed operation omits all interleavings that lead to linearizability violations. However, our technique does not prune all interleavings that lead to linearizability violations for encapsulated composed operations build by the map semantics from Table 3.1.

Figure 2.2a 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 underlying concurrent collection to memoize the value computed by $calculateVal(k)$. 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 2.2b shows sample executions of $compute(K k)$ running concurrently with a general client that performs arbitrary collection operations with arbitrary arguments. Operations of $compute(K$
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 linearizability violation; in practice a smaller fraction of executions reveal the linearizability violation. This execution is non-linearizable with respect to the map sequential specification that executes each operation in an atomic manner. As we show in Chapter 8, a random search of the space of executions fails to find even a single violation in practice.

To trigger a linearizability violation in a composed operation, particular collection operations, with a particular key, must interleave between the collection operations of the composed operation. The set of client operations that can trigger a linearizability violation at a point of execution in the composed operation can be characterized as operations that influence a collection operation in the composed operation. In the example, the executions that reveal the linearizability violation are those in which a client operation influences the result of the \texttt{m.get(k)} in line 7 of \texttt{compute(K k)}. This suggests that an adversarial client should focus on scheduling an operation that influences \texttt{m.get(k)} right before scheduling \texttt{m.get(k)}. An example of an operation that influences \texttt{m.get(k)} is \texttt{m.remove(k)}, as shown in the last execution of Figure 2.2b.

### 2.1.5 Pruning Violating Executions

In this thesis we show that a linearizability violation can be found by testing a composed operation built using a map interface in an adversarial environment that utilizes the influence specification. This result is based on the following:

**Reducing the number of threads**

We show that composed operations are linearizable with respect to two threads where one thread executing one composed operation and the other executing arbitrary base collection operations, must be linearizable w.r.t. an arbitrary number of threads executing arbitrary operations. This reduces the number of interleavings and allows us to deal with programs with an unbounded number of threads. To reduce the number of threads to two we use the base collection linearizability guarantee and the fact that the linearization point of a linearizable composed operation that updates the collection state is the base operation that updates the state.

**Reducing the number of base collection operations**

We show that using the influence specification reduces the number of base operations required without pruning all violation executions (if any exists). This drastically reduces the number of traces to be explored. Note that for the general case, this technique can miss violations. However, for a composed operation built on top of a \texttt{ConcurrentHashMap} with a restricted subset of operations, violations cannot be missed.
1 public Lock getLock(String id) {
2    Lock lock = locks.get(id); // @LP lock != null
3    if (lock == null) {
4        lock = new ReentrantLock();
5        Lock oldLock =
6            locks.putIfAbsent(id, lock); // @LP
7        if (oldLock != null) {
8            lock = oldLock;
9        }
10    }
11    return lock;
12 }

Figure 2.3: A linearizable data-independent composed operation taken from Apache ServiceMix [3]

Even when using two threads and the influence based partial order reduction, one may need to consider an infeasible number of keys and values in order to verify the composed operation as linearizable. In the next section we show the motivation and main idea behind verifying the linearizability of composed concurrent operations by bounding the number of keys and values.

2.2 Verifying Linearizability of Data-Independent Composed Concurrent Operations

2.2.1 Apache ServiceMix Motivating Example

Figure 2.3 shows a composed operation taken from Apache ServiceMix [3] named getLock. This operation uses the underlying ConcurrentHashMap locks to memoize the ReentrantLock allocated at line 4. When the value for a given id is cached in the collection, it is returned immediately; when the ReentrantLock for a given id is not available, it is allocated and inserted into the collection.

The method getLock is linearizable and has two conditional linearization points. The first one is at line 2, and it is a linearization point when locks.get(id) returns a value different than null (“id” is in locks) as then getLock returns the value extracted from locks. The second linearization point is at line 5; when putIfAbsent returns null (indicating the update succeeded) then lock is returned. Otherwise, the current collection value corresponding to id is assigned to oldLock, which is then assigned to lock at line 8 and returned.

Unfortunately, even with the reduction in the number of generated traces (shown in Section 2.1.4), the problem of automatic linearizability verification is still intractable as we need to consider an un-
bounded number of interleavings and global states (inputs to the collection). Next, we introduce concepts that allow us to perform further reduction to the point where we can consider only bounded quantities while still being sound.

We now briefly describe the main ideas that allow us to verify linearizability:

### 2.2.2 Data Independent Composed Operations

We define a class of composed collection operations such that their control flow is independent of the actual collection values being manipulated. We call this class of programs **Singleton Collection Methods** (SCM).

The method `getLock` is an example of an SCM; it takes as input an `id`, which is used only as a key inside the `locks` collection. The return value from the `locks` operations is checked in `getLock` only for equality with `null`, meaning that it is checked only for whether there exists a key `id` inside `locks`. Furthermore, all branches only check the equality of a base operation’s return value to `null`.

**Bounding the number of inputs**  We prove that data-independent composed operations that are linearizable w.r.t. one input key must be linearizable with respect to any input key. In `getLock` this means that:

\[
\text{if } \text{getLock} \text{ is linearizable for a specific } id, \text{ then it is linearizable for any input } id.
\]

We show that data-independent composed operations that are non-linearizable with respect to a specific input key are also non-linearizable for any given key. Thus, the specific choice of input key is irrelevant to the verification. We also support a special case of data-dependent programs that depend on a fixed number of input keys since they also permit effective verification.

**Bounding the global state size**  In this thesis we also prove that the number of map entries needed to expose linearizability violations in data-independent custom collection manipulations is bounded. Thus, for programs with finite local states such as `getLock` checking linearizability is actually decidable.
Chapter 3

Preliminaries

In this chapter we define the concurrent collection interface and present the map interface used in this thesis. We also define composed operations and provide a formal definition of their linearizability [47]. Our definition is nonstandard in that it utilizes the linearizability of the base concurrent collection operations.

3.1 Language

For the rest of this thesis we use the language shown in Figure 3.1. This language is a basic assembly language with labeled statements: assignments, sequencing and conditional gotos. We do not elaborate on the construction of numerical and boolean expressions, which are standard. The language is also equipped with the following features:

- Parallel composition of sequential commands
- Method invocation and response

$x, y \in Var$

$m \in MID$

$l \in Lab$

$B \in BExp := \ldots$

$E \in NExp := \ldots$

$Cm \in Command := l: x = E | l: \text{if } B \text{ goto } l' | Cm; Cm |
\hspace{2cm} l: \text{invoke } m \bar{x} | l: \text{response } m \ x$

$P := Cm \parallel \ldots \parallel Cm$

Figure 3.1: A syntax of a core language for composed operations
We use \( \text{Var} \) to denote the set of local variables for each thread, \( \text{MID} \) to denote a finite set of method identifiers, and \( \text{Lab} \) to denote the set of program labels. We assume that each command that executes in parallel executes in a separate thread, and we use \( T = \{ t_1, t_2, \ldots, t_n \} \) to denote the finite set of thread identifiers \(^1\). We assume that the set of values obtained from expression evaluation includes at least the integers and the Booleans. Note that the language does not support mutual exclusion primitives.

### 3.2 Concurrent Collection and Composed Operations

**Definition 3.2.1 (Concurrent Collection)** A concurrent collection is a type that exposes a set of methods that we call the operations. Each operation may take arguments and has a return value.

**Definition 3.2.2 (Composed Operation)** Given a concurrent collection, a composed operation is a method that may invoke more than one operation of the given concurrent collection. We refer to the given collection operations as base collection operations.

**Definition 3.2.3 (Custom Collection)** Given a concurrent collection and a composed operation defined over it, one can define a new concurrent collection that exposes the composed operation as a new operation. We refer to the new concurrent collection as a custom collection.

**Map** Table 3.1 shows a sequential specification of a map. In this table, \( M : \mathbb{N} \rightarrow \mathbb{N} \) denotes the content of the map. Influenced by the results of our empirical study, in which all the discovered composed operations use concurrent maps, for the rest of the thesis, we assume that a concurrent collection specification is a map. Note that the thesis can be extended to support collections other than maps. Note also that in Java’s \text{ConcurrentHashMap}, \( \bot \) is represented by \texttt{null}, therefore, in some of the examples during this thesis we use \texttt{null} instead of \( \bot \).

Given a map, an updating operation is a base operation where the updated value differs from \( M \).

### 3.3 Histories

A *history* is a sequence of events. Our definition of histories is similar to the standard definition in [47], with a minor exception: atomic operations are permitted during composed operations, as long as they are executed by the same thread. We refer to these atomic operations as low level operations. Any operation that is not low level is referred to as a high level operation. Moreover, because base collection operations are linearizable, they can be viewed as being executed atomically.

An event \( e \in \mathcal{E} \) executed by a thread \( t \in \mathcal{T} \) is one of the following:

---

\(^1\)Our method also easily supports programs with dynamic thread allocations
### 3.3. Histories

<table>
<thead>
<tr>
<th>Operation</th>
<th>Updated Map Value</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>get(k)</code></td>
<td>$M$</td>
<td>$M(k)$</td>
</tr>
<tr>
<td><code>put(k, v)</code></td>
<td>$M[k \mapsto v]$</td>
<td>$M(k)$</td>
</tr>
</tbody>
</table>
| `putIfAbsent(k, v)`    | \[
\begin{align*}
M & \quad M(k) \neq \bot \\
M[k \mapsto v] & \quad o/w
\end{align*}
\] | $M(k)$       |
| `remove(k)`            | $M[k \mapsto \bot]$ | $M(k)$       |
| `remove(k, v)`         | \[
\begin{align*}
M & \quad M(k) \neq v \\
M[k \mapsto \bot] & \quad o/w
\end{align*}
\] | $M(k) = v$       |
| `replace(k, v, v')`    | \[
\begin{align*}
M & \quad M(k) \neq v \\
M[k \mapsto v'] & \quad o/w
\end{align*}
\] | $M(k) = v$       |

| Table 3.1: Sequential specification of a map. $M : \mathbb{N} \rightarrow \mathbb{N}$ denotes the content of the map. |

- An **invocation** $^t_{o/p}/a$ where $op$ is an operation identifier and $a$ is the value of the actual arguments in the invocation.

- A **response** $^t_{o/p}/r$ where $op$ is an operation identifier and $r$ is the return value.

We sometimes refer to a $^t_{o/p}/a$ followed immediately by a $^t_{o/p}/r$ as an *atomic operation* $^t_{o/p}/a/r$. We define a composed operation $op$ for which every atomic operation between its invocation $^[op]$ and its response $^[op]$ are executed by the same thread as an atomic operation. Otherwise, we define the composed operation as non-atomic.

Given a custom collection $C$, we define a *legal history* $h$ of $C$ as a history for which every atomic operation in $h$ follows the specification of $C$.

**Example** An example of a history for the custom map from Figure 2.3 is shown in Figure 3.2. Figure 3.2 shows a history $h_1$, which involves events by two threads (for now, ignore the star symbols). Thread $t_1$ has $^[t_1.getLock/\text{"l"}/t_1\text{.getLock}/l]$ events for the composed operation `getLock`. This operation has two low level operations, $t_1\text{.get/"l"/null}$ and $t_1\text{.putIfAbsent/"l"/l'}$ and $t_1\text{.putIfAbsent/"l"/l'}/l$. It also has two high level collection operations, $t_2\text{.put/"l"/l}/null$ and $t_2\text{.get/"l"/l}$. A sequential history is a history in which every operation is atomic. A thread subhistory, $h|tid$ is the subsequence of all events in $h$, excluding low level operations, that have thread id $tid$. Two histories $h_1, h_2$ are equivalent when for every $tid \in \text{TID}$, $h_1|tid = h_2|tid$.

We denote a history $h$ built by a concatenation of sub-histories $p, m, s,$ and $o$ by $h = p : m : s : o$. Note that a sub-history is a fraction of a history. Also note that two histories with different low level operations may be equivalent.
Figure 3.2: Histories for the custom map builds by \texttt{getLock} from Figure 2.3.
3.4 Linearizability

In this section we formally define the linearizability [47] of custom collections.

A high level operation $op_1$ precedes a high level operation $op_2$ in $h$, and write $op_1 <_h op_2$, if $op_1$ appears before $op_2$ on $h$.

**Definition 3.4.1 (Linearizability)** A history $h$ is linearizable, when there exists an equivalent legal sequential history $s$ of $C$ called a linearization, such that for every two high level 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 preserves the global ordering of non-overlapping high level operations in $h$.

**Example** The history $h_1$ from Figure 3.2 is linearizable; $s$ is a possible equivalent sequential history which preserves the global order of non-overlapping operations and the semantics of the map. We refer to $s$ as a possible linearization of $h_1$ (in general, a concurrent history may have multiple linearizations). Note that even though $s$ is a possible linearization of $h_1$, the low level operations in $getLock$ differ for $h_1$ and $s$.

In contrast, the history $h_2$ is non-linearizable. This is because the first $t_2.get/“l”/l$ returns $l$ after $t_1.get/“l”/null$ returns null. Moving this operation to appear before $getLock$ would yield a null return value and after would yield $l’$.

We know from the definition of linearizability that for every non-atomic high level operation $op$, there exists a point between its invocation $[op$ and response $op$] in the history $h$ where $op$ takes effect. This point is typically referred to as the linearization point $lp(op)$ of the operation $op$. For atomic high level operations, the linearization point is the point of the operation. Given a concurrent history $h$, the (total) ordering between these points induces a linearization marked as $lin(h)$.

We say that a history $h$ is linearizable w.r.t. a linearization point $lp$, when $lin(h)$ is an equivalent legal sequential history of $C$ such that for every two high level operations $op_1, op_2$, if $op_1 <_h op_2$ then $op_1 <_{lin(h)} op_2$. That is, $lin(h)$ is equivalent to $h$ and preserves the global ordering of non-overlapping high level operations in $h$.

**Example** Consider the history $h_1$ of Figure 3.2, where the star symbols in the figure denote the occurrence of a 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 $s$.

3.5 Checking Linearizability

There are two alternative ways to check linearizability [73]: (i) automatic linearization—explore all permutations of a concurrent history to find a valid linearization; (ii) linearization points—build a lin-
earization $\text{lin}(h)$ of a concurrent history, 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 a heuristic for guessing them. This technique checks whether the concurrent execution is equivalent to a specific legal sequential execution defined by the linearization points. In this thesis, we use the second technique to verify the linearizability of composed concurrent operations because it is very easy to identify potential linearization points in all our real-life composed operations.
Chapter 4

Reducing the Number of Threads

In this chapter we show that the linearizability of custom collections can be verified by executing the composed collection operation in an environment executing arbitrary base collection operations. Henceforth we assume that C is a custom map, C.M is its composed operation, and we use \( * \) to denote an arbitrary value. We also assume that C.M does not access any global memory other than C. Note that C is a custom map that exposes C.M and other base collection operations. Therefore, for the rest of the thesis when we say that C is non-linearizable it means that C is non-linearizable in an open environment, however, this does not necessarily mean that C is non-linearizable in a specific environment.

Our proof relies on the fact that if there exists an updating operation of the composed operation, then this point is the linearization point of the composed operation. We first prove that linearizable composed operations cannot include more than one updating operation. The idea behind the proof is to wrap each updating operation and show that the history cannot be rearranged to be sequential. Note that in the following lemmas \( p, m, s, \) and \( o \) are subhistories of C. In addition, \( t.w/\langle k, v \rangle/r \) denotes a put operation executed by thread \( t \) with key \( k \), value \( v \), and return value \( r \).

**Lemma 4.0.1** Given C, if there exists a history \( h = p : [t^{\text{op}}/ \ast : m : t.w_1/\langle k, v \rangle/r : s : t.w_2/\langle k', v' \rangle/r': o : t^{\text{op}}/\ast] \) such that \( w_1 \) and \( w_2 \) are updating operations of C, then C is not linearizable and there exists a non-linearizable history that contains one thread executing one C.M operation and at least one other thread executing base collection operations.

Proof: Consider such a history

\[
h = p : [t^{\text{op}}/ \ast : m : t.w_1/\langle k, v \rangle/r : s : t.w_2/\langle k', v' \rangle/r': o : t^{\text{op}}/\ast].
\]

We build a history \( h_2 \) from \( h \) of the form:

\[
h_2 = p_2 : [t^{\text{op}}/ \ast : m_2 : t.w_1/\langle k, v \rangle/r : s_2 : t.w_2/\langle k', v' \rangle/r': o_2 : t^{\text{op}}/\ast],
\]

where \( p_2, m_2, s_2, \) and \( o_2 \) are generated from \( h \) by removing for each \( op' \neq op \) in \( p, m, s, \) and \( o \), the \( [*^{\text{op}}/ \ast \text{ and } ^{\ast \text{op}}/\ast] \) events. Moreover, for each atomic high level operation (including the new ones
generated by removing \([t^{\text{op}} \rightarrow */] \text{ and } [s^{\text{op}} \rightarrow */] \) events) with a thread identifier \(t\), we replace \(t\) with a thread identifier that differs from \(t\).

Note that \(h_2\) is a history of \(C\) because all atomic operations follow \(C\) specification as in \(h\). Moreover, due to their atomic behavior and their encapsulation, these operations do not overlap and their thread identifier can be changed to any thread.

\(h_2\) is a history of \(C\) with one thread executing one C.M operation and all other threads in \(h_2\) executing base collection operations. Using \(h_2\), we now generate a non-linearizable history of \(C\) with one thread executing one C.M operation and at least one other thread executing base collection operations.

Using \(h_2\), we build a history \(h'\) of the form:

\[
h' = p_2 : [t^{\text{op}} \rightarrow */ : m_2 : t_1.w_3/(k, r)/r : t_1.r_1/k/v : s_2 : t_1.r_2/k'/r'/r' : t_1.r_3/k'/v' : o_2 : t^{\text{op}} \rightarrow */],
\]

where \(t_1 \neq t\) is a new thread identifier and \(r_1, r_2,\) and \(r_3\) are newly created get operations which do not influence \(\text{op}\)'s return value. Also, \(w_3\) is a new put operation which is not an updating operation because it writes the current value.

We note that \(h'\) is a history of \(C\) since \(r_1, r_2, r_3,\) and \(w_3\) are non-updating operations that follow the specification of \(C\). Moreover, because \(r_1, r_2, r_3,\) and \(w_3\) are atomic high level operations with thread identifier \(t_1 \neq t\) \(h'\) has one thread executing one C.M operation (as in \(h_2\)) and at least one other thread executing base collection operations.

We also note that \(\text{op}\) and \(t_1.r_1/k/v\) are overlapping operations in \(h'\). Thus, to complete the proof, we now show by case analysis that it is impossible to rearrange \(h'\) to avoid this overlapping and therefore \(h'\) is non-linearizable.

**Case 1:** Moving \(t_1.r_1/k/v\) before \([t^{\text{op}} \rightarrow */\). The last high level atomic operation that appears before \(t_1.r_1/k/v\) in \(h'\) is \(t_1.w_3/(k, r)/r\). Thus, moving \(t_1.r_1/k/v\) before \([t^{\text{op}} \rightarrow */\) requires moving \(t_1.w_3/(k, r)/r\) right before \(t_1.r_1/k/v\) and consequently in \(t_1.r_1/k/v, v\) cannot differ from \(r\). Because \(w_1/(k, v)/r\) is an updating operation, we know that \(v \neq r\). Therefore, we cannot move \(t_1.r_1/k/v\) before \([t^{\text{op}} \rightarrow */\).

**Case 2:** Moving \(t_1.r_1/k/v\) after \([t^{\text{op}} \rightarrow */\). Because \(t_1.r_2/k'/r'\) appears after \(t_1.r_1/k/v\) in \(h'\), then \(t_1.r_2/k'/r'\) should appear after \(t_1.r_1/k/v\) and \(t_1.r_3/k'/v'\) is the first atomic high level operation that appears after \(t_1.r_2/k'/r'\) in \(h'\). Therefore, moving \(t_1.r_2/k'/r'\) after \(t_1.r_1/k/v\) requires moving \(t_1.r_3/k'/v'\) right after \(t_1.r_2/k'/r'\) and consequently \(r'\) cannot differ from \(v'\). Because \(w_2/k', v'/r'\) is an updating operation, we know that \(v' \neq r'\). Therefore, we cannot move \(t_1.r_1/k/v\) after \([t^{\text{op}} \rightarrow */\).

