CoCo: Sound and Adaptive Replacement of Java Collections

Guoqing Xu

University of California, Irvine, CA, USA

Abstract. Inefficient use of Java containers is an important source of run-time inefficiencies in large applications. This paper presents an *application-level* dynamic optimization technique called CoCo, that exploits *algorithmic advantages* of Java collections to improve performance. CoCo dynamically identifies optimal Java collection objects and safely performs run-time collection replacement, both using pure Java code. At the heart of this technique is a framework that *abstracts* container elements to achieve efficiency and that *concretizes* abstractions to achieve soundness. We have implemented part of the Java collection framework as instances of this framework, and developed a static CoCo compiler to generate Java code that performs optimizations. This work is the first step towards achieving the ultimate goal of automatically optimizing away semantic inefficiencies.

1 Introduction

Large-scale Java applications commonly suffer from severe performance problems. A great deal of evidence [1,2,3,4,5] suggests that these problems are primarily caused by *run-time bloat* [6]—excessive memory usage and computation to accomplish simple tasks—often from inappropriate design/implementation choices (e.g., creating general APIs to facilitate reuse, using general implementations without any specialization, etc.) that exist throughout the program.

Container Inefficiencies. Previous studies [4,7,8,3,9,10] have found that an important source of run-time bloat is the inefficient use of container implementations. Standard libraries of object-oriented languages such as Java and C# contain collection frameworks that provide with users, for each abstract data type (such as `List`), many different implementations (such as `ArrayList` and `LinkedList`), each of which features a different design choice suitable for a specific execution scenario. However, in real-world development, choosing the most appropriate container implementation is challenging. As a result, developers tend to keep using the implementations that are most general or well-known (e.g., `HashSet` for `Set`), regardless of whether or not they fit the usage context. This is especially true in object-oriented programming, as developers can easily write code without deeply understanding implementation details of libraries, and the culture of object-orientation itself encourages generality and quick reuse of library functions. Inappropriate choices of container implementations can lead to significant performance degradation and scalability problems. More seriously, such bottlenecks can never be detected and removed by an existing optimizing compiler that is unaware of these different design principles and trade-offs.

G. Castagna (Ed.): ECOOP 2013, LNCS 7920, pp. 1–26, 2013.

Run-Time Container Replacement to Achieve Efficiency. We propose a novel container optimization technique, called CoCo, that is able to (1) determine at run time, for each container instance (e.g., a `LinkedList` object) used in the program, whether or not there exists another container implementation (e.g., `ArrayList`) that is more suitable for the execution; and (2) automatically and safely switch to this new container implementation (e.g., replace the old `LinkedList` object with a new `ArrayList` object *online*) for increased efficiency. While there exists work (such as Chameleon [7] and Brainy [9]) that could identify Java collection inefficiencies and report them to users for *offline* inspection, none of these techniques can change implementations *online*. An online approach outperforms an offline approach in the following two important aspects. First, a real-world execution is often made up of multiple phases, each with a different environment processing different types of data. Much evidence shows [11,12] that it is difficult to find a solution that is optimal for the entire execution. Our experimental results also demonstrate that many containers in large applications experience multiple switches during execution. This calls for novel techniques that can change implementations immediately after they become suboptimal. Second, an offline approach relies completely on the developer's effort to fix the detected problems while an online approach shifts this burden to the compiler and the run-time system.

Challenge 1: How to Guarantee Safety. Developing an online technique is significantly more difficult and constitutes the main contribution of this paper. Switching container implementations can easily cause inconsistency issues, leading to problematic executions or even program crashes. To illustrate, consider again the previous example where we replace a `LinkedList` object with an `ArrayList` object. Any subsequent invocation of a `LinkedList`-specific method (e.g., `getLast`) or type comparison (e.g., o `instanceof LinkedList`) can either cause the program to fail or change the semantics arbitrarily.

CoCo uses a combination of manually-modified library code and automatically-generated glue code (both are pure Java code) to perform optimizations. Unlike a traditional systems-level (blackbox) dynamic optimization technique that is completely separated from the applications being optimized, CoCo (1) advocates a methodology that can be used by library designers to create optimizable container classes, and (2) provides a static compiler to generate glue code that performs run-time optimizations on these optimizable container classes. All optimizations are performed at the application level within the (manually modified and automatically generated) Java classes, which encode human insight to direct optimizations.