We showed that it is impossible to rearrange \(h'\) to avoid the overlapping of \(r_1\) and \(\text{op}\). Therefore, \(h'\) is non-linearizable.

As we previously showed, \(h'\) contains one thread executing one C.M operation (\(\text{op}\)) and at least one other thread executing base collection operations. Therefore, we proved that there exists a non-linearizable history \(h'\) contains one thread executing one C.M operation (\(\text{op}\)) and at least one other thread executing base collection operations.
```plaintext
1 V twoWrites(K) {
2   m.put(K, 4);
3   m.put(K, 4);
4 return 4;
5 }
```

(a)

(b)

Figure 4.1: An example with two writes. (a) shows a method `twoWrites` that writes value 4 twice to the same key and returns it. (b) shows a non-linearizable history involving `twoWrites`.

Figure 4.1a shows a method `twoWrites` that writes value 4 twice to the same key and returns 4. Figure 4.1b shows a non-linearizable history for C. This history is non-linearizable because moving `t2.put/⟨2, 7⟩/4` before `twoWrites` changes the return value of `put` to null and moving `t2.put/⟨2, 7⟩/4` after `twoWrites` forces `t2.get/2/4` to move after `t2.put/⟨2, 7⟩/4` and also changes its return value to 7.

Similar histories can be built for every composed operation with two mutating operations by adding the corresponding operations that modify and observe the state (such as `put` and a `get`) at the appropriate places.

Next we use the result of Lemma 4.0.1 to prove that updates uniquely determine linearization points. Recall that, according to this Lemma, linearizable composed collection operations cannot include more than one update to the base collection.

Lemma 4.0.2 Given C and a linearization point lp for C.M, if there exists a history \( h = p : [\mathcal{L} \mathcal{O} / \ast : m : t.w/⟨k,v⟩/v'/s : \mathcal{L} \mathcal{O} / \ast] \cdots \) such that \( w \) is an updating operation and \( w \) is not \( lp \), then there exists a non-linearizable history (w.r.t to \( lp \)) that contains one thread executing one C.M operation and at least one other thread executing base collection operations.

Proof: Consider such a history

\[
h = p : [\mathcal{L} \mathcal{O} / \ast : m : t.w/⟨k,v⟩/v'/s : \mathcal{L} \mathcal{O} / \ast]
\]
If $h$ has two updating operations then, using Lemma 4.0.1, we conclude that there exists a non-linearizable history (w.r.t to $lp$) that contains one thread executing one C.M operations and at least one other thread executing base collection operations.

Otherwise, we build a history $h_2$ of the form:

$$h_2 = p_2 : [t_{op}/\ast : m_2 : t.w/\langle k, v\rangle/v': s_2 : t_{op}/r]$$

, where $p_2$, $m_2$, and $s_2$ are generated by removing, for each $op' \neq op$ in $m$, and $s$, the $[\ast_{op'}/\ast$ and $\ast_{op'}/\ast$] events. Moreover, for each atomic high level operation (including the new ones generated by removing $[\ast_{op'}/\ast$ and $\ast_{op'}/\ast$] events) with a thread identifier $t$, we replace $t$ with a thread identifier that differs from $t$.

Note that $h_2$ is a history of C because all atomic operations follow C specification as in $h$. Moreover, due to their atomic behavior, these operations do not overlap, and their thread identifier can be changed to any thread identifier.

$h_2$ is a history of C with one thread executing one C.M operation and all other threads in $h_2$ executing base collection operations. We now show by case analysis that if $t.w/\langle k, v\rangle/v'$ is not a linearization point of $op$ in $h_2$, then C is not linearizable with respect to $lp$ and there exists a non-linearizable history of C with one thread executing one C.M operation and at least one other thread executing base collection operations. We distinguish between the following cases:

Case 1: The linearization point of $op$ appears before $t.w/\langle k, v\rangle/v'$ in $h_2$. In this case, we build a history of the form:

$$h' = p_2 : [t_{op}/\ast : m_2 : t_1.w_1/\langle k, \bar{v}\rangle/v' : t.w/\langle k, v\rangle/\bar{v} : t_1.r_1/k/v : \cdots]$$

where $t_1 \neq t$ is a new thread identifier, $r_1$ is a new get operation, and $w_1$ is a new updating put operation writing a value $\bar{v}$ that differs from $v$.

We note that $h'$ is a history of C since $r_1$ is a non-updating operation that follows the specification of C. Moreover, following the specification of C in $h'$, $w_1$ returns the value returned by $w$ in $h_2$ and $w$ returns the value written by $w_1$ in $h'$.

We also note that since $t_1$ is a new thread identifier, $h'$ has one thread executing one C.M operation and at least one other thread executing base collection operations.

We now show that the value returned by $r_1$ in $h'$ differs from the value returns by $r_1$ in $lin(h')$ and conclude that C is not linearizable w.r.t $lp$.

$r_1$ returns the value $v$ in $h'$ because it appears after $t.w/\langle k, v\rangle/\bar{v}$. $p_2$ and $m_2$ are the same in $h_2$ and $h'$, and the linearization point of $op$ appears before $w$ in $h_2$. Therefore, the linearization point of C.M in $h'$ appears before $w_1$ and consequently, in $lin(h')$, $op$ appears before $w_1$. $r_1$ is the first high level operation that appears after $w_1$ in $h'$. Thus, since C.M is the only non-atomic high level operation in $h'$, $r_1$ appears right after $w_1$ in $lin(h')$. Therefore, in $lin(h')$, $r_1$ returns $\bar{v}$, which as stated, differs from $\bar{v}$.
We conclude that \( h' \) is a non-linearizable history of \( C \) (w.r.t. \( lp \)), with one thread executing one C.M operation and at least one other thread executing base collection operations.

**Case 2:** the linearization point of \( op \) appears in \( s \). In this case, we build a history of the form:

\[
\tilde{h} = p : \left[ t_{op} */ m : t.w / \langle k, v \rangle / v' : t_1.r_1 / k / v : \cdots \right]
\]

where \( t_1 \neq t \) is a new thread identifier and \( r_1 \) is a new get operation.

We note that \( \tilde{h} \) is a history of \( C \) since \( r_1 \) is a non-updating operation that follows the specification of \( C \).

We also note that since \( t_1 \) is a new thread identifier, \( \tilde{h} \) has one thread executing one C.M operation and at least one other thread executing base collection operations.

We now show that the value returned by \( r_1 \) in \( \tilde{h} \) differs from the value returned by \( r_1 \) in \( \text{lin} (\tilde{h}) \) and conclude that \( C \) is non-linearizable w.r.t. \( lp \).

\( r_1 \) returns the value \( v \) in \( \tilde{h} \) because it appears after \( t.w / \langle k, v \rangle / v' \). \( p_2 \) and \( m_2 \) are the same in \( h_2 \) and \( \tilde{h} \), and the linearization point of \( op \) appears after \( w \) in \( h_2 \). Therefore, the linearization point of C.M in \( \tilde{h} \) appears after \( r_1 \) and consequently, in \( \text{lin} (\tilde{h}) \), \( op \) appears after \( r_1 \).

\( w \) is the only updating low level operation of \( op \) in \( h_2 \). Therefore, because \( p_2 \) and \( m_2 \) are the same in \( h_2 \) and \( \tilde{h} \), \( w \) is the first mutating operation of \( op \) in \( \tilde{h} \). Consequently, the return value of \( r_1 \) in \( \text{lin} (\tilde{h}) \) is \( v' \) as returned by \( w \). Because \( w \) is an updating operation in \( \tilde{h} \), \( v \) differs from \( v' \) and consequently \( C \) is non-linearizable w.r.t. \( lp \).

We conclude that \( \tilde{h} \) is a non-linearizable history of \( C \) (w.r.t. \( lp \)), with one thread executing one C.M operation and at least one other thread executing base collection operations.

We showed that if \( w \) is not \( lp \), then there exists a non-linearizable history (w.r.t. \( lp \)) that contains one thread executing one C.M operation and at least one other thread executing base collection operations.

\[\square\]

Before stating the main result of this chapter, we now prove that every linearizability violation can be observed with one thread executing the composed operation and the other threads executing base collection operations. The proof relies on the fact that a linearizable composed operation may have at most one updating operation and this operation is the linearization point.

**Lemma 4.0.3** Given \( C \), a linearization point \( lp \) for C.M, and a non-linearizable history \( h \) (w.r.t. \( lp \)), there exists a non-linearizable history \( h' \) where \( h' \) contains one thread executing one C.M operation and at least one other thread executing base collection operations.

Proof:

Consider such a history \( h \), non-linearizable with respect to \( lp \).

The consequence of Lemma 4.0.1 and Lemma 4.0.2 is that if \( h \) either a C.M operation with two updating operations or an updating operation that is not \( lp \) then there exists a non-linearizable history
(w.r.t. $lp$) that contains one thread executing one C.M operation and at least one other thread executing base collection operations.

It remains to show that, for histories for which each C.M operation has at most one updating operation $w$, and $w$ is $lp$, there exists a non-linearizable history (w.r.t. $lp$) that contains one thread executing one C.M operation and at least one other thread executing base collection operations.

We build a history $h'$ from $h$ such that after every linearization point of a C.M operation in $h'$, we add $\texttt{get}$ operations $t_1.r_k/k/*$, with a new thread identifier $t_1$, for each $k$ used as a key in an operation in $h$ or in $\text{lin}(h)$, and we update the return value for each $r_k$ using the specification of C.

Note that because we determine the return value for each $r_k$ using the specification of C, and all $r_k$ operations are performed by a new thread, $t_1$, $h'$ is a history of C. Moreover, $h'$ is non-linearizable because it is built from $h$ and each $r_k$ is a $\texttt{get}$ operation that does not update the map and consequently, does not influence the return value of other $h'$ operations.

In the rest of the proof we use case analysis to build from $h'$ a non-linearizable history that contains one thread executing one C.M operation and at least one other thread executing base collection operations.

**Case 1:** There exists an operation $op$ in $h'$ returning a value different than $op$'s return value in $\text{lin}(h')$, and every $r_k$ preceding $op$ in $h'$ has the same return value in $\text{lin}(h')$. In this case, we build a history $\tilde{h}$ by removing, for each $op' \neq op$, its $[\star op'/*$ and $\star op'/*]$ events, from $h'$ and use the specification of C to calculate the return value of operations in $\tilde{h}$.

Note that $\tilde{h}$ is non-linearizable because, as in $h'$, the collection value after each linearization point of C.M before $op$ is the same in $\tilde{h}$ and in $\text{lin}(\tilde{h})$. This is because each $r_k$ returns the same value on $h'$ and $\text{lin}(h')$. Moreover, in every C.M operation there is at most one updating operation, which is the linearization point. Therefore, a C.M operation cannot change the collection value after the linearization point. For these reasons, every operation $op' \neq op$ of type C.M can be replaced by its low level operations, and the return value of $op$ in $\tilde{h}$ remains as in $h'$ and also the return value of $op$ in $\text{lin}(\tilde{h})$ remains as in $\text{lin}(h')$.

Note also that because we calculate the return value of the operations in $\tilde{h}$ using the specification of C, $\tilde{h}$ is a history of C. Moreover, $\tilde{h}$ contains one thread executing one C.M operation and at least one other thread executing base collection operations, as required.

**Case 2:** The first operation on $h'$ with a different return value on $h'$ and $\text{lin}(h')$ is an operation $r_k$ that appears after a linearization point of an operation $op$. In this case, we build a non-linearizable history $\tilde{h}$ by replacing every operation $op' \neq op$ of type C.M in $h'$ with its internal low level operation and use the specification of C to calculate the return value of operations in $\tilde{h}$. This generates a non-linearizable history $\tilde{h}$ that contains one thread executing one C.M operations and at least one other thread executing base collection operations. Note that as in case 1, $\tilde{h}$ is non-linearizable because each C.M operation in $h'$ before $op$ has at most one updating operation, and this operation is the linearization point, and each
operation $r_i$ that appears before $r_k$ returns the same value in $h'$ and $lin(h')$.

We show that there exists a non-linearizable history $\tilde{h}$ that contains one thread executing one C.M operation and at least one other thread executing base collection operations.

Finally, we are ready to state a reduction theorem for threads. The proof for this theorem uses Lemma 4.0.3 and the linearizability guarantee of base collection operations.

**Theorem 4.0.1** Given $C$, a linearization point $lp$ for C.M, and a non-linearizable history $h$ (w.r.t. $lp$), there exists a non-linearizable history $h'$ where $h'$ contains one thread executing one C.M operation and exactly one other thread executing base collection operations.

**Proof:**

Consider such a history $h$, non-linearizable with respect to $lp$. According to Lemma 4.0.3 there exists a non-linearizable history $\tilde{h}$ where $\tilde{h}$ contains one thread executing one C.M operation and at least one other thread executing base collection operations. Using the history definition in Chapter 3, we conclude that no atomic operations overlap and therefore can be performed by one thread without changing the linearizability of $h$.

We generate a history $h'$ by changing all the thread identifiers of atomic high level operations in $\tilde{h}$ to a new thread identifier $t_2$ and changing the thread identifier of the C.M operation and its low level operations to a new thread identifier $t_1$. This history is a non-linearizable history of $C$ (w.r.t. $lp$) that contains one thread ($t_1$) executing one C.M operation and exactly one other thread ($t_2$) executing base collection operations.

**Corollary 4.0.4** The linearizability of composed operations C.M can be checked by executing C.M operation in an environment executing base collection operations.
CHAPTER 4. REDUCING THE NUMBER OF THREADS
Chapter 5

Thread-Centric Linearizability Testing

In this chapter, we describe a simple thread-centric procedure for testing the linearizability of composed concurrent operations. This approach is based on the paper “Testing Atomicity of Composed Concurrent Operations” that was presented at OOPSLA’11 [64].

The procedure utilizes the result of Corollary 4.0.4 and checks the linearizability of C.M in an environment that performs collection operations. The main insight behind our approach is to leverage the influence of collection operations in order to define the environment behavior. With our approach, we do not produce sub histories that we know are certainly linearizable. This enables us to significantly prune the massive search space of possible executions, without pruning violating executions, and focus on schedules that can trigger violations.

5.1 Influence Guided Environment

Definition 5.1.1 (influence) Given a history \( h = p : t_1.op_1/a_1/r_1 : t_2.op_2/a_2/r_2 \) of C, we say that \( op_1 \) influences \( op_2 \) in \( h \) if there exists a history \( h' = p : t_2.op_2/a_2/r_2' \) of C and \( r_2 \neq r_2' \).

An influence specification for some of the map operations is shown in Table 5.1. Such specifications are easy to obtain from the specification of map from Table 3.1. The table is used as follows: i) select an operation \( op_1 \) from a row; ii) select an operation \( op_2 \) from a column; iii) pick a prefix history \( h \) such that the state at the end of \( h \) satisfies the condition in the box. Then, by the definition of influence, \( op_1(k) \) influences \( op_2(k) \) in \( h \). If the box is false, \( op_1 \) can never influence on \( op_2 \).

General Approach It is possible to augment the procedure’s environment to be influence-aware. That is, the environment selects an operation with such keys and values so that the operation influences the operation that is about to be performed by C.M.

The intuition behind this approach is conceptually simple: if the environment operation does not influence all of C.M's operations, then we can shift it to not interleave with C.M. That is, we can always
Table 5.1: Influence specification for a map. For simplicity, we assume that \( v_1 \) and \( v_2 \) are non-null.

produce a linearization of the concurrent history by simply moving all of environments operations to precede or follow the operations in C.M. Indeed, producing concurrent histories that we know can always be linearized is not useful for finding linearizability violations.

5.2 Exploration Procedure

A local state \( l \in L \) describes the values of the local variables. For simplicity we assume that there is an extra local variable \( pc \) denoting the current program location. We also assume that the stack is part of the local state.

Let \( C \) be the set of base collection values and let \( A \) be the set of values for actual arguments and return values of the collection. Let \( \Sigma = L \times C \) denote the set of program states.

We say that a method of \( C \) is a base method if it is not C.M. We use \( Stmt \) to denote the set of possible program statements.

5.2.1 Transition Relation

Given a method of interest C.M, we define the transition relation \( \Pi \subseteq \Sigma \times Stmt \times \Sigma \) as a small-step operational semantics. We define two kinds of transitions:

- Local Transition:

  \[
  \langle l, c \rangle \xrightarrow{s} \langle l', c \rangle
  \]

  That is, executing a local statement in C.M on state \( \langle l, c \rangle \) yields a state \( \langle l', c \rangle \). It is essential that the stack be part of the local state. Here, \( s \) can be any statement except those that invoke C methods.

- Collection Transition: This transition always invokes base methods of \( C \). We assume that invocation (calling and returning) of a base method is atomic. Depending on how the local state is affected and who performs the transition, we define two kinds of collection transitions:
5.2. Exploration Procedure

- **Main Transition:**

\[
\langle l, c_i \rangle \xrightarrow{x = \text{C.b}(\text{args})} \langle [(\text{pc} \mapsto q, x \mapsto r)], c_o \rangle
\]

Here, \(x = \text{C.b}(\text{args})\) is always a statement of C.M at \(\langle l, c_i \rangle\) and the statement is followed by a label \(q\) in C.M. Thus, executing the transition from state \(\langle l, c_i \rangle\) involves: (i) evaluating \(\text{args}\) to \(a\) in \(l\); (ii) invoking method \(b\) with \(a\), which produces a new collection \(c_o\); and (iii) setting return value \(r \in A\) to the local variable \(x\). Note that main transitions always invoke methods that return a value.

- **Environment Transition:**

\[
\langle l, c_i \rangle \xrightarrow{C.b(\text{args})} \langle l, c_i' \rangle
\]

An environment transition does not access local states and always modifies the collection state, i.e., \(c_i \neq c_i'\).

An environment transition is enabled if:

\* \(\exists c_o, c_o' \in C\) and \(\exists l', l'' \in L\) s.t. the following two main transitions exist:

\[
\langle l, c_i \rangle \xrightarrow{x = \text{C.b}_i(\text{args})} \langle l', c_o \rangle
\]

\[
\langle l, c_i' \rangle \xrightarrow{y = \text{C.b}_j(\text{args})} \langle l'', c_o' \rangle
\]

where \(r_1\) is the return value of \(b_i\)'s invocation, \(r_2\) is the return value of \(b_j\)'s invocation, and \(b_i = b_j\).

\* \(r_1 \neq r_2\).

Note that the **Main Transition** executes atomic low level operations and the **Environment Transition** executes atomic high level operations. Also note that in the environment transition \(C.b(\text{args})\) influences the return value of \(x = \text{C.b}_i(\text{args})\) \((b_i = b_j)\) in state \(c_i\). For example, if \(c_i = \emptyset\) and \(x = \text{C.b}_i(\text{args})\) is \(x = \text{C.get}(2)\) then \(C.b(\text{args})\) could be a \(\text{C.put}(2, 4)\).

5.2.2 Traces and Exploration Procedure

A trace \(\pi = \langle l_0, c_0 \rangle \xrightarrow{\delta} \langle l_1, c_1 \rangle \ldots \xrightarrow{\delta} \langle l_n, c_n \rangle\) is a sequence of transitions such that:

- \(\langle l_0, c_0 \rangle\) is an initial state.
- \(\forall i. 0 \leq i \leq n. \langle l_i, c_i \rangle \xrightarrow{\delta} \langle l_{i+1}, c_{i+1} \rangle \in \Delta.\)
• \( (l_{n-1}, c_{n-1}) \xrightarrow{\text{return } e} (l_n, c_n) \) is a main transition (denoting completion of C.M).

We denote the set of traces as \([\Pi]\).

A program trace is sequential if does not include environment transitions.

Recall that we use the proofs of Chapter 4 and consider only traces with two thread s.t. the main thread executes C.M and the environment thread executes base operations of C.

**Definition 5.2.1 (Linearization Point)** Given a program trace \( \pi = (l_0, c_0) \xrightarrow{s} (l_1, c_1) \ldots \xrightarrow{s} (l_n, c_n) \), we define linearization point \( i \) as a transition \( (l_i, c_i) \xrightarrow{s} (l_{i+1}, c_{i+1}) \), where \( 0 \leq i \leq n \).

The following provides a constructive linearizability check performed directly on traces.

**Definition 5.2.2 (Linearizable Trace)**
A trace \( (l_0, c_0) \ldots (l_i, c_i) \xrightarrow{s} (l_{i+1}, c_{i+1}) \ldots (l_n, c_n) \) is linearizable if it satisfies the following conditions:

(i) there exists a unique linearization point \( i \),

(ii) the only possible update of the collection by a main transition occurs in transition \( (l_i, c_i) \xrightarrow{s} (l_{i+1}, c_{i+1}) \), and

(iii) for every sequential run \( \pi_s \) starting in a state \( (l_0, c_i) \) and ending with a state \( (l', c') \), it is the case that \( c' = c_{i+1} \) and the return value of \( \pi_s \) and \( \pi \) are equal.

Note that Definition 5.2.2 defines the specification that needs to be checked in order to verify that a trace is linearizable. Therefore, our testing algorithm uses the transition relation from Section 5.2.1 to generate traces and the specification from Definition 5.2.2 to check whether the generated traces are linearizable.

**5.3 Example**

We now show a non-linearizable example, together with a trace generated using our algorithm, showing a linearizability violation.

Figure 5.1 shows an example of a non-linearizable method, `doubleNonLin`, inspired by bugs from Adobe BlazeDS, Vo Urp and Ehcache-spring-annotations. This method uses an underlying concurrent collection \( m \) to memoize the value \( K*2 \) for each \( K \). When the value for a given key is cached in the collection it is read and returned; when the value for a given key is not available, it is computed and inserted into the collection and then read and returned.

Figure 5.2 shows a trace exposing a linearizability violation of `doubleNonLin` that was generated using our algorithm. In this figure, a state is represented by a box, a linearization point is represented by
5.4. Pruning Violating Traces

In the general case, the presented procedure may miss linearizability violations. Next we show two examples where our technique can prune all violating executions.

5.4.1 Non-Encapsulated Composed Operations

Our technique may miss all violating traces when the composed operation has global variables other than C (not encapsulated) and all the non-linearizable traces are feasible only when a global variable is updated during the trace. Figure 5.3 shows a non-encapsulated method writeTwice that accesses a
Figure 5.2: A trace of the `doubleNonLin` method from Figure 5.1 showing a linearizability violation. The violation is revealed by the different return value (val) at the end of the sequential trace. A state is represented by a box, a linearization point is represented by a black box, a state of the sequential run is represented by a circle, an arrow represents a transition, and a dashed arrow represents an environment transition together with its operation.
5.4. PRUNING VIOLATING TRACES

global variable update. This method accesses the map m only when update is assigned to true between lines 2 and 3. In this case, there are two updating operations at lines 4 and 5 accessing the map. Therefore, as Lemma 4.0.1 shows, writeTwice is non-linearizable. Our technique misses this violation because our environment does not change the value of global variables other than C. Therefore, our environment does not change the value of update to true and consequently there are no accesses to m in any of the explored traces.

5.4.2 Type State Based Interface

Our technique will prune all violating traces when an interface with operation op is influenced only by a sequence of operations preceding op in a certain order. Such an interface is shown in Table 5.2 with operations init(k), inc(k), and cPut(k). In this figure, $M : \mathbb{N} \rightarrow \mathbb{N}$ denotes the content of the map and cnt denotes an integer initialized by 0. This interface has an integer cnt that is initialized by 0. In this interface, in order that a cPut operation will update the map, cPut should be preceded by operations init and inc. Our technique executes only one base operation that influences the next base operation of the composed operation. In this interface, there is no single operation that influences the cPut operation of the composed operation writeTwiceInt; therefore, our technique cannot detect the linearizability violation of method writeTwiceInt from Figure 5.4.

Note that our technique can be extended to support this interface as well by executing a few environment operations where the last operation influences the next operation of the composed operation. In this example, before cPut(k, 7), the technique can execute the operations init(k), inc(k), and cPut(k, 12). For the common map interface presented in Table 3.1 our technique is sufficient and does prune violating executions. Therefore, to keep the technique simple and with a low overhead, we did not extend the technique to support other interfaces.

In the next chapter we show that for the map specification defined in Table 3.1, our technique cannot prune all violating executions.
### Operation, Updated $M$ Value, Updated $cnt$ Value, Return Value

<table>
<thead>
<tr>
<th>Operation</th>
<th>Updated $M$ Value</th>
<th>Updated $cnt$ Value</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$init(k)$</td>
<td>$M$</td>
<td>$\begin{cases} 1 &amp; cnt = 0 \ cnt &amp; o/w \end{cases}$</td>
<td>$\begin{cases} 1 &amp; cnt = 0 \ 0 &amp; o/w \end{cases}$</td>
</tr>
<tr>
<td>$inc(k)$</td>
<td>$M$</td>
<td>$\begin{cases} 2 &amp; cnt = 1 \ cnt &amp; o/w \end{cases}$</td>
<td>$\begin{cases} 2 &amp; cnt = 1 \ 0 &amp; o/w \end{cases}$</td>
</tr>
<tr>
<td>$cPut(k, v)$</td>
<td>$\begin{cases} M[k \mapsto v] &amp; cnt = 2 \ M &amp; o/w \end{cases}$</td>
<td>$\begin{cases} 0 &amp; cnt = 2 \ cnt &amp; o/w \end{cases}$</td>
<td>$\begin{cases} 3 &amp; cnt = 2 \ 0 &amp; o/w \end{cases}$</td>
</tr>
</tbody>
</table>

Table 5.2: Sequential specification of a map where our technique may miss all linearizability violations. $M : \mathbb{N} \rightarrow \mathbb{N}$ denotes the content of the map and $cnt$ denotes an integer initialized by 0.

```c
1 void writeTwiceInt(K k) {
2   m.cPut(k, 7);
3   m.cPut(k, 4);
4 }
```

Figure 5.4: A composed operation built using the interface from Table 5.2.
Chapter 6

Thread-Centric Correctness Guarantees

In this chapter we show that a composed operation built using the map interface defined in Table 3.1, is not linearizable w.r.t. linearization point $l_p$ if and only if there exists a non-linearizable trace of the exploration procedure presented in Chapter 5.

We start by using the influence specification of the map to define an influence history and show that for every non-linearizable composed operation there exists a non-linearizable influence history. Then, we show that C is not linearizable w.r.t. linearization point $l_p$ if and only if there exists a non-linearizable program trace $\pi$.

Recall that in Chapter 4 we showed that if C.M has either more than one updating operation or an updating operation which is not its linearization point then C is non-linearizable. Therefore, for the rest of the thesis, we assume that C.M has at most one updating operation and if C is linearizable and there exists an updating operation of C.M then this operation is C.M’s linearization point.

The proof that our technique does not prune all histories where C.M either has two updating operations or an updating operation which is not the linearization point of C.M is similar.

Note also that for the rest of the thesis, non-linearizable means non-linearizable w.r.t. linearization point $l_p$.

6.1 Influence History

**Definition 6.1.1 (Influence History)** An influence history $h_{inf}$ is a history satisfying the following conditions:

(i) $h_{inf}$ has two threads, one executing the C.M operation and another executing base collection operations.

(ii) Every updating high level atomic base operation $w$ in $h_{inf}$ should appear right before a low level atomic operation $op$, and $w$ influences $op$ in $h_{inf}$.
(iii) Every non-updating high level atomic base operation \( r \) should appear right after a linearization point of an operation \( op \) of the C.M operation and is influenced by a low level operation of \( op \) in \( h_{inf} \) or in \( \text{lin}(h_{inf}) \).

(iv) Every updating high level atomic base operation \( w \) in \( h_{inf} \) returns the same value in \( h_{inf} \) and \( \text{lin}(h_{inf}) \).

**Example** An example of an influence history for the custom map from Figure 2.3 is shown in Figure 6.1. Figure 6.1 shows a history \( h_1 \), which involves events by two threads. Thread \( t_1 \) has \( t_1.\text{getLock} / l \) and \( t_1.\text{getLock} / l \) events for the composed operation \( \text{getLock} \). This operation has two low level operations, \( t_1.\text{get} / l \) and \( t_2.\text{putIfAbsent} / l \). It also has two low level collection operations, \( t_2.\text{put} / l \) and \( t_2.\text{get} / l \). The history \( h_1 \) is an influence history that satisfies conditions (i) – (iv) from Definition 6.1.1.

In contrast, the history \( h_2 \) is not an influence history. This is because according to Lemma 4.0.2 the linearization point of \( \text{getLock} \) in \( h_2 \) is \( t_1.\text{putIfAbsent} / l \) and the first \( t_2.\text{get} / l \) appears before the linearization point of \( \text{getLock} \) in contrast with condition (iii) from Definition 6.1.1. Moreover, history \( h_3 \) is also not an influence history because the operations \( t_2.\text{put} / l \) and \( t_1.\text{putIfAbsent} / l \) have different keys and therefore, \( t_2.\text{put} / l \) does not influence \( t_1.\text{putIfAbsent} / l \) in contrast with condition (ii) from Definition 6.1.1.

Next, we prove that a custom map that is non-linearizable w.r.t. linearization point \( lp \) has a non-linearizable influence history. Here we utilize the linearizability guarantee of base collection, the map specification from Table 3.1, and the fact the C.M operation has at most one updating low level operation, which is the linearization point of the C.M operation.

**Lemma 6.1.2** Given \( C \), a linearization point \( lp \) for C.M, and a non-linearizable history \( h \) (w.r.t to \( lp \)), there exist a non-linearizable influence history \( h' \) (w.r.t to \( lp \)) of \( C \).

**Proof:** Consider such a history \( h \), non-linearizable w.r.t. \( lp \). Theorem 4.0.1 implies that there exists a non-linearizable history \( h_2 \) where \( h_2 \) contains one thread executing the C.M operation and exactly one other thread executing base collection operations.

In this proof we use \( h_2 \) and case analysis to construct \( h' \) and show that \( h' \) is a non-linearizable influence history of \( C \) where the return value of every high-level atomic updating operation in \( h' \) is the same in \( h' \) and \( \text{lin}(h') \). \( h_2 \) is non-linearizable and therefore there exists an operation \( op \) in \( h_2 \) where the return value of \( op \) in \( h_2 \) differs from the return value of \( op \) in \( \text{lin}(h_2) \).

**Case 1:** The C.M operation is the first operation that returns different values in \( h_2 \) and \( \text{lin}(h_2) \). In this case, high-level non-updating operations cannot change the return value of the C.M operation and thus do not affect the non-linearizability of \( h_2 \). Hence, we discard these operations from \( h_2 \).
Figure 6.1: Influence and non-influence histories for the custom map builds by getLock from Figure 2.3.
In addition, we use the C commutativity specification to move every high-level atomic operation in $h_2$ as far as possible to the right and discard any high-level operation that appears after the C.M operation. These actions generate a new history $h'_2$.

Note that since we use the C commutativity specification to move every high-level atomic operation in $h_2$ as far as possible to the right, $h'_2$ is a history of C. Moreover, since the C.M operation is the first operation that returns different values in $h_2$ and $\text{lin}(h_2)$, and we removed only operations appearing after the C.M operation, $h'_2$ is non-linearizable.

Further note that $h'_2$ has only high level updating operations and, since the map specification from Table 3.1 guarantees that every two operations with different input key are commutative, every high level operation in $h'_2$ appears before an operation with the same key.

We now generate a non-linearizable influence history of C contains only high-level updating operations. The map specification guarantees that each operation $\text{op}$ is influenced only by the last updating operation that appears before $\text{op}$ in $h'_2$ and has the same input key as in $\text{op}$. Therefore, we remove all high-level atomic operations in $h'_2$ that are not the last high level operation that appears before a low level operation of the C.M operation in $h'_2$. Moreover, we replace the last high level operation $\text{op}$ that appears before each low level operation with a put operation with the same input key and input value as in $\text{op}$ and calculate the return value of the new put using the C specification. If the new put is not an updating operation, we discard it from the history. This generates a new history $h'$.

Note that $h'$ is a history of C since $h'_2$ is a history of C and the transformation from $h'_2$ to $h'$ uses the C specification.

Note also that because the C.M operation is the first operation in $h_2$ with a different return value in $h_2$ and $\text{lin}(h_2)$ and because we used the C specification to replace operations, each high level atomic operations in $h'$ has the same return value in $h_2$ and $\text{lin}(h_2)$.

We generated a non-linearizable influence history $h'$ of C as required.

Case 2: The first operation with a different return value in $h_2$ and $\text{lin}(h_2)$ is a high level atomic operation $t.\text{op}/(k,v)/\text{res}$. In this case, we show that $\text{op}$ is influenced by a low level operation and therefore, should occur after the linearization point of Cin $h_2$.

We start by showing that $\text{op}$ can occur after the linearization point of the C.M operation in $h_2$, and then we continue by showing that $\text{op}$ is influenced by a low level updating operation in $h_2$ or in $\text{lin}(h_2)$.

The C.M operation has at most one low level updating operation in $h_2$ and this operation is the linearization point of the C.M operation in $h_2$. Therefore, the linearizability of the base operations guarantees that every high level operation occurring in $h_2$ before the linearization point of the C.M operation in $h_2$ returns the same return value in $h_2$ and in $\text{lin}(h_2)$. Thus, $\text{op}$ occurs in $h_2$ after the linearization point of the C.M operation.

Note that as in case 1, we use the C commutativity specification to move every high-level atomic operation in $h_2$ as far as possible to the right and discard any high-level operation that appears after the
6.2 Traces and Influence Histories

C.M operation. Moreover, because $op$ has a different return value in $h_2$ and $\text{lin}(h_2)$, we can discard all operations appearing after $op$ on $h_2$ and all high level non-updating operations other than $op$ in $h_2$ and generate a non-linearizable history $h_1^3$ of C.

Also note that because $op$ is the last operation on $h_1^3$, the map specification guarantees that replacing $op$ with a $\text{get}$ operation $g$ with the same input key as $op$ generates a non-linearizable history $h_2^3$ of C.

We now show that $g$ is the first high level operation that appears after the linearization point of C in $h_2^3$ and is influenced by a low level operation of C.

The linearizability guarantee and the specification of the map guarantee that if $g$ occurs after a high level updating operation with the same input key as in $g$, then $g$ must return the same value in $h_2^3$ and $\text{lin}(h_2^3)$. Therefore, $g$ is not influenced by a high level updating operation and is influenced by a low level updating operation.

Note that every operation that does not influence a non-updating operation $g$ is commutative to $g$. Therefore, because we previously use the commutativity of C to move operations to the right, we conclude that $g$ is the first high level operation that appears after the linearization point of C in $h_2^3$ and is influenced by a low level updating operation of C.

Also note that building an influence history where every high level atomic updating operation should occur only before a low level operation can be done as in Case 1 and generates a new history $h'$. Moreover, as we showed in Case 1, all high level updating operations that appear before the linearization point of the C.M operation have the same return value in $h'$ and in $\text{lin}(h')$. Thus, because $g$ is the only high level operation after the linearization point, we conclude that every updating atomic high level operations in $h'$ has the same return value in $h'$ and in $\text{lin}(h')$.

We showed that there exists a non-linearizable influence history $h'$ as required.

The following corollary is a consequence of Definition 3.4.1 and the preceding lemma:

**Corollary 6.1.3** Given C and linearization point $lp$ for C.M, C is non-linearizable if and only if there exists a non-linearizable influence history $h$ (w.r.t. to $lp$) of C.

In the next section, we show the correlation between traces and influence histories.

6.2 Traces and Influence Histories

In this section we show the translation of traces to influence histories. Using the translation, we obtain the main result of this chapter: that a custom map C is not linearizable w.r.t. linearization point $lp$ if and only if there exists a non-linearizable program trace $\pi$.

In order to utilize Corollary 6.1.3 for traces, we start by showing a possible translation from traces to histories.
6.2.1 History Generation

Given a program trace $\pi = (l_0, c_0) \xrightarrow{s} (l_1, c_1) \ldots \xrightarrow{s} (l_n, c_n)$, we generate histories $h_\pi$ and $\text{lin}(h_\pi)$ by traversing $\pi$ and utilize $h_\pi$ to show that our technique checks the linearizability of $C$.

The translation is done in the following manner while traversing $\pi$:

- **C.M invocation** if $\langle l_i, c_i \rangle = \langle l_0, c_0 \rangle$ then $h_\pi = [\text{t.C.M}/\langle \text{key}(l_0), \text{val}(l_0) \rangle]$.

- **C.M response** if $\langle l_i, c_i \rangle = \langle l_n, c_n \rangle$ then $h_\pi = h_\pi : \text{t.C.M}/\text{the return value of } \pi$.

- **Local transition** if $\text{pc}(l_i)$ is a local transition then $h_\pi = h_\pi$.

- **Main transition** if $\text{pc}(l_i)$ is a collection main transition $x = \text{op}(\text{args})$, and when the collection operation $\text{op}$ is executed in $c_i$ with arguments $\text{args}$, it returns the value $r \in \mathcal{A}$ while changing $c_i$ to $c_{i+1}$ and $l_i$ to $l_{i+1}$, then $h_\pi = h_\pi : \text{t.op}/\text{args}/r$.

- **Environment transition** if in $\langle l_i, c_i \rangle \xrightarrow{s} \langle l_{i+1}, c_{i+1} \rangle$, $l_i = l_{i+1}$ and $s = C.\text{op}'(\text{args}')$ with return value $r'$, then $h_\pi = h_\pi : \text{t'.op'}/\text{args'}/r'$ and $\text{lin}(h_\pi) = \text{lin}(h_\pi) : \text{t'.op'}/\text{args'}/r'$.

- **Linearization point** $\text{lin}(h_\pi) = \text{lin}(h_\pi) : [\text{t.C.M}/\langle \text{key}(l_0), \text{val}(l_0) \rangle : \text{t.C.M}/\text{the return value of } \pi]$.

If the map comparison has a different value for a key $k$, we also add a $\text{get}$ operation as follows: $h_\pi = h_\pi : \text{t'.get}/k/r_1$ and $\text{lin}(h_\pi) = \text{lin}(h_\pi) : \text{t'.get}/k/r_2$, where $r_1$ is the value that corresponds to $k$ in $\pi$’s map and $r_2$ is the value that corresponds to $k$ in $\pi_s$’s map.

6.2.2 Trace and History Correlation

The translation from traces to histories, presented in Section 6.2.1, clearly guarantees that $h_\pi$ is an influence history of $C$. Note that the reverse translation to a set of traces can be performed in a similar way given an influence history.

The next lemma guarantees that the techniques presented in Chapter 5 cannot miss violations. The proof is obtained from the translation from Section 6.2.1 and Definition 5.2.2.

**Lemma 6.2.1** Given $C$, a linearization point $lp$ for C.M, and a non-linearizable influence history of $C$, there exists a non-linearizable trace $\pi$ s.t. $h = h_\pi$.

Proof: Consider such an influence history $h$, non-linearizable w.r.t. $lp$.

The translation from Section 6.2.1 guarantees that $h_\pi$ is an influence history of $C$. Therefore, for each influence history $h$, one can generate a trace $\pi$ using the backward translation s.t. $h = h_\pi$. Since $h$ is an influence history, every updating atomic high level operation has the same return value in $h$ and $\text{lin}(h)$. Therefore, $h$ is non-linearizable due to one of the following: (i) the return value of the
low level non-updating operation in \( h \) differs from its return value in \( \text{lin}(h) \); (ii) the return value of the C.M operation in \( h \) differs from its return value in \( \text{lin}(h) \). According to Definition 5.2.2, a trace is non-linearizable if either the maps after the linearization point are different or the result of \( \pi \) differs from the result of \( \pi_s \). (i) guarantees that the maps after the linearization point are different, and thus \( \pi \) is non-linearizable. (ii) guarantees that the result of \( \pi \) differs from the result of \( \pi_s \), and thus \( \pi \) is non-linearizable.

\[\square\]

**Corollary 6.2.2** Given \( C \) and a linearization point \( lp \) for C.M, then if \( C \) is non-linearizable there exists a non-linearizable trace \( \pi \).

The above corollary is obtained from Corollary 6.1.3 and from Lemma 6.2.1. From Corollary 6.1.3 we know that if \( C \) is non-linearizable then there exists a non-linearizable influence history \( h \) (w.r.t. to \( lp \)) of \( C \) and from Lemma 6.2.1 we know that given such an \( h \), there exists a non-linearizable trace \( \pi \).

Next, we show that if a program trace \( \pi \), generated using our technique, is non-linearizable then its corresponding history \( h_{\pi} \) generated using the procedure from Section 6.2.1 is also non-linearizable.