CoCo stands for *co*ntainer *co*mbination—for each container object (whose type is supported by CoCo) created in a Java program, we additionally create a group of other container objects to which this particular object may be switched at run time. This group, together with the original container, is referred to as a *container combo* henceforth. CoCo initially uses the original container object as the active container, leaving this group of associated containers in the inactive state. All operations are performed only on the active container object. Upon adding a new element, the *concrete* element object is added only into the active container while CoCo creates an *abstraction* for this element and adds the abstraction to all inactive containers. This abstraction is actually a placeholder from which we can find the concrete object(s) it corresponds to.

Once a container switch occurs, an inactive container (e.g., a) that contains element abstractions is activated (and the originally active container b becomes inactive). If later a retrieval operation on a locates an abstraction, this abstraction is *concretized* by finding its corresponding concrete element (from b) to guarantee the safety of the subsequent execution. Section 4 discusses how to instantiate this framework to implement `List`, `Map`, and `Set` combos.

Challenge 2: How to Reduce Run-Time Overhead. It is obvious to see that naively creating container groups and performing abstraction-concretization operations can introduce a great amount of overhead. As our ultimate goal is to improve performance, it is equally important for CoCo to reduce overhead as well as ensuring safety. We employ three important optimization techniques to tune CoCo's performance: (1) drop inactive containers once we observe no optimization is possible; (2) use sampling to reduce profiling overhead; and (3) perform lazy creation of inactive containers.

Note that the performance benefit that CoCo may bring to a program results primarily from the *algorithmic advantage* of the selected implementation over the source implementation in an execution environment that favors the former, instead of from other factors such as a more compact design of the data structure or improved cache locality. In fact, performing online container replacement can, in many cases, increase memory space needed (as more objects are created) and/or hurt locality (as concrete elements can be separated in different containers). In order to improve the overall performance, we have to carefully select the container types to be optimized and evaluate various container replacement decisions to find the appropriate configurations under which the performance benefit outweighs the negative impact of the online replacement.

Section 2 gives an overview of how CoCo performs optimizations on containers and Section 3 presents a formalism that shows the general methodology to perform sound container replacement. Section 4 describes our implementations of the CoCo `List`, `Map`, and `Set`. Section 5 discusses our optimization techniques. We have evaluated CoCo on both a set of micro-benchmarks and a set of 5 real-world large-scale Java applications from which container bloat has been found previously. Our experimental results are reported in Section 7. Although we modify only a small portion of the Java Collections Framework, our experimental results show that CoCo is useful in optimizing micro-benchmarks (e.g., the speedup can be as large as $74\times$). For the 5 large-scale, real-world programs, CoCo is still effective—an average 14.6% running time reduction has been observed. Large opportunities are possible if more (both Java and user-defined) container classes can be modified to support CoCo optimizations.

The contributions of this work are:

- A general methodology to perform sound online container replacement for increased execution efficiency;
- an instantiation of this methodology in the Java Collections Framework that offers automated optimizations for 7 `Set`, `Map`, and `List` containers;
- three optimization techniques developed to reduce replacement overhead;
- an experimental evaluation of CoCo on both a set of micro-benchmarks and a set of real-world benchmarks; the results suggest that CoCo is effective and may be used in practice to optimize container usage.

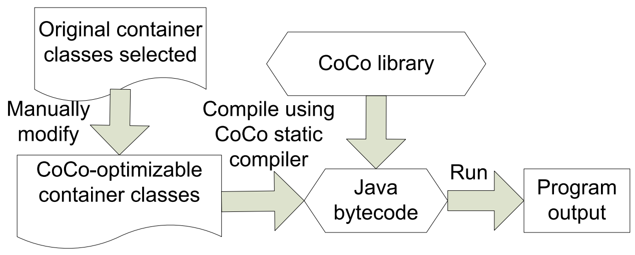
Original container classes selected
Manually modify
CoCo library
Compile using CoCo static compiler
CoCo-optimizable container classes
Java bytecode
Run
Program output

Fig. 1. An overview of the CoCo system

2 Overview

This section uses a `List` example to illustrate how CoCo performs container optimizations. The current version of CoCo focuses on Java built-in container classes, while the CoCo methodology can be applied to optimize arbitrary user-defined containers.