**Lemma 6.2.3** Given a program trace \( \pi \in [\Pi] \), then if \( \pi \) is non-linearizable, \( h_{\pi} \) is a non-linearizable history of \( C \).

**Proof:**

Consider such a non-linearizable program trace \( \pi \). We generate histories \( h_{\pi} \) and \( \text{lin}(h_{\pi}) \) from \( \pi \) using the procedure from Section 6.2.1.

\( \pi \) can be non-linearizable if either the maps are different after the linearization point of C.M or the return values of \( \pi \) and \( \pi_s \) are different. We now use case analysis to prove that if \( \pi \) is a non-linearizable trace then \( h_{\pi} \) is non-linearizable.

**Case 1:** The maps after the linearizable point of the C.M operation are different. In this case, the map comparison has a different value for a key \( k \). Then the translation adds a get operation as follows:

\[
h_{\pi} = h_{\pi} : t'.get/k/r_1 \quad \text{and} \quad \text{lin}(h_{\pi}) = \text{lin}(h_{\pi}) : t'.get/k/r_2,
\]

where \( r_1 \) is the value that corresponds to \( k \) in \( \pi \)'s map and \( r_2 \) is the value that corresponds to \( k \) in \( \pi_s \)'s map. Consequently, \( h_{\pi} \) non-linearizable.

**Case 2:** The return values of the C.M operation in \( \pi \) and \( \pi_s \) are different. In this case, because the return value of C.M in \( h_{\pi} \) is the result of \( \pi \) and the return value of C.M in \( \text{lin}(h_{\pi}) \) is the result of \( \pi_s \), then the return value of the C.M operation in \( h_{\pi} \) and in \( \text{lin}(h_{\pi}) \) is different. Consequently, \( h_{\pi} \) non-linearizable.

We conclude that if a program trace \( \pi \) is non-linearizable, then \( h_{\pi} \) is a non-linearizable history of \( C \).

\[\square\]

We are now ready to state the main theorem. The proof of this theorem uses Lemma 6.2.3 and Corollary 6.2.2.
Theorem 6.2.1  Given \( C, C \) is non-linearizable w.r.t. linearization point \( lp \) if and only if there exists a non-linearizable program trace \( \pi \).

Proof:

\( \Leftarrow \)

Consider such a non-linearizable program trace \( \pi \). Lemma 6.2.3 shows that if \( \pi \) is non-linearizable, then \( h_\pi \) is a non-linearizable history of \( C \). Therefore, we conclude that if there exists a non-linearizable trace, then \( \pi \) then \( C \) is non-linearizable w.r.t. \( lp \).

\( \Rightarrow \)

Consider such a non-linearizable composed map \( C \). Corollary 6.2.2 shows that if \( C \) is non-linearizable, then there exists a non-linearizable trace \( \pi \).

We showed that \( C \) is non-linearizable w.r.t. linearization point \( lp \) if and only if there exists a non-linearizable program trace \( \pi \).

\[ \blacksquare \]

6.3 Testing vs. Verification

Theorem 6.2.1 shows that for the map interface defined in Table 3.1, a composed operation is non-linearizable if and only if there exists a non-linearizable trace of the presented approach. Unfortunately, while useful for testing, the procedure cannot be used to verify linearizability of an arbitrary C.M. Linearizability can only be verified by exploring an unbounded number of inputs and unbounded number of environment operations, which is clearly infeasible.

In the next chapter, we define the notion of data-independence and show that for a data-independent method C.M, it is possible to verify linearizability by considering only a bounded number of inputs and a bounded number of collection elements.
Chapter 7

Data-Independent Operations

In this chapter we define a class of composed operations, named Data Independent, that can be verified for linearizability by considering a bounded number of inputs and collections of a bounded number of entries. While verifying data-independence is undecidable in the general case [75], we provide simple sufficient restrictions that can be used to establish a composed operation as data-independent.

The main idea is to restrict the inputs to the composed operation, and the control flow inside the composed operation, to guarantee that it treats all keys uniformly. First, we limit the parameters to C.M to only be a key parameter and an optional value parameter. Then, we restrict the effect of these parameters on the control flow inside C.M. Since we do not want the control to depend on specific values, we also limit how the return values of basic collection operations are used. Finally, we preclude certain basic collection operations such as replace(k,v,v') and remove(k,v), which depend on v and the state of the collection. In Section 7.2.3 we relax some of these restrictions to capture a wider set of composed operations.

7.1 Data Independence

In this section we define data-independent composed operations and a set of restrictions that guarantee data-independence. We start by defining the notion of history renaming and use it to define data-independent composed operations. We then define a set of restrictions that guarantees that a composed operation is data-independent.

Definition 7.1.1 (History Renaming) Given a composed operation C.M with key and value parameters, an influence history h of C, and values k, v, and v' ≠ v, we define a history h[k,v,v'] as a history obtained from h by: (i) replacing the input key for each operation in h by k; (ii) replacing the input value for the C.M operation and for each low level operation in h by v; (iii) replacing the input value for each atomic high level operation other than the C.M operation in h by v'; (iv) calculating the return value of operations using the C specification.
Example Figure 7.1 shows a history $h$ of the data-independent composed operation `getLock` from Figure 2.3 and a renamed history $h[l_1^*l_2,l_1]$ achieved by replacing the input key for each operation in $h$ by “$l_1^*”$, the input value for the `getLock` operation and its low-level operations by $l_2$, and the input value for each atomic high-level operation other than the `getLock` operation in $h$ by $l_1$. Moreover, the return value of the `getLock` operation in $h[l_1^*l_2,l_1]$ is $l_1$ according to `getLock` semantics.

We define data-independence over histories because we use it to establish linearizability. In the more general case, this notion can be defined over program traces (extending the formulation of Chapter 5).

Definition 7.1.2 (Data-Independent Composed Operation for Linearizability) A composed operation $C.M$ with parameters key and value is data-independent if for each non-linearizable influence history $h$ of $C$ and for each values $k$, $v$, and $v' \neq v$, $h[k,v,v']$ is a non-linearizable history of $C$.

Corollary 7.1.3 If $C.M$ is data-independent, then $C$ is non-linearizable if and only if there exists a non-linearizable influence history $h$ and values $k$, $v$, and $v' \neq v$ s.t. $h[k,v,v']$ is a non-linearizable influence history of $C$.

Corollary 7.1.3 is implied by Definition 7.1.2 and Lemma 6.1.2. Lemma 6.1.2 guarantees that each non-linearizable composed operation has a non-linearizable influence history and Definition 7.1.2 guarantees that for a data-independent composed operation with an influence history $h$, for each $k$, $v$, and $v' \neq v$, $h[k,v,v']$ is non-linearizable history of $C$.

Example Figure 7.1 shows a history $h$ of the data-independent composed operation `getLock` from Figure 2.3 and a history $h[l_1^*l_2,l_1]$ of $C$. It is easy to see that for each “$l^*$”, $l$, and $l' \neq l$, $h[l^*l',l']$ is a history of $C$.

As we noted before, verifying data-independence is undecidable in the general case [75]. Next, we provide simple sufficient restrictions that can be used to establish a composed operation as data-independent.

### 7.1.1 Rule Based Data Independence

We start by defining rule based data-independent composed operations and continue by showing that rule based data-independent composed operations are indeed data-independent.

Definition 7.1.4 (Data Independence Rules for Linearizability) A composed operation $C.M$ with parameter $k$ and optional parameter $v$, is rule based data-independent if:

1. $k$ is used as a key in all basic collection operations.
2. $k$ and $v$ are immutable: no statement in $C.M$ can assign to $k$ and $v$. 


3. C.M can only invoke the following methods on C: put\((k, \text{val})\), get\((k)\), remove\((k)\), and putIfAbsent\((k, \text{val})\) (here, \(k\) is the parameter \(k\), but \(\text{val}\) need not be the \(v\)).

4. if a return value \(r\) of a collection operation is used in a condition, then the condition can only check the (in)equality of \(r\) to \(\bot\) (this also applies when the return value is assigned to other variables).

5. exit statements such as return and throw can only depend on the return value of collection operations.

Restrictions 1 and 2 guarantee that the composed operation only accesses a single key in the map and in a uniform way. Restriction 3 limits the basic operations used to compose to data-independent operations. The operations put\((k, \text{val})\), get\((k)\), remove\((k)\), and putIfAbsent\((k, \text{val})\) are individually data-independent, since the update and the return value of these operations (appear in Table 3.1) do not depend on a specific value. Note that putIfAbsent\((k, v)\) depends only on the existence of \(k\) in the map and does not depend on a specific value. The challenge is to guarantee that these operations are composed in a way that preserves data-independence. This motivates restriction 4 and 5, which limit control-flow decisions to those that depend on the existence of a value in the collection without referring to a specific value.
Table 7.1: Sample composed operations and their specifications. compute is a data-independent composed operation and Replace is a data-dependent operation. $M : \mathbb{N} \rightarrow \mathbb{N}$ denotes the content of the map.

Table 7.1 shows a data-independent composed operation, named compute, and a data-dependent composed operation, named Replace, together with their update and return value. It is easy to see that both the update and return value of compute do not depend on a specific value. However, in Replace, both the update and return value depend on the input argument $v'$, and thus Replace is data-dependent. Moreover, it easy to see that while compute satisfies the restrictions from Definition 7.1.4, Replace does not satisfy Restriction 4 due to condition $get(k) == v'$ inside the if statement.

Next, we use the restrictions from Definition 7.1.4, the map specification, and the influence specification of maps to state the main theorem for histories: that rule based data-independent composed operations are data-independent. We use the simple case of data-independence where $k$ is used as a key and $v$ is used as a value in all operations. Theories for composed operations satisfying other restrictions can be stated and proved in a similar way.

**Theorem 7.1.1** Given $C$, where $C.M$ is a rule based data-independent operation, and given $h$, a non-linearizable influence history of $C$, then for each input key $k$ and values $v, v' \neq v$, $h_{[k,v,v']}$ is a non-linearizable history of $C$.

Proof: Consider such a non-linearizable influence history $h$. 
We start by building a history using $h$ showing that there exists such an history for each key $k$ and for each pair of values $v$ and $v' \neq v$.

Let $k$ be the input key and let $v$ be the input value for the C.M operation on $h_1$. C.M is a rule based data-independent composed concurrent operation. Therefore, Restriction 1 in Definition 7.1.4 and our assumption guarantee that the input key for every low level operation in $h$ is $k$ and the input value for every updating low level operation in $h$ is $v$. Because $h$ is an influence history every high-level updating operation in $h$ influences a low-level operation. Because every low-level operation in $h$ has an input key $k$, the influence specification of maps (Table 5.1) guarantees that every high-level updating operation in $h$ has an input key $k$. Moreover, because the single non-updating high-level operation in $h$ is influenced by a low-level operation, the influence specification of maps guarantees that every non-updating high level operation in $h$ has an input key $k$.

We conclude that every operation in $h$ has an input key $k$.

C.M is rule based data-independent. Therefore, Restriction 4 in Definition 7.1.4 guarantees that the only branch conditions allowed in C.M are checking whether the return value of operations is (or is not) equal to $\bot$. The influence specification of maps guarantees that an operation $op_1$ influences operation $op_2$ if $op_1$ and $op_2$ have the same input key and $op_1$ writes a value that differs from the one returned by $op_1$. C.M is restricted to the data-independent operations $\text{put}(k,v)$, $\text{get}(k)$, $\text{remove}(k)$, and $\text{putIfAbsent}(k,v)$, which guarantees that the update performed by these operations do not depend on a specific value. Therefore, setting the value in all high level operations in $h$ to a value $v'$ that differs from $v$ and calculating a new return value for the C.M operation using the C specification, guarantees that the generated history, marked $h'$, is a non-linearizable history of C.

Note that the only non-determinism in calculating the value of the C.M operation is the return value of the base operation. Moreover, Restriction 5 in Definition 7.1.4 guarantees that exit statements such as $\text{return}$ and $\text{throw}$ can only depend on the return value of collection operations. Thus, because in C.M the return value of base operations is only checked for (in)equality to $\bot$, then the return value of the C.M operation in $h$ differs from the return value of the C.M operation in $\text{lin}(h)$ if and only if the return value of the C.M operation in $h'$ differs from the return value of the C.M operation in $\text{lin}(h')$.

Also note that when replacing the input value for each high level atomic operation by $v'$, a high level operation might not influence the next low-level operation of the C.M operation. This occurs because all high level operations writes the same value. This does not affect the non-linearizability of the history because all branch conditions only check the return value of an operation for (in)equality to $\bot$ and also the update of $\text{put}(k,v)$, and $\text{remove}(k)$ are not influenced by any operation, and the update of $\text{putIfAbsent}(k,v)$ is influenced only by the (in)equality of $\text{get}(k)$ to $\bot$.

Next, we discard from $h'$ all high-level operations that do not influence the next low-level operation. This results in a non-linearizable influence history of C, $h_{[k,v,v']}$. We now show that for each $k_1$, $v_1$, and $v'_1 \neq v_1$, $h_{[k_1,v_1,v'_1]}$ is a non-linearizable history of C.
Because all operations in \( h[k,v,v'] \) have the same input key \( k \), we can generate a new history \( h[k_1,v,v'] \) by calculating the return value of the C.M operation using the C specification. Note that the map specification and the non-determinism restriction of C (previously noted) guarantee that \( h[k_1,v,v'] \) is a non-linearizable history of C.

Because all low-level operations in \( h[k,v,v'] \) have the same input value \( v \), we can generate a new history \( h[k_1,v_1,v'] \) by calculating the return value of all operations using the C specification. Note that the map specification, the influence specification of maps, and the non-determinism restriction of C (previously noted) guarantee that \( h[k_1,v_1,v'] \) is a non-linearizable history of C.

Because all high-level operations in \( h[k,v_1,v'] \) have the same input value \( v' \), we can generate a new history \( h[k_1,v_1,v'] \) by calculating the return value of all operations using the C specification. Note that the map specification, the influence specification of maps, and the non-determinism restriction of C (previously noted) guarantee that \( h[k_1,v_1,v'] \) is a non-linearizable history of C.

We showed that given a composed operation C, where C.M is a rule based data independent, and given \( h \), a non-linearizable influence history of C, for each input key \( k \) and values \( v,v' \neq v \), \( h[k,v,v'] \) is a non-linearizable history of C.

\[\square\]

Theorem 7.1.1 and Definition 7.1.2 imply the following corollary.

**Corollary 7.1.5** A rule based data-independent composed operation C.M is data-independent.

Now we are ready to state the main correctness theorem for traces. In order to state the theorem we define a program trace \( \pi = (l_0, c_0) \xrightarrow{s} (l_1, c_1) \ldots \xrightarrow{s} (l_n, c_n) \) as limited by \( C_0 \) if \( \forall i.0 \leq i \leq n.c_i \in C_0 \).

The following main correctness theorem is for data-independent composed operations with an input key and an input value. Similar theorems can be stated for data-independent composed operations without an input value and for data-dependent composed operations, defined in Section 7.2, that can be proved as linearizable using a bounded map.

**Theorem 7.1.2** If C.M is data-independent with a key parameter \texttt{key} and a value parameter \texttt{val}. Then, C is non-linearizable w.r.t. linearization point \( lp \) if and only if for every set \( C_0 = \{\emptyset, \langle k, v \rangle, \langle k, v' \rangle\} \), where \( v \neq v' \), there exists a non-linearizable program trace limited by \( C_0 \) starting from a state \( (l_0, \emptyset) \) where pc(\( l_0 \)) is the initial location, \texttt{key}(l_0) = k, and \texttt{val}(l_0) = v.

\[\Rightarrow\]

Consider C a non-linearizable custom map with a data-independent composed operation C.M.

Lemma 6.1.2 guarantees that C has a non-linearizable influence history \( h \). Definition 7.1.2 guarantees that for each such non-linearizable influence history \( h \) of C and for each values \( k, v, \) and \( v' \neq v \),
$h_{[k,v,v']} \text{ is also a non-linearizable influence history of } C$. Lemma 6.2.1 guarantees that for each such non-linearizable influence history $h = h_{[k,v,v']}$ there exists a non-linearizable trace $\pi$ where $h = h_{\pi}$.

Note that following the translation from Section 6.2.1, $\pi$ and $h$ has the same high level operations and each operation has the same argument. Therefore, $\pi$ starts from a state $\langle l_0, \emptyset \rangle$ where $pc(l_0)$ is the initial location, $\text{key}(l_0) = k$, and $\text{val}(l_0) = v$, as in $h_{[k,v,v']}$.  

Note also that because all map operations in $\pi$ use the same input key $k$ and either a value $v$ or a value $v'$. The map specification from Table 3.1 guarantees that $\pi$ is limited by $\{\emptyset, \langle k, v \rangle, \langle k, v' \rangle\}$.

Consider a non-linearizable trace $\pi$. Lemma 6.2.3 guarantees that there exists a non-linearizable history of $C$. Therefore, using Definition 3.4.1 we conclude that $C$ is not linearizable.

\[\blacksquare\]

Theorem 7.1.2 implies the following corollary.

**Corollary 7.1.6** Checking linearizability of data-independence composed operations can be done using a fixed set of map entries.

Note that the complexity of the exploration does not depend on the size of the local key parameter and the size of the value local parameter. In other words, for programs in which these are the only infinite values, the problem becomes decidable.

Also note that Theorem 7.1.2 shows that verification of linearizability of data-independent composed operations can be done using any key and any value. Therefore, verifying linearizability of data-independent composed operations can be done by exploring all traces generated by the transition relation from Section 5.2.1 using a single input key and a single input value and checking whether the specification from Definition 5.2.2 holds in these traces. This can be done using a model checker such as SPIN.

### 7.1.2 Example

We now show a linearizable example, together with all the traces explored by our algorithm. Figure 7.2 shows a corrected version of the method `doubleNonLin` from Figure 5.1 called `doubleLin`. Figure 7.3 shows all the runs generated by our algorithm on `doubleLin`. Because `doubleLin` is an SCM, we used only a single input to verify the linearizability of custom collection built by `doubleLin`. In addition, we restricted the environment to write a single value other than the one calculated by `doubleLin`, in this case the value 3 was used. It is easy to see that in each trace the return value is the same as the return value returned at the sequential trace. Therefore, since all runs generated by our algorithm using a single input key meet the condition of Definition 5.2.2, we conclude that `doubleLin` is linearizable. We use a variant of Theorem 7.1.2 where the input value is calculated inside the method.
```java
int doubleLin(K) {
    val = m.get(K);
    if (val == null) {
        nv = K*2;
        val = m.putIfAbsent(K, nv);
        if (val == null)
            val = nv;
    }
    return val;
}
```

Figure 7.2: A possible linearizable fix for the example from Figure 5.1. This example is an SCM.

Figure 7.3: The traces generated by our algorithm, for `doubleLin` method from Figure 7.2. A state is represented by a box, a linearization point is represented by a black box, A state of the sequential run is represented by a circle, an arrow represents a transition, and a dashed arrow represents an environment transition together with its operation.
7.2 Composed Operations Classes

7.2.1 Singleton Collection Method

We refer to the class of data-independent composed operations as **Singleton Collection Methods (SCM)**.

Figure 2.3 shows a composed operation taken from Apache ServiceMix [3] named `getLock`. This composed operation is data-independent. The operation uses the underlying `ConcurrentHashMap` locks to memoize the `ReentrantLock` allocated at line 4. When the value for a given `id` is cached in the collection it is returned immediately; when the `ReentrantLock` for a given `id` is not available, it is allocated and inserted into the collection.

This composed operation meets all the conditions of Definition 7.1.4 and is therefore an SCM. It has a single input `id` which is used only as a key in the `ConcurrentHashMap` locks.

And the return value of the collection operation is only checked for (in)equality to `null` (lines 3 and 7).

7.2.2 Fixed Collection Method

We generalize the class SCM to deal with some programs dependent on a finite number of values. We refer to this more general class as **Fixed Collection Methods (FCM)**.

We generalize SCM by weakening the restriction that exit statements and collection operations are not control-dependent on expressions using `k`. In FCM, the restriction permits exit statements and collection operations to be control-dependent on expressions using `k` when the expression compares `k` to a constant value. We use these expressions to build a set of inputs `I` for the FCM by adding all constants, as well as, one additional value that is not in `I`.

Figure 7.4 shows an example of a method `forOsFamily` taken from Autoandroid [5]. This method returns an instance of AndroidTools suitable for the given operating system. This method either returns a new instance and updates the `ConcurrentHashMap androidTools` or returns an existing instance from `androidTools`. AutoAndroid supports only tools for the “unix” and “windows” operating systems, therefore, lines 6 – 13, check whether the input string is either “unix” or “windows” and throws an exception otherwise.

Clearly, this function is not an SCM because there is an abnormal exit statement (throw at line 11) that is control dependent on the input key `osFamily` (lines 6 and 8). However, the input key `osFamily` is only compared to the constants “unix” and “windows”. Moreover, `forOsFamily` meets all the other SCM requirements. Therefore, `osFamily` is an FCM and the fixed input set is the set `{ “unix”, “windows”, X }` s.t. `X` is some value other than “unix” and “windows”.
```java
AndroidTools forOsFamily(String osFamily) {
    AndroidTools instance = androidTools.get(osFamily);

    if (instance == null) {
        AndroidTools newInstance = null;
        if (osFamily.equals("windows")) {
            newInstance = new WindowsAndroidTools();
        } else if (osFamily.equals("unix")) {
            newInstance = new UnixAndroidTools();
        } else {
            throw new UnsupportedOperationException("...");
        }

        instance = androidTools.putIfAbsent(osFamily, newInstance);
        if (instance == null)
            instance = newInstance;
    }

    return instance;
}
```

Figure 7.4: An example, taken from AutoAndroid [5] which is an FCM and not an SCM
7.2. COMPOSED OPERATIONS CLASSES

1 void inc(Class<? key) {
2 for (;;) {
3     Integer i = get(key);
4     if (i == null) {
5         if (putIfAbsent(key, 1) == null) //LP if succeeds
6             return;
7     } else {
8         if (replace(key, i, i + 1)) //LP if succeeds
9             return;
10     }
11 }
12 }

Figure 7.5: VCM example, taken from OpenJDK [15]

7.2.3 Value Collection Method

We generalize SCM to deal programs with replace(k,v,v’). We refer to this class as Value Collection Methods (VCM).

We generalize SCM by allowing replace(k,v,v’) to be used in a composed operation. In VCM, the restriction permits replace(k,v,v’) to appear in the composed operation as long as v is a value returned by a get(k) operation.

Figure 7.5 shows an example of a method inc taken from OpenJDK [15]. This method implements an increment operation for a concurrent histogram (as a map of counters). This function is not an SCM because it has a replace(k,v,v’) operation at line 8.

However, this method is a VCM because the value i used as v in line 8 is extracted from the collection at line 3.

7.2.4 Example of a Data-Dependent Method

Figure 7.6 shows part of a method getInstance taken from fleXive [8]. This method takes an input language and when the input is null returns a predefined FxValueRenderer corresponding to the key DEFAULT in the ConcurrentHashMap renderers (lines 2–3). Otherwise, the method either returns a new object FxValueRenderer and updates the renderers or returns an existing FxValueRenderer from renderers (lines 5–9).

This method is not linearizable, but can be fixed to be linearizable (see Figure 8.4 in Chapter 8). However, even its linearizable version is neither SCM nor FCM. The reason is that in line 3 there is an access to renderers with the key DEFAULT which might be other than the value of language.
Figure 7.6: A data-dependent method example, taken from fleXive [8]
Chapter 8

Experimental Evaluation

In this chapter we explain the COLT implementation and evaluate COLT’s effectiveness in checking the linearizability of composed concurrent operation on a range of real-world applications. Using our approach, we show that all linearizable real-world composed operations that we identified are either SCMs, FCMs, or VCMs and can be easily verified by our technique. Moreover, the large number of violations we found using the COLT tool substantiates our hypothesis that programmers often make incorrect assumptions when using concurrent collections. Together with each violation, we reported a suggested fix to the development team. In many cases, our fixes were accepted by the development team and incorporated in the application.

8.1 Implementation

An outline of the COLT implementation is shown in Figure 8.1. The programmer provides a multithreaded program to the Composed Operation Extractor. The Composed Operation Extractor uses a simple static analysis that identifies methods that syntactically include multiple collection operations. In most cases (71% of our examples) the Composed Operation Extractor extracts a method that is a composed operation and is provided to the FCM VERIFIER. However, in some cases the Composed Operation Extractor identifies a specialized concurrent data structure operation inside a large method and the user needs to manually extract and generate the composed operation (this occurred in the remaining 29% of our examples).

The composed operation (CO) is then provided to the FCM VERIFIER. This module checks whether the given composed operation is an SCM, an FCM, a VCM, or data dependent. If it is an FCM, the FCM VERIFIER also provides a set of inputs. This part is currently manual; however, it can be implemented using a simple CFG analysis.

Our tool requires an additional specification of (conditional) linearization points.

When the FCM VERIFIER returns a data-dependent result, we automatically generate drivers that are
given to the bytecode instrumentor. The bytecode instrumentor instruments the CO and generates our linearizability tester using the library’s influence specification. Then, the instrumented CO is repeatedly executed until either a linearizability violation is found or a time bound is exceeded.

When the FCM verifier returns an SCM, an FCM, or a VCM result, we automatically generate PROMELA models, which are fed to SPIN in order to verify linearizability.

This can be done for methods with primitive keys and values. However, for composed operations with more complex keys or values, we require that the programmer provides an input driver for generating the actual values for keys and values. In addition, in this case, the user provides an influence driver that gets an input object and returns a single different object. This driver is used by the environment of our technique.

Note that for SCM, the input driver needs to provide only an arbitrary key and an arbitrary value and the influence driver needs to provide another value which differs from the one provided by the input driver. Also note that the sequential specification we used for checking linearizability is by executing the custom collection operations in an atomic manner.

8.2 Applications

Table 8.1 lists 57 real-world applications using Java’s concurrent collections. In many cases concurrent collections were introduced to address observed scalability problems, replacing manual locking of a sequential map. Each of the applications contains at least one method that was extracted and tested by our tool. The methods were extracted using the Composed Operation Extractor, which identifies methods that syntactically include multiple collection operations. In 33 out of the 112 extracted methods, the Composed Operation Extractor identified a composed operation inside a large method and we manually extracted and generated the composed operation. The extracted methods together with explanations for each method can be found at [6].

Table 8.1: Applications used for experiments

<table>
<thead>
<tr>
<th>Program</th>
<th>LOC</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adaptive Planning</td>
<td>1,103,453</td>
<td>Automated budgeting tool</td>
</tr>
<tr>
<td>Adobe BlazeDS</td>
<td>180,822</td>
<td>Server-based Java remoting</td>
</tr>
<tr>
<td>Amf-serializer</td>
<td>4,553</td>
<td>AMF3 messages serializaton</td>
</tr>
<tr>
<td>Annsor</td>
<td>1,430</td>
<td>runtime annotation processor</td>
</tr>
<tr>
<td>Apache Cassandra</td>
<td>54,470</td>
<td>Distributed Database</td>
</tr>
<tr>
<td>Apache Derby</td>
<td>618,352</td>
<td>Relational database</td>
</tr>
</tbody>
</table>

Continued on Next Page...
<table>
<thead>
<tr>
<th>Program</th>
<th>LOC</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache MyFaces Trinidad</td>
<td>201,130</td>
<td>JSF framework</td>
</tr>
<tr>
<td>Apache ServiceMix</td>
<td>78,340</td>
<td>Enterprise Service Bus</td>
</tr>
<tr>
<td>Apache Struts</td>
<td>110,710</td>
<td>Java web apps framework</td>
</tr>
<tr>
<td>Apache Tomcat</td>
<td>165,266</td>
<td>Java Servlet and Server Pages</td>
</tr>
<tr>
<td>Apache Wicket</td>
<td>142,968</td>
<td>Web application framework</td>
</tr>
<tr>
<td>ApacheCXF</td>
<td>311,285</td>
<td>Services Framework</td>
</tr>
<tr>
<td>Autoandroid</td>
<td>19,764</td>
<td>Tools for automating android projects</td>
</tr>
<tr>
<td>Beanlib</td>
<td>42,693</td>
<td>Java Bean library</td>
</tr>
<tr>
<td>Carbonado</td>
<td>53,455</td>
<td>Java abstraction layer</td>
</tr>
<tr>
<td>CBB</td>
<td>16,934</td>
<td>Concurrent Building Blocks</td>
</tr>
<tr>
<td>Clojure</td>
<td>25,421</td>
<td>dynamic programming language for the JVM</td>
</tr>
<tr>
<td>cometdim</td>
<td>5,571</td>
<td>A web IM project</td>
</tr>
<tr>
<td>Daisy</td>
<td>334,337</td>
<td>Content and information management</td>
</tr>
<tr>
<td>Direct Web Remoting</td>
<td>26,094</td>
<td>Ajax for Java</td>
</tr>
<tr>
<td>dyuproject</td>
<td>26,593</td>
<td>Java REST framework</td>
</tr>
<tr>
<td>Ehcache Annotations for Spring</td>
<td>3,184</td>
<td>Automatic integration of Ehcache in spring projects</td>
</tr>
<tr>
<td>Ektorp</td>
<td>6,261</td>
<td>Java API for CouchDB</td>
</tr>
<tr>
<td>EntityFS</td>
<td>79,820</td>
<td>OO file system API</td>
</tr>
<tr>
<td>eXo</td>
<td>13,298</td>
<td>Portal</td>
</tr>
<tr>
<td>FindBugs</td>
<td>106,031</td>
<td>Static analysis tool</td>
</tr>
<tr>
<td>fleXive</td>
<td>910,780</td>
<td>Java EE 5 content repository</td>
</tr>
<tr>
<td>GlassFish</td>
<td>260,461</td>
<td>JavaServer faces</td>
</tr>
<tr>
<td>Granite</td>
<td>28,932</td>
<td>Data services</td>
</tr>
<tr>
<td>gridkit</td>
<td>8,746</td>
<td>Kit of data grid tool and libs</td>
</tr>
<tr>
<td>GWTEventService</td>
<td>17,113</td>
<td>Remote event listening for GWT</td>
</tr>
<tr>
<td>Hazelcast</td>
<td>59,139</td>
<td>Data grid for Java</td>
</tr>
<tr>
<td>Hudson</td>
<td>14,991</td>
<td>Automatic build system</td>
</tr>
<tr>
<td>hwasjtim</td>
<td>4,371</td>
<td>A Struts plugin for Java</td>
</tr>
<tr>
<td>ifw2</td>
<td>54,888</td>
<td>Web application framework</td>
</tr>
<tr>
<td>Jack4j</td>
<td>4,477</td>
<td>Interface to Jack Audio Connection Kit (JACK) library</td>
</tr>
<tr>
<td>JBoss AOP</td>
<td>1,013,073</td>
<td>Aspect oriented framework</td>
</tr>
<tr>
<td>Jetty</td>
<td>64,039</td>
<td>Java HTTP servlet server</td>
</tr>
</tbody>
</table>

Continued on Next Page...
Table 8.1 – continued from previous page

<table>
<thead>
<tr>
<th>Program</th>
<th>LOC</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jexin</td>
<td>11,024</td>
<td>functional testing platform</td>
</tr>
<tr>
<td>JRipples</td>
<td>148,473</td>
<td>Program analysis</td>
</tr>
<tr>
<td>JSefa</td>
<td>27,208</td>
<td>Object serialization library</td>
</tr>
<tr>
<td>keyczar</td>
<td>4,720</td>
<td>Cryptography Toolkit</td>
</tr>
<tr>
<td>memcache-client</td>
<td>4,884</td>
<td>Memcache client for Java</td>
</tr>
<tr>
<td>Module Glassfish</td>
<td>745</td>
<td>Module system for Glassfish</td>
</tr>
<tr>
<td>OpenEJB</td>
<td>191,918</td>
<td>Server</td>
</tr>
<tr>
<td>OpenJDK</td>
<td>1,634,818</td>
<td>JDK 7</td>
</tr>
<tr>
<td>P-GRADE</td>
<td>1,154,884</td>
<td>P-GRADE Grid Portal</td>
</tr>
<tr>
<td>Project Tammi</td>
<td>163,913</td>
<td>Java development framework</td>
</tr>
<tr>
<td>Project Track</td>
<td>5,160</td>
<td>Example application</td>
</tr>
<tr>
<td>RESTEasy</td>
<td>81,586</td>
<td>Java REST framework</td>
</tr>
<tr>
<td>Retrotranslator</td>
<td>27,315</td>
<td>Automatic compatibility tool</td>
</tr>
<tr>
<td>Streamy</td>
<td>483,418</td>
<td>Audio/video/network recorder</td>
</tr>
<tr>
<td>Tersus</td>
<td>165,160</td>
<td>Visual Programming Platform</td>
</tr>
<tr>
<td>torque-spring</td>
<td>2,526</td>
<td>Torque support classes</td>
</tr>
<tr>
<td>Vo Urp</td>
<td>24,996</td>
<td>UML data models translator</td>
</tr>
<tr>
<td>WebMill</td>
<td>57,161</td>
<td>CMS portal</td>
</tr>
<tr>
<td>Xbird</td>
<td>196,893</td>
<td>XQuery processor &amp; XML db</td>
</tr>
<tr>
<td>Yasca</td>
<td>326,502</td>
<td>Program analysis tool</td>
</tr>
</tbody>
</table>

8.3 Results

We tested 112 methods in 57 applications. All of our experiments were carried out using AMD Opteron 2.4GHz dual hyper threaded CPUs, 8GB RAM platform running on a 64 bit Linux. The runtime for analyzing each method took less than one CPU second. COLT verified 31 composed operations as linearizable and found 59 linearizability violations.

Figure 8.2 shows that 51 methods are data-independent. Out of these, 49 are SCM, 1 is in a special class of data-independent (VCM) methods, and 1 depends on fixed input keys (FCM). As the figure shows, 31 methods were verified as linearizable, and a linearizability violation was detected in the other 20 methods. We then manually fixed the linearizability violations in all of these 20 methods and verified that their fixed versions are linearizable.

Figure 8.2 also shows that 61(54%) of the tested methods are data-dependent. The data-dependent
Figure 8.1: COLT overview
Figure 8.2: Classification of 112 benchmark methods into SCM, VCM, FCM, and data dependent methods, and classification into linearizable and non-linearizable.
8.3. Results

Figure 8.3: Methods, out of the 112, that are either linearizable or can be fixed to be linearizable in each one of our benchmarks.

methods cannot be verified by our technique. However, as the figure shows, a close inspection revealed that all the data-independent methods are non-linearizable. Moreover, it is not clear whether these methods can be re-written as equivalent linearizable methods because, as Section 8.3.2 shows, the results of most of these methods depend on the values of global variables. Thus, all the linearizable methods can be verified by our technique.

We manually fixed 3 of the non-linearizable methods. For example, Figure 8.4 shows a fixed version of the method from Figure 7.6. This composed operation was fixed by taking into account the value returned by \texttt{putIfAbsent} at line 10 in determining the value returned by \texttt{getInstance}.

Figure 8.3 shows the distribution of the data-independent methods that are linearizable, or can be fixed to be linearizable in each of our benchmarks. In 22 out the 57 applications we automatically verified the linearizability of all composed operations. The remaining 35 applications also include data-dependent methods that cannot be verified by our technique.

8.3.1 Linearizability Violations

Using COLT’s testing procedure, we found linearizability violations in 59 methods. Figure 8.5 shows that 17(29\%) of these methods were not linearizable in an open environment, but were safe when implicit (and unchecked) application invariants were taken into account. Moreover, 42(71\%) of these methods had atomicity violations that could be triggered in the current application.
FxValueRenderer getInstance(FxLanguage language) {
    if (language == null) {
        return renderers.get(DEFAULT);
    }

    FxValueRenderer oldV = renderers.get(language);
    if (oldV == null) {
        FxValueRenderer V = new FxValueRendererImpl(language);
        FxValueRenderer oldV = renderers.putIfAbsent(language, V);
    }
    if (oldV == null)
        oldV = V;
    return oldV;
}

Figure 8.4: The data-dependent getInstance method taken from fleXive, fixed to be linearizable.

Figure 8.5: Composed operations distribution per result. As the graph shows, 42 composed operations are non-linearizable in their current applications and 17 composed operations are non-linearizable only in an open environment.
Figure 8.6 shows the violation reported by COLT for each benchmark. The ‘X’ axis shows the application from Table 8.1 and the ‘Y’ axis shows the number of composed operations 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 client environment. “Modular NL” shows the methods in our experiments that are non-linearizable in an open environment but are linearizable in the client’s current environment.

For example, Apache Tomcat includes 10 methods that are non-linearizable in an open environment as well as in the client environment. For all of the non-linearizable methods (“Non-Lin” and “Modular NL”), COLT reported an interleaving in which the composed operation 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 environment no longer reports the same violation.

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 includes 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, GridKit, 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 identified violations in 10 out 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.

**Naive Environment Thread** Testing with a naive environment thread that does not use influence specification failed to find a single violation within a timeout of 10 hours per method. In contrast, COLT reported a violation for all of these non-linearizable methods in less than one CPU second.

### 8.3.2 Benchmark Classification

Figure 8.7 shows the classification of our 112 data-dependent and data-independent composed operations according to the data-independent rules they falsify or satisfy.
Figure 8.7: Classification of our 112 data-dependent and data-independent composed operations according to the rules they falsify or satisfy.
1 public HttpDestination getDestination(Address remote, boolean ssl) {
2     if (remote == null)
3         throw new UnknownHostException("Remote socket address cannot be null.");
4
5     HttpDestination destination = _destinations.get(remote);
6     if (destination == null)
7         {  
8             destination = new HttpDestination(this, remote, ssl, _maxConnectionsPerAddress);
9             if (proxy != null && (_noProxy == null || !_noProxy.contains(remote.getHost())))
10                {  
11                    destination.setProxy(proxy);
12                    if (proxyAuthentication != null)
13                        destination.setProxyAuthentication(proxyAuthentication);
14                }
15     }
16     HttpDestination other = _destinations.putIfAbsent(remote, destination);
17     if (other!=null) destination=other;
18 }
19 return destination;
20 }

Figure 8.8: A non-linearizable composed operation from Granite. The result of the operation depends on the global variables _proxy and _noProxy.

The classification of the data-dependent operations is shown in the box on the right and is according to the violated data-independent rules. Out of the 112 composed operations, 61 are data dependent. 41 of these are not encapsulated and include global variables (marked as “globals”). These operations are non-linearizable in an open environment. Moreover, it is not obvious whether these methods can be rewritten as equivalent linearizable methods because the result of these methods depend on the values of global variables. For example, Figure 8.8 shows a method from the Granite benchmark, where the result depends on the global variables _proxy and _noProxy (note the condition in line 10). An environment that changes the value of _proxy affects the value of destination, calculated in line 12.

5 of the data-dependent operations contain conditional remove operation (marked as “remove”), 6 may exit based on the key argument (marked as “exit K”), and 4 access the map with two different keys (marked as “Two K”). An example where the map may access with two different keys is the function getInstance shown in Figure 7.6. This method accesses the DEFAULT key in line 3 and the language key in lines 6 and 10. Finally, in 2 of the data-dependent composed operations, a map operation might be control dependent on the key (“branch K”), in 2 a size operations exists (“size”),
8.4 Conflict Serializability vs. Linearizability

None of the 112 application methods we checked are conflict serializable [40]. The conflict serializability checker reports violations 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.

8.5 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 due to duplication, the possibility that the computation might fail, etc. Influenced by this thesis, the interface of Java concurrent collections is being enriched with a new operation that supports the most common functionality of this recurring example [7, 13].

Figure 8.9 shows the desired functionality of a function compute(K k) that memoizes the result of calculateVal(k) (note that Java does not actually have an atomic keyword). Figure 8.10 shows some of the implementations that we encountered for compute, including the buggy version from Figure 2.2a.

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

Figure 8.9: The desired functionality of a function `compute(K k)` that memoizes the result of `calculateVal(k)` (note that Java does not actually have an `atomic` keyword)

inserted into the collection.

Figure 8.10(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. Another reason is that `calculateData` can often be time consuming or cause a side effect, and hence should not be executed more than once.

The implementation in Figure 8.10(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 `putIfAbsent`. In cases where `putIfAbsent` succeeds in updating `m`, a `null` value is returned by `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 8.10(iv), Figure 8.10(v), and Figure 8.10(vi) are common non-linearizable implementations of Figure 8.10(ii) and the implementation in Figure 8.10(iii) is a common non-linearizable implementation of Figure 8.10(i).

An interleaving that reveals the non-linearizability of the implementation in Figure 8.10(iv) occurs when `remove(k)` with the same key `k` is executed between `putIfAbsent(k, *)` and `get(k)` of `compute(k)`. In this case `compute(k)` return a null result. A similar interleaving can reveal the non-linearizability of Figure 8.10(iii) and Figure 8.10(vi) where a `remove(k)` operation occurs between `putIfAbsent(k)` and `get(k)`.

An interleaving that reveals the non-linearizability of Figure 8.10(v) occurs when `put(k, *)` with
the same key \( k \) is executed between \( \text{if} \ (\text{val} == \text{null}) \) and \( \text{putIfAbsent} \) of \( \text{compute} \). In this case, \( \text{putIfAbsent} \) fails and the value returned by \( \text{compute} \) is not the value corresponding to \( k \) in the map \( m \).

### 8.5.1 Advanced Example

When the \( \text{calculateVal} \) function is either time consuming or causes a side effect, the code in Figure 8.10(vii) is used. This implementation uses Java’s \( \text{Future} \) construct to guarantee that function \( \text{calculateData} \) executed only once for each added key. This way only one thread is responsible for the calculation while the others may block until the calculation completes. If \( \text{Future} \) is canceled, this \( \text{Future} \) is removed from the map and a new memoization iteration continues until another \( \text{Future} \) terminates successfully and its value is returned. A linearization point for this implementation occurs.
when \( f.\text{get()} \) returns successfully (marked by @LP (f.isDone())) due to the fact that the \( f \) calculation terminated successfully. The reason this is that there might be a case where another thread canceled \( f \)'s execution, as a result of which the \texttt{compute} execution should continue.

Even though the implementation in Figure 8.10(vii) is linearizable, COLT reports a violation. The problem is that COLT is not aware of \texttt{Future} semantics and thus it is also not aware that it should treat the value 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 \texttt{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 not encountered such examples in practice.

### 8.5.2 Benchmark Distribution

Figure 8.11 shows the distribution of the different kinds of composed concurrent operations (as given in Figure 8.11) in our benchmark.

Overall 74% of the composed operations we checked were memoization examples. The most common bug pattern (17%) was the implementation in Figure 8.10(v), followed by the implementation in Figure 8.10(iv) (8%).

### 8.6 Reasons for Success

#### 8.6.1 Source of Bugs

Most of the composed operations were originally written using a concurrent collection. However, in some cases, the composed operation 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 composed operations were refactored in a non-linearizable manner.

#### 8.6.2 Bug Characteristics

Figure 8.10 reveals that these implementations, even in cases where they are not data-independent, behave uniformly on most input keys. This characteristic occurs in all our checked composed operations and implies that if the composed operation is not linearizable then, for most keys, there exists an interleaving revealing its non-linearizability.

The fact that programmers tend to write buggy specialized concurrent collections and that their corresponding composed operations are mostly uniform significantly eases bug detection in real code. The
Figure 8.11: The distribution of our 112 composed operations of the types shown in Figure 8.10. Type (i) is missing because it is not a composed operation.

uniformity characteristic implies that bugs can be detected using most input keys so long as the environment thread performs the right operation with the right interleaving. Our influence-aware environment thread easily detects these non-linearizable interleavings.
Chapter 9

Related Work

This Chapter surveys some of the work related closely to the results in this thesis.

9.1 Dynamic Atomicity Checking

Dynamic atomicity checkers such as [39, 40] check for violations of conflict serializability. As noted earlier, conflict-serializability is inappropriate for concurrent data structures. Therefore, in this thesis we check for violations of linearizability. Vyrd [37] is a dynamic checking tool that checks a property similar to linearizability. It requires manual annotation of linearization points. Line-Up [27] is a dynamic linearizability checker that enumerates schedules. The formal result of [38] 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 [21].

In contrast with COLT, all of these existing tools are non-modular and not directed using influence specification. As shown in Chapter 8, when influence information is not utilized, the ability to detect collection-related atomicity violations remains low. In fact, the poor results of the random adversary in Chapter 8 are obtained when underlying collection operations are considered atomic. Moreover, unlike COLT, which can prove linearizability on data-independent programs manipulating maps, these methods may produce false negatives for that class of programs.

GAMBIT [32] is a unit testing tool for concurrent libraries built on top of the CHESS tool [61]. GAMBIT employs 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 works 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 influence to guide the scheduling to quickly reveal bugs. From the above reasons GAMBIT is unlikely to detect the collection-related atomicity violations. Moreover, unlike COLT, which can prove linearizability on
data-independent programs manipulating maps, GAMBIT may produce false negatives for that class of programs.

An active testing technique for checking conflict serializability is presented in [62]. 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 influence. Therefore, its ability to detect collection-related violations would likely be low. Moreover, unlike COLT, which can prove linearizability on data-independent programs manipulating maps, this technique may produce false negatives for that class of programs.

9.2 Static Checking and Verification

A manual proof of correctness of several interesting concurrent data structure implementations using rely-guarantee reasoning is presented in [71].

The PVS system has been successfully used to semi-automatically verify linearizability [31, 36, 42] of several interesting programs. These proofs provide crucial insights on essential points of the algorithm. However, this is a very time consuming task that needs to be repeated for each new program. In contrast, our approach works on a restricted class of programs but is much more automatic.

The work of [19] introduced the idea of using abstract interpretation [33] to develop an automatic over-approximation for checking linearizability. Thus, the algorithm can prove linearizability in certain programs but may fail due to overly conservative abstraction. The algorithm assumes that the concurrent and sequential runs differ by at most one element, thereby drastically simplifying the task of checking linearizability. The work in [23, 57] generalizes [19] using a thread-centric approach to verify programs with an unbounded number of threads. In [70], the idea of bounded difference is combined with rely guarantee reasoning and shape abstractions to perform fast linearizability checks. In contrast, the work in this thesis is not only sound but also guaranteed to be complete for data-independent collection manipulations. Note that we prove that linearizability violations can be observed using maps with at most one key, which implies bounded differences.

Works such as [73] focus on model checking individual collections. This thesis operates at a higher level, as a client of already verified collections, and leverages the specifications of the underlying collections and the composed operation’s data-independent characteristics to reduce the search space. Also, our proofs bound the global state without any over-approximation.

Recently, it was shown in [28] that checking linearizability is decidable for programs with simple linked-list manipulations. We focus on collections that hide the internal representation and use data independence to show that the state space reduces to the state space of the local state. This implies decidability for finite-state local states and is effective for all the data independent programs we have encountered.
9.3 Effective Techniques for State Space Reduction

This thesis employs several techniques for reducing the state space to effectively prove linearizability. We use the single updating property which is unique to maps. We use data independence, which was inspired by [75], to bound the size of the global state space.

9.3.1 Partial Order Reduction

Partial order reduction techniques such as [43] utilize commutativity of individual memory operations to filter out execution paths which cannot lead to new violations. While this technique has proven to be very effective for drastically reducing the state space in explicit state model checking, it is in general difficult to infer commutativity for real life software. This thesis follows [64] and uses the fact that the abstract interface of the collection is known at library design time to build an effective tool which incorporates influence checking. Indeed, we show that specializing to collections, using partial order reduction and data independence, leads to effective linearizability checking.

9.3.2 Thread-Centric Approach

The idea of using a general client over-approximating the thread environment is common in modular verification. Previous work represents the environment as invariants [48] or relations [51] on the shared state. This idea has also been used early on for automatic compositional verification [30]. In addition, this approach has led to the notion of thread-modular verification for model checking systems with finitely-many threads [41], and has also been applied to the domain of heap-manipulating programs with coarse-grained concurrency [45]. The main idea in these works is to approximate the thread environment.

In contrast to these approaches, we prove in Theorem 6.2.1 that the thread-centric approach can be employed in our situation without losing precision. Moreover, we leverage data independent characteristics of the composed operation to restrict the inputs and the environment.
Chapter 10

Future Work

In this work we focused on testing and verifying linearizability of concurrent maps. However, this work can be extended to other data structures besides maps. Extending this work to other data structures is currently conducted in Mooly Sagiv’s research group by Ariel Jarovsky. The basic idea of that work is to identify the ADT characteristic that our technique exploits in order to easily prove the linearizability of this custom ADT.

The extension to other data structures is not only dependent on the data structure itself but also on the operations that its interface exposes. For example, in this work we preclude the usage of size operation. The reason is that proving linearizability of a composed operation that uses size cannot be done using a bounded global state. This is because the size of the map might influence the control flow of the composed operation and therefore, one cannot know how many keys should be added to the map in order to explore all possible traces that may lead to a linearizability violation.

Another operation we preclude in this work is the replace(K,V,V’) operation. This operation is similar to compare-and-swap and assigns the value V’ to the key K only if in the current collection state V is assigned to K. Therefore, replace updates the collection state only if in the current state a specific value is assigned to the key K. For this reason, one cannot explore all possible traces that might lead to linearizability violation using any key and any value.

For the above reasons, extending this work for multi-map interface that exposes a size operation that either returns the number of keys or the number of values assigned to a specific keys cannot be done. Moreover, for the same reason the interface cannot exposes operations as replace. However, this work can be extended to multi-maps using the limitation we described.

Extending the work to arrays, queues, and stacks can also be done while limiting the interface to guarantee that linearizability verification can be done while bounding the number of inputs.
Chapter 11

Conclusion

Verifying the linearizability of composed concurrent operations is considered to be a very difficult problem. In this thesis we show that by identifying a class of programs that is frequently used in practice and focusing on that class, we can devise a reduction that allows us to automatically prove linearizability for many real-world composed operations in a simple and efficient manner. We describe three restricted classes of programs, enabling us to define a reduction and a verification procedure that considers only a bounded number of threads and map entries.

We also present an effective technique for testing composed concurrent operations that do not belong to these classes. Our technique leverages influence information about the underlying collections to guide the execution towards linearizability violations.

We implemented our techniques in a tool called COLT and showed its effectiveness by detecting 59 previously unknown linearizability violations and proved the linearizability of 31 composed operations in 57 popular open-source applications such as Apache Tomcat.
Bibliography


[65] Ohad Shacham, Nathan Bronson, Alex Aiken, Mooly Sagiv, Martin Vechev, and Eran Yahav. Practical verification of composed concurrent operations. Will be submitted to *OOPSLA’12*, 2012.


Appendix A

Chameleon: Adaptive Selection of Collections

This chapter is based on the paper “Chameleon: Adaptive Selection of Collections” that was presented at PLDI’09 [66].

A.1 Introduction

Programming languages such as Java, C#, Python and Ruby include a collection framework as part of the language runtime. Collection frameworks provide the programmer with abstract data types for handling groups of data (e.g., Lists, Sets, Maps), and hide the details of the underlying data-structure implementation.

Modern programs written in these languages rely heavily on collections, and choosing the appropriate collection implementation (and parameters) for every usage point in a program may be critical to its performance.

Real-world applications may be allocating collections in thousands of program locations, making any attempt to manually select and tune collection implementations into a time-consuming and often infeasible task. It is therefore not surprising that recent studies [60] have shown that in some production systems, the utilization of collections might be as low as 10%, that is, 90% of the space consumed by collections in the program is overhead.

Existing profilers ignore collection semantics and memory layout, and aggregate information based on types. Offline approaches using heap-snapshots (e.g., [59, 60]) lack information about access patterns, and cannot correlate heap information back to the relevant program site.

In this Chapter, we present the first practical tool that automatically selects the appropriate collection implementations for a given application. Our tool uses what we call semantic profiling together with a set of collection selection rules to make an informed choice. This approach is markedly different from
existing profiling tools where the user is forced to manually filter massive amounts of irrelevant data, typically offline, in order to make an educated guess.

Semantic Collections Profiling The semantic profiler consists of a tightly integrated collections-aware production virtual machine and a runtime library. During program execution, these two components collect a myriad of complementary context-specific collection-usage statistics such as continuous space utilization and access patterns for each object. This information is obtained online and transparently to the programmer, without any need for an offline analysis of a general (non-targeted) heap dump. The ability of CHAMELEON to map all statistics back to the particular allocation context in the program is extremely useful as it enables the developer to focus on collections with maximum benefit. We have also pre-equipped our tool with a set of collection selection rules which are evaluated on the dynamic statistics. The output of the tool is a set of suggestions on how to improve the collections allocated at a particular allocation context. We are not aware of any other tool that can automatically produce such effective information in a low-overhead manner.

Selection from Multiple Implementations In this work, we assume that we are given a set of interchangeable implementations for every collection type. The requirement is that the different implementations have the same logical behavior. For example, a Set may be implemented using an underlying array, or a linked-list, but all implementations have to maintain the functional behavior of a set (e.g., have no duplicates). We focus on optimizing the choice of collection implementation, and do not address the orthogonal problem of showing that two collection implementations are logically equivalent (as done, e.g., in [58]).

Our library provides a number of alternative implementations, and we allow the user to add her own implementations, and implementations obtained from other sources (e.g., [2, 10, 12, 16]).

A.1.1 Main Contributions

The main contributions of this Chapter can be summarized as follows:

- A semantic profiler which tracks useful collection usage patterns across space and time. The profiler aggregates and sorts data for each collection allocation-context.

- A collection-aware garbage collector which continuously gathers statistics for a collection ADT rather than individual objects. This is very useful as collection ADTs usually consists of several objects (that can be described by maps). The collector is parametric on the semantic ADT maps, and can be reused for any (including user-specific) collection implementation.

- A flexible rule engine that selects the appropriate collection implementation based on the profiling information. Our rule engine allows the programmer to write implementation selection rules
over the collected profile information using a simple, but expressive implementation selection language.

- A complete implementation of our tool over IBM’s J9 JVM.
- Evaluation of our tool on a small set of benchmarks where we show that following CHAMELEON recommendations can lead to significant improvement in program space requirements, as well as running time.

A.2 Overview

In this section we provide a high-level overview of CHAMELEON by demonstrating how it is applied to an example, and briefly discuss some of the tradeoffs of collection implementations.

Figure A.1 shows an overview of CHAMELEON. The tool works in two automated phases: (i) semantic collection profiling—gathering a wide range of collection statistics during a program run; (ii) automatic selection using a rule engine—using a set of selection rules evaluated over the collected statistics to make implementation selection decisions. The tool is parametric on the semantic maps used for profiling (Section A.3.2), and on the set of selection rules (Section A.3.3).

A.2.1 Motivating Example

tvla [55] is a flexible static analysis framework from Tel-Aviv University. The framework performs abstract interpretation with parametric abstractions, and computes a set of abstract states that over-approximate the set of all possible concrete program states. tvla is a memory-intensive application, and its ability to tackle key verification challenges such as concurrent algorithms (which have large state spaces) is mostly limited by memory consumption. The tvla framework makes extensive use of collections.

Our goal in this example is to optimize the collections usage in tvla. The first step towards that goal is to check the potential for collection optimizations in the application.

Figure A.2 shows the percentage of live-data that is consumed by collections in tvla running on a particular analysis problem—showing that Lindstrom’s binary-search-tree traversal preserves the shape of the underlying tree. This figure is the actual output as produced by the semantic profiler in CHAMELEON.

The figure shows three measures as percentage of the live data: (i) total live data consumed by collections (live); (ii) total size of the used parts of collections (used); (iii) lower-bound space consumption of the actual collection content (core).

The gap between live and core is the best-case potential space saving for collections (see more details on the nature of this space overhead in Section A.2.2). Of course, some collection implementations,
Appendix A. Chameleon: Adaptive Selection of Collections

Figure A.1: CHAMELEON overview

Figure A.2: Percentage of live data consumed by collections in tvla running on a benchmark. The “X” axis shows the GC cycle and the “Y” axis shows the percentage of the live data.
Figure A.3: Combined results for top 4 allocation contexts in tvla.
such as hash tables, introduce additional space in order to facilitate efficient operations, so comparison to core is non realistic. We provide the core measure as a lower-bound for the space requirement, and to see how changes to the volume of the actual stored data affect the space consumed by collections.

The gap between live and used corresponds to the total space allocated by the collection implementation that is not used to store application entries. In the figure, we see that collections constitute up to 70% of the live data, and the part used to store collection elements is only up to 40% of the live data.

At this stage, it seems that there is a realistic potential for space saving, but the question is — how do we realize this potential? In particular, how do we relate this information to the program? What can be done in order to avoid this space overhead? What points in the program should be modified?

Using several heap-snapshots taken during program execution may reveal the types that are responsible for most of the space consumption. However, a heap snapshot does not correlate the heap objects to the point in the program in which they are allocated. Therefore, finding the program points that need to be modified requires significant effort, even for programmers familiar with the code. Moreover, once the point of collection allocation is found, it is not clear how to choose an alternative collection implementation. In particular, choosing an alternative collection implementation with lower space overhead is not always desirable. Some structures, such as hash-tables, have inherent space overhead to facilitate more time-efficient operations. In order to pick an appropriate implementation, some information about the usage pattern of the collection in the particular application is required.

CHAMELEON is the first tool to integrate heap-information with information about the usage-pattern of collections. The semantic profiler in CHAMELEON produces a ranked list of allocation contexts in which there is a potential for space saving. For each such allocation context, the profiler in CHAMELEON provides comprehensive information such as the distribution of operations performed on collections allocated at the context, the distribution of collection sizes, etc. Figure A.3 shows an example for such a summary. It shows the top 4 allocation contexts in tvla, with their corresponding space saving potential. For example, for context 1, there is a space potential of roughly 10 percent of total live heap. Additionally, for each context, the tool provides the distribution of operations (represented as circles in the figure). For brevity, we don’t show the names of the operations. For contexts 1, 3 and 4, the operation distribution is entirely dominated by get operations, while for context 2 there is also a small portion of add and remove operations. In addition to profiling information for each context, CHAMELEON produces suggestions on which collection implementations to use. For this example, we get the following succinct messages (for brevity we only show suggestions for contexts 1 and 4):

- **1**: HashMap:tvla.util.HashMapFactory:31;tvla.core.base.BaseTVS:50
  - replace with ArrayMap

- **4**: ArrayList:BaseHashTVSSet:112; tvla.core.base.BaseHashTVSSet:60
A.2. Overview

To produce this report, CHAMELEON combines information on how the collections are used, with information on the potential saving in each context. The combined information is used by our rule-engine, to yield collection tuning decisions that are presented to the user. The final report usually includes the precisely tracked context, which in our case consists of the call-stack when allocation occurred (usually of depth 2 or 3). This is often required when the application uses factories for creation of collections (as is done sometimes in tvla).

Next, we apply the collection decisions CHAMELEON advocated in 5 top allocation sites in tvla, and re-run the application. The overall effect on the total space required by the application is dramatic. In particular, the minimal heap-size required to run the application has been reduced by 50%. The effect on the overall running time is also significant. The total time required to complete the verification by using the modified version based on the collection implementations advocated by CHAMELEON is more than 2.5 times faster than the original one (from 49 to 19 minutes). Further saving of time and space is possible by modifying additional program points.

In contrast to standard profiling tools which require heavy manual involvement, by using the semantically focused, context-specific suggestions provided by CHAMELEON, we were able to achieve dramatic performance improvements quickly and with little manual effort.

A.2.2 Tradeoffs in Collection Implementations

Selecting an appropriate collection implementation is more complicated than it seems at first sight.

Time It is possible to base the selection on asymptotic time complexity of collection operations. However, the asymptotic time complexity of collection operations is not a good measure of their behavior when the collections contain a small number of items. In the realm of small sizes, constants matter. Furthermore, in practice, the actual performance of a collection is affected by different aspects, such as the locality of the selected structure, the cost of computing a hash function, cost of resizing the structure etc.

Space Collections vary in how much space overhead is consumed for storing a specific amount of data. They typically have different fixed overhead per element in the collection. For example, every element stored in the LinkedList implementation has an Entry object associated with it, where the Entry object stores a reference to the actual element, and two references to the next and previous entries in the list.

At each allocation site in the program, we define the utilization of a data structure as the ratio between the size of the data that it represents and the total amount of memory that this instance currently

– set initial capacity
uses. Similar utilization metrics are used in the context of memory health measures [60]. As utilization varies during the execution, we consider both the utilization along points of program execution, and the overall average utilization of the collection.

There are several causes of low utilization: (i) the initial capacity of the collection is not suited to the average size of data stored in it; (ii) the collection is not compacted when elements are removed from it; (iii) high overhead per item in the collection.

For example, an ArrayList expands its capacity whenever it runs out of available space. The capacity grows by the function $newCapacity = (oldCapacity \times 3)/2 + 1$. Consider an ArrayList that has an initial capacity of 100 and contains 100 elements. Adding another element increases the size of the allocated array to 151 while only containing 101 elements.

**Space/Time Tradeoffs** It is key to note the tradeoff between time and utilization (space). We can improve utilization by taking more time to perform operations. For example, given an ArrayList implementation, we can resize the array on every operation exactly to the number of elements it contains. This would incur a significant time penalty, but would keep the utilization at close to 100% (accounting for the meta-data in the collection object header etc.). Conversely, if we don’t care about utilization, we can pre-allocate the array at the maximal number of elements, which would yield a very low utilization, but would avoid the need for resizing the array. Similarly, choosing an array over a linked-list would improve utilization, but would make update operations more costly.

**A.2.3 Possible Solutions for Low Utilization**

There are several seemingly reasonable solutions that can be used to tackle the poor utilization of data structures.

First, we can set the initial size of all allocated collections to one and then resize the collection size whenever an insertion or removal operation takes place. Second, we can use a hybrid collection mechanism. Initially the structure is implemented as an array. Then, whenever, the size of the collection increases beyond a certain bound, we can convert the array structure to the original implementation.

The advantage of both of these solutions is that they operate based only on local knowledge. That is, decisions for the collections implementation and size are determined within the specific collection object and are not based on any kind of global information such as allocation context.

Unfortunately, we were unable to reduce the memory footprint using these solutions with a reasonable time penalty.

Using small initial sizes does not reduce the memory footprint due to the fact that in Hash-based ADT, such as HashMap, each hash entry is represented by a new object containing three pointer fields. The first is a next pointer referencing the next entry. The second is a prev pointer referencing the previous entry. The third is a pointer to the data itself. The entry object alone on a 32-bit architecture
consumes 24 bytes (object header and three pointers). Therefore, even when starting with a small initial size, significant memory not related to actual data is consumed, in this case, due to the large entry size.

The second (hybrid) solution can be effective in reducing footprint; however, choosing the size when the conversion from an array based implementation should take place is very tricky without causing significant runtime degradation. In TVLA for example, making the conversion of ArrayMap to HashMap at size 16 provides a relatively low footprint with 8% performance degradation. However, increasing the conversion size to a larger number than 16 does not provide a smaller footprint and leads to performance degradation. Moreover, reducing the conversion size to 13 provides the same footprint as the original implementation does.

A.3 Automated Collection Selection

In this section, we discuss our solution for automatically selecting the appropriate collections for a given user program. First, in Section A.3.1, we define the problem. Then, we show how we address this problem with a combination of semantic collection profiling (Section A.3.2), and a rule engine (Section A.3.3).

A.3.1 Optimal Selection of Collection Implementations

Given a program that uses collections, our goal is to find an assignment of collection implementations that is optimal for that program. An optimal choice of collection implementations tries to balance two dimensions: minimizing the time required to perform operations while also minimizing the space required to represent application data.

The problem of optimal collection selection can be viewed as a search problem: for every point in a program allocating a collection, for each possible collection implementation, run the program, and compare the results in terms of space consumption and overall running time. However, this approach is not likely to scale for anything but the smallest programs. Furthermore, comparing results across executions is a daunting task in the presence of non-determinism and concurrency.

An alternative approach is to select collection implementations based on collection usage statistics extracted from the client program. Since there is no a priori bound on the number of collection objects in a program, and there is no a priori bound on the sequence of operations applied on a collection object, it is not practical to represent all operation sequences directly, and an abstraction of the usage patterns is required.

In principle, an abstraction of the collection usage pattern in a program can be obtained either statically or dynamically. However, static approaches to this problem typically abstract away the operation counts, which are a crucial component of usage patterns, and are not likely to scale to realistic applications. Seeking a scalable approach, we focus our attention on selection based on dynamic information.
A dynamic approach would have to track, in a scalable manner, enough information on the usage of collections to enable the choice of appropriate implementations.

A.3.2 Semantic Collections Profiling

In this section we describe the information collected by the semantic collections profiler of CHAMELEON and how this information is used in order to make collection selection decisions.

Allocation Context

Our work is based on the hypothesis that the usage patterns of collection objects allocated at the same allocation context are similar. More precisely, we define the allocation context of an object $o$ to be the allocation site in which $o$ was allocated, and the call stack at the point when the allocation occurred.

For allocation contexts in which we observe similarity between usage patterns to hold within reasonable statistical confidence, we determine the type of collections that should be allocated in the context based on the average usage pattern.

Definition A.3.1 (Stability) We define the stability of a metric in a partial allocation context $c$ as the standard deviation of that metric in the usage profile of collections allocated in $c$.

Examples of metrics are: the number of times a certain operation is performed on a collection instance and the maximal size of the collection during its lifetime. For every metric we define a threshold that determines the limit under which the metric is considered stable.

Practically, the full allocation context is rarely needed, and maintaining it is often too expensive. Therefore, we use a partial allocation context, containing only a call stack of depth two or three. The observation that (a small) allocation context is crucial to object behavior is in line with the recent study of Jones et. al [52]. In that study, the authors observed that objects allocated in the same context tend to behave similarly and a garbage collection strategy should be made aware of this correlation. In our work, we exploit this insight for optimizing collection usage.

Collection Statistics

CHAMELEON records a wide range of statistics indicating how collections in the program are used. Much of the information recorded by the tool is per allocation context, and is an aggregation of the information collected for objects allocated at that context.

Dynamically Tracked Data The tracked data is shown in Table A.1. The collected information is a combination of information about the heap (e.g., the maximal heap size occupied by collection objects during execution), and information about the usage pattern (e.g., the total number of times contains was invoked on collections in the context).
A.3. **Automated Collection Selection**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overall live data</td>
<td>Total/Max size of all reachable objects</td>
</tr>
<tr>
<td>Collection live data</td>
<td>Total/Max size of collection objects</td>
</tr>
<tr>
<td>Collection used data</td>
<td>Total/Max used part of collection objects</td>
</tr>
<tr>
<td>Collection core data</td>
<td>Total/Max core part of collection objects</td>
</tr>
<tr>
<td>Collection object number</td>
<td>Total/Max # of live collection objects</td>
</tr>
<tr>
<td>Number of operations</td>
<td>Total number of operations performed</td>
</tr>
<tr>
<td>Avg/Var operation count</td>
<td>Average # of times an operation was performed, and its standard deviation.</td>
</tr>
<tr>
<td>Avg/Var of maximal Size</td>
<td>Average maximal size of collections allocated, and its standard deviation.</td>
</tr>
</tbody>
</table>

Table A.1: Heap and trace statistics for each execution. Information is aggregated per *allocation context*.

**Heap Information** The heap information provides a comprehensive summary of the space behavior of collections during program execution. This information is collected on every garbage collection (GC) cycle. The GC computes the total and maximal live data of the program where the total live data is the sum of all live data’s accumulated over all of the GC cycles and the maximal live data is the largest live data seen in any GC cycle. The GC has been augmented with *semantic maps* and routines to compute various *context-specific* collection information (discussed further in Section A.4). First, it computes the total and maximal space consumed by reachable collection objects across all GC cycles. Second, it computes the total and maximal space actually used by these collection objects (collection used data). This is important for knowing how much of the collection object is really utilized. Thirdly, it computes the total and maximal collection core size, which would be the ideal space that would be required to store the core elements of the collection object in an array. This statistic is useful to provide a lower bound on the space requirement for the content of the collection (hence indicating the limit of any optimization). Finally, the total and maximum number of live collection objects are computed.

**Trace Information** As mentioned earlier, recording the full sequence of operations applied to a collection object has a prohibitive cost. Instead, our trace information records the distribution of operations, as well as the maximal size observed for collections at the given context. The average operation counts provide a count of all possible collection operations. For brevity, we do not list all of them here. For some operations, those that involve interactions between collections, we introduce additional counters that count both sides of the interaction. For example, when adding the contents of one collection into another using the `c1.addAll(c2)` operation, we record the fact that `addAll` was invoked on `c1`, but also the fact that `c2` was used as an argument for `addAll`. Similarly, we record when a collection was used...
in a copy constructor. These counters are particularly important for identifying temporary collection objects that are never operated upon directly, other than copying their content.

**Using Profiling Information** The statistics from the tool can be used in several ways. For example, as the program runs, the user can request the tool to output the current top allocation contexts, sorted by maximum benefit. In the case where the user wants to make manual changes, she can focus on the most beneficial contexts instantly. Alternatively, she can use the recommendations automatically computed by the tool, which are based on a set of selection rules. To allow flexibility in querying the information collected by the tool, and select appropriate implementations based on it, we let the user write rules in a simple language. We describe these next.

### A.3.3 Rule Engine

#### A Simple Rule Language

We allow the user to write replacement rules, using the language of Figure A.4. The language is pretty standard, and in the figure we abbreviate rules that contain standard combinations of operations, such as boolean combinations for `cond` and arithmetic operators for `expr`. The language allows to write conditional expressions under which a replacement takes place. The conditional expressions use the metrics of Table A.1 as the basic vocabulary. The language allows to write conditional expressions comparing the ratios between operation counts (e.g., the ratio of `contains` operations \#contains/\#allOps), the operation count itself (e.g., \#remove == 0) etc. It also allows the user to check the variance of counts (e.g, \@add). The language also allows the user to query the live-data

\[
\text{rule} \quad := \quad \text{srcType} \quad \text{cond} \quad \text{implType} | \\
\text{srcType} \quad \text{cond} \quad \text{implType}(\text{capacity}) \\
\text{srcType} \quad := \quad \text{Collection} \mid \text{ArrayList} \mid \text{LinkedList} \mid \ldots \\
\text{implType} \quad := \quad \text{ArrayList} \mid \text{ArrayMap} \mid \text{HashSet} \mid \ldots \\
\text{cond} \quad := \quad \text{comparison} \mid \text{cond} \wedge \text{cond} \mid \ldots \\
\text{comparison} \quad := \quad \text{expr} \leq \text{constant} \mid \text{expr} == \text{constant} \mid \ldots \\
\text{expr} \quad := \quad \text{opCount} \mid \text{opVar} \mid \text{data} \mid \text{expr} + \text{expr} \mid \ldots \\
\text{opCount} \quad := \quad \#\text{add} \mid \#\text{get}(\text{int}) \mid \#\text{get}(\text{Object}) \mid \ldots \\
\text{opVar} \quad := \quad \@\text{add} \mid \@\text{remove} \mid \ldots \\
\text{data} \quad := \quad \text{tracedata} \mid \text{heapdata} \\
\text{tracedata} \quad := \quad \text{size} \mid \text{maxSize} \mid \text{initialCapacity} \\
\text{heapdata} \quad := \quad \text{maxLive} \mid \text{totLive} \mid \text{maxUsed} \mid \text{totUsed} \ldots \\
\text{capacity} \quad := \quad \text{INT} \mid \text{maxSize}
\]

Figure A.4: Simple language for implementation selection rules.
occupied by collections at the context, and the used-data occupied by collections at the context. These are typically used to figure out whether the potential saving in this allocation context \((\text{totLive} - \text{totUsed})\) is greater than some threshold.

**Chameleon Collection Selection**

Table A.2 shows several examples of selection rules that are built into CHAMELEON. The constants used in the rules are not shown, as they may be tuned per specific environment. For example, the rule

\[
\text{ArrayList: \#contains} > X \land \text{maxSize} > Y \rightarrow \text{LinkedHashSet}
\]

specifies that if the type allocated at this context is an ArrayList, and the average number of contains operations performed on collections in this context is greater than some threshold \(X\), and the average maximal size of the collection is greater than some threshold \(Y\), then the selected type should be a LinkedHashSet. This rule corresponds to the fact that performing a large number of contains operations on large-sized collections is better handled when the collection is a LinkedHashSet. Of course, the rule can be refined to take other operations into account. The user can write various expressions in this language that dictate which implementation to select. For example, when the potential space saving is high, one may want to apply a different collection selection even if it results in a potential slowdown. For instance, the space benefit of the rule selecting an ArraySet instead of HashSet may outweigh the time slowdown when the potential space saving \((\text{totLive} - \text{totUsed})\) is greater than some threshold. Conversely, we can avoid any space-optimizing replacement when the potential space savings seems negligible.

Section A.5 shows that using CHAMELEON recommendations based on rules such as those of Table A.2 can yield significant performance improvements.

**Stability**

If stability is not specified explicitly in the rule, it is assumed that any metric has to have its standard deviation less than a fixed constant (in the current implementation, size values are required to be tight, while operation counts are not restricted). Generally, different metrics may require different measures of variance based on their expected distribution. For example, while the operation counters usually distribute normally, maximal collection sizes are often biased around a single value (e.g., 1), with a long tail. Our current implementation uses standard-deviation as the stability measure, but in the future we plan to evaluate the suitability of other measures of variance for different metrics.

**Towards Complete Automation**

The current operation mode in which CHAMELEON is used is to evaluate all selection rules at the end of program execution, when complete information has been obtained for all collections allocated at a
<table>
<thead>
<tr>
<th>Type</th>
<th>Condition</th>
<th>Category: Message</th>
<th>Suggested Fix</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArrayList</td>
<td>#contains &gt; X \land \text{maxSize} &gt; Y</td>
<td>Time: Inefficient use of an ArrayList: large volume of contains operations on a large sized list</td>
<td>LinkedHashSet</td>
</tr>
<tr>
<td>LinkedList</td>
<td>#get(int) &gt; X</td>
<td>Time: Inefficient use of a LinkedList: large volume of random accesses using get(i)</td>
<td>ArrayList</td>
</tr>
<tr>
<td>LinkedList</td>
<td>((#\text{add(int, Object)}+#\text{addAll(int, Collection)})+#\text{remove(int)}+#\text{removeFirst}) &lt; X</td>
<td>Space: LinkedList overhead not justified when adding/removing elements from the middle/head of the list is hardly performed</td>
<td>ArrayList</td>
</tr>
<tr>
<td>Collection</td>
<td>\text{maxSize} == 0</td>
<td>Space: Redundant collection allocation</td>
<td>LazyArrayList</td>
</tr>
<tr>
<td>HashSet</td>
<td>\text{maxSize} &lt; X</td>
<td>Space: ArraySet more efficient than an HashSet. Time: operations on a small array might be faster than on an HashSet</td>
<td>ArraySet</td>
</tr>
<tr>
<td>Collection</td>
<td>#\text{allOps} = 0</td>
<td>Space/Time: redundant collection</td>
<td>avoid allocation</td>
</tr>
<tr>
<td>Collection</td>
<td>#\text{allOps} == #\text{copied}</td>
<td>Space/Time: redundant copying of collections</td>
<td>eliminate temporaries</td>
</tr>
<tr>
<td>Collection</td>
<td>\text{maxSize} &gt; \text{initialCapacity}</td>
<td>Space/Time: incremental resizing</td>
<td>set initial capacity</td>
</tr>
<tr>
<td>Iterator</td>
<td>\text{collection.size} == 0</td>
<td>Space: Redundant iterator</td>
<td>remove</td>
</tr>
</tbody>
</table>

Table A.2: Example of built-in rules.
given context. The suggested implementations can then be applied by the programmer (or by the tool) and the program can be executed again (with or without profiling).

An interesting challenge is whether the act of replacement can be applied while the program is running. Such an online solution may be beneficial for several reason:

* Lack of Stability: It is possible that collection objects from a given allocation context exhibit wide variation in behavior, for example due to different program inputs, phasing or non-determinism. Hence, detecting these cases and allocating the appropriate collection object may be more advantageous than sticking to a single implementation for all cases.

* Optimization of Underlying Framework: Most real-world software makes use of framework code. The framework code itself may make extensive use of collection. Online selection can specialize the collection-usage in underlying frameworks, that is typically outside the scope of programmer’s manual modifications. In general, this follows a theme of specializing the library for a particular client, as part of the client’s execution in the runtime environment.

* No Programmer Effort: Manual replacement may require nontrivial code-modifications to deal with factories and deep allocation contexts. Dynamic selection is performed as part of the runtime environment and requires no manual modifications to the source code.

Dealing with completely automatic replacement is challenging because decisions may have to be based on partial information: at what point of the execution can we decide to select one collection implementation over another? For example, if the tool replaces the type allocated at a given context from a HashMap to an ArrayMap on the premise that objects allocated at that context have small maximal sizes, even a single collection with large size may considerably degrade program performance. Additionally, such a tool must run with sufficiently low overhead to be enabled during production deployment. Therefore, it is crucial to reduce overhead costs and in particular, it is vital to be able to obtain allocation context cheaply.

Towards the vision of fully automatic management of collections at runtime, we performed preliminary experiments where we used CHAMELEON in a mode where all replacements are done completely automatically at runtime, without any user involvement. We describe our results in Section A.5.

### A.4 Implementation

In this section we present the design and implementation of our tool. The tool consist of two complementary components: the library and the virtual machine, which are integrated in a manner that is transparent to the end user. The design of these components is such that they can be used separately by switching on and off each component on demand. However, for maximal benefit we typically use them together. By selectively instrumenting the library, we are able to record various useful statistics such as frequency of operations and distributions of operations for a given collection size. While this infor-
mation is useful, it still does not provide us with a relative view of how collections behave with respect to the whole system. However, such global information can be extracted from the virtual machine and in particular from the garbage collector (GC). By instrumenting the GC to gather semantic information about collections, we are able to answer questions such as the total live data occupied by collections at a specific point in time. Such information, while cheap to obtain from the GC, is very costly to obtain at the library level. Next, we describe each component separately as well as how they interact with each other.

A.4.1 Design Choices

One of the core principles that we followed in our approach is to avoid as much as possible any changes to the original program. A key place where a dilemma between portability and slightly better efficiency occurs is during allocation of a collection object. For example, if the user program requests an allocation of a HashMap object and the system determines that for this context, it is best to implement that HashMap object with an ArrayMap, we are faced with two possible implementation choices. First, we can make ArrayMap a subtype of HashMap and then return ArrayMap. The problem is, that ArrayMap would then inherit all fields from HashMap. Further, any program expressions that depend on the precise type being HashMap may work incorrectly. Another solution is to have ArrayMap and HashMap as sibling types, but to return an object of type ArrayMap. In that case, we need to make sure that all type declarations in the program match ArrayMap (that were HashMap before) and that all semantic behavior depending on a specific type is preserved. This is the approach taken by Sutter et. al for details [69]. However, statically re-writing the type declarations of the program is intrusive, challenging, can lead to subtle errors due to language features such as dynamic typing, and is generally difficult to scale on large programs. Our solution in that case has been true to Lampson’s statement that all problems in computer science can be solved by another level of indirection. Hence, we add another level of indirection between the program and the collection implementation. That is, each allocation of a collection object requires a wrapper. In our example, whenever HashMap is allocated, it will be a small wrapper object. Then, internally, the wrapper object can point to any implementation of HashMap. We believe that a small delta in inefficiency is worth the software reliability gains. Further, we believe that with VM support we can reduce this inefficiency further (e.g. via object inlining).

A.4.2 Library Architecture

Figure A.5 shows the architecture of the CHAMELEON libraries.

Our wrappers delegate collection operations to the underlying selected collection implementation (similar to the Forwarding types in Google’s Collections [10]). The only information kept in the wrapper object is a reference to the particular implementation. In our solution, the actual backing implementation
can be determined statically by the programmer (by explicitly providing the constructor with an appropriate constant), left as the default choice that the programmer indicated, or determined dynamically by the system.

As the wrapper allocates the backing implementation object, it also obtains the call stack (context) for this allocation site and constructs a VMContextKey object that records it (via the locationId fields inside it). This object is then used to look up the corresponding ContextInfo object, which records aggregate information for this context. In order to collect information on the collection usage pattern for this context, the backing implementation may allocate an ObjectContextInfo. This object is used to store the various operation counters, collection maximal size, etc. When the collection implementation object dies, the contents of its object information object are aggregated into the corresponding ContextInfo object (via finalizers as described later).

**Obtaining Allocation Context**  CHAMELEON tracks information at the level of an allocation context. This requires that an allocation context be obtained whenever a collection object is allocated. We have implemented two methods for obtaining the allocation context: (i) a language-level method based on walking the stack frames of a Throwable object; (ii) a method using JVMTI.

The JVMTI-based implementation is significantly faster than the Throwable-based implementation which requires the expensive allocation of a Throwable object, and the manipulation of method signatures as strings (our native implementation works directly with unique identifiers, without constructing intermediate objects to represent the sequence of methods in the context).
We are currently working on a third implementation using a modification of the JVM to obtain bounded context information in a lightweight manner. In addition, there are many approaches that target the problem of obtaining context \([22, 25, 26, 76]\), we intend to explore some of these in future work.

**Sampling of Allocation Context:** To further mitigate the cost of obtaining the allocation context, CHAMELEON can employ sampling of the allocation contexts. Moreover, when the potential space saving for a certain type is observed to be low, CHAMELEON can completely turn off tracking of allocation context for that type. (Technically, sampling is controlled at the level of a specific constructor.)

**Available Implementations** Our goal in this work is to study the problem of collection implementation selection, and not to improve the default collection implementations. There are many alternative open-source collection implementations \([2, 10, 12, 16]\), varying in terms of robustness, compatibility, and performance. In principle, these implementations can be swapped-in as additional possible implementations for the collection interfaces, with appropriate selection rules on when they should be used.

In our experiments, however, we used our own alternative implementations for collections, for example:

- List:
  - ArrayList - resizable array implementation.
  - LinkedList - a doubly-linked list implementation.
  - LazyArrayList - allocate internal array on first update.
  - IntArray - array of ints. (Similar for other primitives)

- Set (and similarly for Map):
  - HashSet (default) - backed up by a HashMap
  - LazySet - allocates internal array on first update
  - ArraySet - backed up by an array
  - SizeAdaptingSet - dynamically switch underlying implementation from array to HashMap based on size.

Further performance improvements can be achieved by swapping additional implementations under the appropriate conditions. However, some of these conditions are subtle. For example, selecting an open-addressing implementation of a HashMap (e.g., from the Trove collections) requires some guarantees on the quality of the hash function being used to avoid disastrous performance implications. This is hard to determine in Java, where the programmer can (and does) provide her own `hashCode()` implementation.
**Context Information**  As mentioned previously, the `ObjectContextInfo` object collects the usage pattern for collection instances. This information is aggregated into the `ContextInfo` maintained for the corresponding allocation context. As we will see later, with VM support, the context information can also contain information about the heap usage of collections allocated at the given allocation context. As we mentioned already, our design allows us to benefit from VM support, but can also be used when such VM support is absent.

**A.4.3 VM Support**

While gathering information at the library level is useful, it is often very difficult to obtain any kind of global view of how collections fit into the whole behavior of the program. For example, even though a particular context allocates memory at a high rate, it is still not clear whether there is much benefit globally in tracking collection usage, for it may be the case that it is a small percent of total memory. Also, it may often be useful to monitor the application with very low overhead, without tracking any library usage, in order to determine whether there is any potential whatsoever in changing the implementation of collections.

One place where much of this global information can be accessed is during the GC cycle. By examining the program heap during a GC cycle, we can calculate various collection parameters such as distribution of live data and collection utilization. Moreover, with careful techniques, this valuable information can be obtained with virtually no additional cost to the program execution time, and as part of normal operation of the collector. To that end, we extended the GC to gather valuable *semantic* information pertaining to collections. At the end of each cycle, the collector aggregates this information in the `ContextInfo` object (which also contains trace-based information). The library can then inspect the combination of trace and heap information at the same time.

**Context-Sensitive Collection Data**

Note that simply examining the heap is often not enough, especially in large applications with thousands of program sites allocating collections. In particular, we would like to focus on specific allocation sites in the program which have the highest potential for gain. To that end, if the library maintains context information, the collector will automatically take advantage of this and record various context-specific information into the `ContextInfo` object.

**Collector Modifications**

In our implementation, we used the base parallel mark and sweep garbage collector, which works in the standard way. First, the roots of the program are marked (thread stacks, finalizer buffers, static class members, etc). Then, several parallel collector threads perform the tracing phase and compute transitive
 closure from these roots, marking all objects in that transitive closure. Finally, during the sweeping phase, all objects which are not marked are freed. In our system, the number of parallel threads is the same as the number of cores available in hardware. We note that our choice of this specific collector can possibly lead to different results than if we had used for example a generational collector. However, the improvements in collection usage are orthogonal to the specific GC.

We have instrumented the base collector to compute various semantic metrics during its marking phase. The set of metrics computed by collector is shown in Table A.3. From these metrics, we can compute aggregate per-context metrics over all GC cycles as described in Section A.3.2 and shown in Table A.1.

### Semantic ADT Maps

Typically, a collection object may contain several internal objects that implement the required functionality. For example, an ArrayList object may contain an internal array of type java.lang.Object[] to store the required data. This means that if we blindly iterate over the heap, we will not be able to differentiate object arrays that are logically part of ArrayList and those object arrays that have nothing to do with collections (e.g. allocated outside of ArrayList methods). This lack of semantic correlation between objects is a common limitation of standard profilers. Therefore, to efficiently obtain accurate statistics (such as size) about collections, we use what we call semantic maps. In brief, every collection type is augmented with a semantic map which describes the offsets that the collector use to find information such as the size of the object (which may involve looking up the size of the underlying array), the actual allocated size and its underlying allocation context pointer. Semantic maps are pre-computed for all collection types on VM startup. Using semantic maps allows us to obtain accurate information by avoiding expensive class and field name lookups during collection operation. Further, because the whole process is parametric on the semantic maps, we can run the system on any collection implementation (including custom implementations).

### Operation

Every time the collector visits a non-marked object, it checks whether it is an object of interest (a collection object). In that case, it consults the semantic map of its type and quickly gathers

---

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Live Data</td>
<td>The size of all reachable objects</td>
</tr>
<tr>
<td>Collection Live Data</td>
<td>Total occupied size of collection objects</td>
</tr>
<tr>
<td>Collection Used Data</td>
<td>Total used size of collection objects</td>
</tr>
<tr>
<td>Collection Core Data</td>
<td>Total core size of collection objects</td>
</tr>
<tr>
<td>Collection Object Number</td>
<td>Total number of live collection objects</td>
</tr>
<tr>
<td>Type Distribution</td>
<td>Live size breakdown for each type</td>
</tr>
</tbody>
</table>

Table A.3: Statistics gathered on every garbage collection cycle for each allocation context
the necessary statistics such as the live data occupied by the object (and its internal objects), the used data and the core data (the ideal space if we had only used a pointer array to represent the application data). Further, if the object tracks context information, using the semantic map, the collector finds the ContextInfo object and records the necessary information for that allocation context (as described in Table A.3).

A.4.4 Discussion

By augmenting the GC with semantic ADT maps, we were able to automatically and continuously compute various useful context-sensitive utilization metrics specific to the semantics of collections. Moreover, because the statistics are gathered during normal collection operation, no additional performance overhead is incurred.

The information obtained from the collector can be used in various ways. In our case, we propagate the information back to the ContextInfo object in the library in order to allow the tool to make a more informed decision by combining this with the library trace-based information. In addition, we also record the results for each cycle separately (it is up to the user to specify what they want to sort the results by as well as how many contexts to show) for further analysis. This information can be readily used by the programmer to quickly focus on contexts which have the most potential for further improvement.

Finalizer Usage In our early versions of this tool, we extended all collection implementation types with finalizer methods. However, we found that finalizers noticeably slowed the system down. One of the main reasons for this is that finalizer objects live for an additional collector cycle and hence all objects transitively reachable from the finalizable object will also live for an additional cycle (even if they are never referred by the finalize() method). We still rely on selective usage of finalizers and we use them only for ObjectContextInfo objects. These objects are usually very small (few words) and do not have other objects in their transitive closure. Moreover, in the online version, ObjectContextInfo objects are not always allocated, further mitigating any costs associated with finalizers. Note that for our purposes, we can also easily compute these statistics in the sweeping phase of the garbage collection cycle (and not rely on finalizers).

A.5 Experimental Results

A.5.1 Benchmarks

Because our tool runs on top of a production virtual machine and requires no changes to the application program, we were able to quickly run CHAMELEON on various applications. In our results, we focus on
space-critical applications such as soot [72], tvla [55] and findbugs [50]. We also ran CHAMELEON on all of the Dacapo benchmarks [24]. Most of the Dacapo benchmarks do not make intensive use of collections, and hence our tool showed little potential saving for those. However, it did show that there is potential on the benchmarks bloat, fop, and pmd. Hence, we concentrated our efforts on the results for these benchmarks and we present those later in this section. The inputs we used for our benchmarks are an internal Dacapo version for soot, tvla source code for findbugs, the large inputs for Dacapo benchmarks, and an analysis problem—showing that Lindstrom’s binary-search-tree traversal preserves the shape of the underlying tree for tvla. Also, in our experiments we did not track the potential in benchmarks such as hsqldb which use their own collection classes. However, with very little manual effort in the library, we can also profile such applications. The collection-aware GC can profile them already as it is parametric in the semantic maps that describe the custom collection classes.

A.5.2 Methodology

For each benchmark, we took the following steps towards optimizing collection usage:

1. Run CHAMELEON on the application. Based on the results, evaluate whether there is any saving potential. If there is no potential, move on to the next application, otherwise, proceed to the next step.
2. For benchmarks with potential, CHAMELEON reports the allocation contexts in sorted order with the appropriate suggestions.
3. Modify the top allocation contexts using the tool suggestions. This is a replacement step and hence can be easily automated.
4. Repeat steps 1-3 on the modified version.
5. Compare the gains for the top allocation contexts in the before and after versions.
6. Evaluate the effect of collection optimizations in terms of the minimal-heap size required to run the program, and the execution time when running with the original minimal-heap size.

A.5.3 Results

Figure A.6 shows the improvement of minimal space required to run the benchmark after applying fixes suggested by CHAMELEON. Figure A.7 shows the improvement of running times of the benchmarks after applying fixes suggested by CHAMELEON. The running times were obtained by running each benchmark with its corresponding original minimal-heap size requirement. Our experiments were obtained on an Intel Xeon 3.8Ghz dual hyper threaded CPUs, 5GB RAM platform running a 64 bit Linux. Next, we discuss each application we considered in more details.
A.5. EXPERIMENTAL RESULTS

Figure A.6: Minimal heap size required to run the benchmark after applying fixes suggested by CHAMELEON, shown as percentage of the original minimal heap size.

Figure A.7: Running times of the benchmarks after applying fixes suggested by CHAMELEON, shown as percentage of the original running time. Running times were obtained by running each benchmark with its corresponding original minimal-heap size.
Figure A.8: Percentage of collections in original version of bloat
bloat  The potential for bloat is shown in Figure A.8. The x-axis is the number of the GC cycle, while the y-axis is the percentage of the total live data computed at the end of the GC. This output is obtained directly from the collection-aware GC. The figure shows that bloat’s footprint is dominated by a spike of collections (at GC#656 in the figure), where the true required space for the collections is significantly lower.

The top allocation context reported by CHAMELEON for bloat corresponds to this spike of collections, and had a potential that dominated the potential of all other contexts. Furthermore, CHAMELEON reported that most of the LinkedLists allocated at that context remained empty and were never used. Around 25% of the heap at that point of execution was consumed by LinkedList$Entry objects that are allocated as the head of an empty linked list. More than 20% of space can be saved by making the lists into LazyArrayLists, but a simple manual modification in the code can make the allocation itself lazy, which reduces the minimal-heap size required to run the program by 56%.

FOP  In FOP (v0.95), based on the tool recommendations, some HashMaps were replaced with ArrayMaps and initial sizes of other collections were tuned. There was also one context that allocated collections that were never used (in InlineStackingLayoutManager). The result is a 7.69% reduction in the minimal-heap size required to run the program.

Findbugs  Based on CHAMELEON suggestions, we replaced some HashMaps by ArrayMaps, HashSets by ArraySets, and the initial sizes of other collections were tuned. We also performed lazy allocation for HashMaps in contexts where large percentage of the collections remain empty. The overall result is a reduction of 13.79% in the minimal-heap size required to run the program.

PMD  PMD was already manually optimized to a correct collection usage. EMPTY.LIST was assigned to List pointers when needed and the initial size of many ArrayLists was manually set. CHAMELEON discovered many empty and small sized ArrayLists that were mistakenly initialized to a high number. We manually performed lazy allocation for these ArrayLists which reduced more than 20 million ArrayList allocations. In addition, we set the tuned size of lists and replaced ArrayList allocation by SingletonList. And also replaced some HashSets by SizeAdaptingSet (similarly for maps). Unfortunately, all these changes did not reduce the minimal heap size required to run PMD. There are two main reasons for this. The first is that most of the reduced collections are short lived. The second is that most of the long-lived collection data in PMD is large and stable HashSets as well as large ArrayLists. However, even though our modifications did not reduce the minimal heap size, the number of GCs reduced by 16% which led to a runtime improvement of 8.33%.

Soot  Soot’s heap consists of many small objects that are long-lived. Its intermediate representation of program entities makes intensive use of Collection classes. For the most part, Soot uses ArrayLists
for flexibility. However, the initial capacity of the lists is rarely provided, and the overall utilization of the lists is rather low (overall, around 25%). For cases in which lists are known to be singletons, SOOT sometimes uses a designated type `SingletonList` to reduce space overhead.

The collection choices we applied in SOOT were simple. Using our tool, we first observed that in the few top contexts in which `ArrayList` was used to store singletons (by construction), the constructed collections are never modified, and replaced them with immutable `SingletonList` (e.g., in `JIfStmt`). We note that the SOOT team has made a similar selection for other commonly used types. The second suggestion CHAMELEON pointed out is the large potential saving for `ArrayList`s created in `useBoxes` methods. The idiom there is one of aggregation of used values up a tree. Every node creates an `ArrayList` of its uses, and aggregates uses from its children. The result is the creation of many `ArrayList`s that are being rolled into other `ArrayList` using `addAll`. Avoiding all temporaries requires a major rewrite of the code, but even without rewriting the code, we selected proper initial sizes for these lists. The overall result for SOOT was a saving of 6% in space, and 11% improvement in the running time.

**TVLA**
Most of the heap in TVLA is dedicated to storing the abstract program states that arise during abstract interpretation. The abstract program states use collections to store the state information. Most of the collection data is stored in `HashMaps` from seven contexts. CHAMELEON points this collections as ones that can be replaced by `ArrayMaps`. Replacing these collections provides a minimal-heap reduction of 53.95%. CHAMELEON also pointed an initial size setting for several contexts and `LinkedList` that can be replaced by an `ArrayList`.

**A.5.4 Discussion**

**Experience with Fully Automatic Replacement**
Our tool can run in fully-automatic mode in which replacement of collections is performed during program execution. Due to the high cost of obtaining allocation contexts, we expected the tool to incur a high time overhead, and only evaluated its effectiveness in terms of space reduction. We ran the tool in the fully automatic mode for all of our benchmarks to evaluate the quality of its replacement decisions. Much to our surprise, for most benchmarks, the overall slowdown was noticeable, but not prohibitive.

For TVLA, the space saving achieved was identical to the one we got with the manual modification. However, the impact on running time was significant, due to the cost of obtaining allocation contexts. Overall, TVLA suffered a slowdown of 35%. For space-critical applications such as TVLA this may be an acceptable tradeoff in practice. The only benchmark for which the slowdown was prohibitive (6x slowdown) was PMD, which performs massive rapid allocation of short-lived collections, which amplified the cost of obtaining allocation contexts.

Our experiments indicate that the performance bottleneck standing in the way to fully automatic
replacement is the task of obtaining an allocation context. We believe that with better VM support for this functionality, fully online replacement is within reach.

**Iterators** In many of our benchmarks, we have observed the (somewhat expected) massive creation of iterator objects. Quite often, the iterators were created over empty collections. For some of the collection interfaces (e.g., Set), the creation of a new iterator object can be avoided in this case in favor of returning a fixed static empty iterator. However, some collection interfaces allow addition of new items through an iterator, and therefore require that a new iterator object will be created even when the collection is empty.

**Specialized Partial Interfaces** The Java collection interfaces are rather rich, and pose significant restrictions on the underlying implementations. More efficient implementations could be introduced if collection interfaces are minimized, or at least separated. For example, the *List* interface currently supports a list iterator that can traverse the list both forward and backward. For practical purposes, such interface precludes an underlying implementation of using a singly-linked list. While we can leverage static analysis to identify the usage of the interface and select the library accordingly, it seems more desirable to modify the library interfaces to permit additional implementations.

### A.6 Related Work

Recent work by Jones and Ryder [52] suggests that allocation context is indicative of object behavior and argues that GC should take advantage of this (rather than relying on fixed heuristics). We use a similar observation to gather semantic-oriented object metrics and perform corrective actions accordingly.

The challenge of freeing the user from managing and choosing the right data structure for their application is not a new one. For example, in the context of the high-level language SETL, Dewar et. al [34] suggest the usage of a special sublanguage to declaratively specify the type of a data structure that a set or a map of a SETL program should use. A compiler then takes as input the SETL program and the data structure specification in the sublanguage and outputs an efficient implementation. More work on this subject by Schonberg et. al [63] focuses on eliminating the need for manually specifying the structures in a sub-language. It proposes an analysis that takes as input a pure SETL program and automatically infers suitable data structures for it. A similar line of work based on static analysis is presented by Low [56]. In contrast, our work is done in the context of a lower-level language (Java) where the operations on the data-structure (collection) are explicit. Further, our work is centered around dynamic (rather than static) analysis. The mere size of current programs combined with modern language features make it challenging to statically optimize collection usage.

Active Harmony [29] is a system for automatic tuning of programs. The system contains a layer
for automatic tuning of parameters as well as a library specification layer that helps the application select the right library to execute. Active Harmony requires each library to provide a performance-evaluation function, and a cost-estimate function. The functions are used by the tuning algorithm to evaluate the performance of the library, or estimate its possible behavior. In contrast, our work targets general purpose object oriented programs where the program is executed in a runtime environment, and optimizes this environment to collect valuable information. In addition, we use allocation contexts to share historical information between objects as well as gain some metric of stability of collection behavior. Moreover, our work combines the GC and VM information per context to decide which contexts are worthy to optimize.

More recent work dealing with the challenges of using custom collections in Java is that of Sutter et al [69]. They apply static analysis to determine when a replacement of an existing collection type with a custom type is possible without violation of type constraints. Further, they profile several applications to determine where a replacement may be possible. Subject to the type constraints, their analysis automatically replaces existing types with custom types. Their replacement is based on allocation site (rather than context as is in our case). Their profiling information does not include heap information (as we do via VM support). We see our work as complementary. We can provide a more detailed profiling information and then use a static analysis to determine when it is safe to replace one type with another. However, because our system supports wrappers, we are able to always make a replacement as the type safety cannot be violated.

Recently, there has been work on application-specific selection of GCs, see Soman and Krintz [68] for details. The challenges they face are broadly similar to ours: when should one switch from one GC to another and what application characteristics should switching take into account. For example, the authors describe a scenario of switching to a GC that is tuned for resource-constrained environments when the memory becomes scarce. GC switching occurs at pre-defined points when all application threads are stopped. Switching GCs is complex as it may involve on stack replacement to adjust methods to the specific GC (e.g. use write barriers for generational GC). In our case, switching is localized as it occurs when a collection object is allocated which does not require us to stop application threads. An interesting item of future work is looking into GC strategies that have semantic knowledge of collection objects. For example, the GC may allocate ArrayList and its internal object array together for locality purposes.

Recently, there has been work on semantically modifying the GC to detect various correctness properties, see [17, 20]. In our case, we extended the GC to gather context-specific collection information. We believe that exploring conceptually small but highly beneficial semantic extensions to the VM is a fruitful area of research.

Additional research were done in the field of automatic tuning. A works for automatically tuning linear algebra was done by Whaley and Dongarra [74] in the ATLAS project. Their work automatically
A.6. RELATED WORK

generates efficient linear algebra routines for a given microprocessor. They show an automatic generation of matrix multiplication routines for different hardware architectures. A work for automatically choosing a decision heuristic for a SAT solver was done by Lagoudakis and Littman [54]. In their work a decision heuristic is chosen according to a value function, which is calculated on the current state of the search. The value function is created beforehand, using a training set. There are also works on dynamic pretenuring in GC [46, 53]. These works use the same notion as ours of automatic tuning, however, these works do not try to select and manage data structures and tackle different problems in a different domain.