2.1 General Methodology

CoCo performs same-interface optimizations—given a container class that implements interface `I`, CoCo looks for potentially switchable candidates only within the types that implement `I`. Performance gains may be seen sometimes from switching implementations across different interfaces. For example, work from [7] shows that it can be beneficial to switch from ArrayList to LinkedHashSet in certain cases. It would be easy to extend CoCo to support multi-interface switches— the developer may need to create a wrapper class that serves as an adapter between interfaces. This class implements APIs of the original interface using methods of the new interface. We leave the detailed investigation of this approach to the future work.

From a set of all container classes that implement the same interface, we select those among which the online replacement may result in large performance benefit (at least large enough to offset the replacement overhead). In this paper, we focus on containers that have clear algorithmic advantages (e.g., lower worst-case complexity) over others in certain execution scenarios. For example, switching from a `LinkedList` to an `ArrayList` upon experiencing many calls to method `get`(i) may reduce the complexity of `get` from O(n) (where n is the size of this `List`) to O(1). This may have much larger benefit than switching from `ArrayList` to `SingletonList` (upon observing there is always one single element)—in this case, no significant algorithmic advantage can be exploited and the benefit resulting from space reduction may not be sufficient to offset the overhead of creating and maintaining multiple containers.

Figure 1 gives an overview of the CoCo system. For the selected container classes, we first modify them manually to add abstraction-concretization operations. The CoCo static compiler then generates glue code that connects these modified classes, performs run-time profiling, and makes replacement decisions. Next, both the generated glue classes and the modified container classes are compiled into Java bytecode, which is executed to enable optimizations. The entire container replacement task is achieved in

```
1 package coco.util;
2 class LinkedList extends java.util.LinkedList
3                     implements OptimizableList{
4    ListCombo combo;  // the combo object
5    LinkedList() { ... }   // the original constructor
     // constructor for CoCo
6    LinkedList(ListCombo c) {
7         this();
8         if(c == null) combo = new ListCombo(this);
9         else combo = c;
10  }
11
12  void add(Object o) {
13        if(combo == null)  this.add$CoCo(o);
14        else                    combo.add(o);
15  }
16  // this used to be add(Object o)
17  void add$CoCo(Object o) {/* add obj o*/}
18
19  Object get(int i) {
20        if(combo == null)  return this.get$CoCo(i);
21        else                    return combo.get(i);
22  }
23  //this used to be get(int i)
24  Object get$CoCo(int i) { ... }
25}

26  Interface OptimizableList{
27    void add$CoCo(Object o);  Object get$CoCo(); ...
28    void addAbstractElement(Object o, int containerID);
29    void replaceConcreteWithAbstract(Predicate p,
                                                int containerID);
30 }
```

(a) CoCo-optimizable **LinkedList** (manually modified from **java.util.LinkedList**) and interface **OptimizableList**

```
1 package coco.util;
2 class ListCombo implements List {
3       OptimizableList active;  OptimizableList[] inactive;
4       ListCombo(OptimizableList l){
5             active = l;
6             createInactiveLists();
7       }
8       OptimizableList[]  createInactiveLists(){
9             inactive = new OptimizableList[...];
10            ...
11            inactive[i] = new ArrayList(this);
12      }
13      void add(Object o) {
14           doProfiling(OPER_ADD);
15           active.add$CoCo(o);
16       }
17      Object get(int i) {
18           doProfling(OPER_GET);
19           return active.get$CoCo(i);
20      }
21      void doProfiling(int operation){
22          incCounter(operation);
23          if(active instanceof LinkedList &&
24                         NUM_GET > THRESHOLD) {
25           //Let's switch to ArrayList
26           // i  is the index of the ArrayList object in inactive
27              swap (active, inactive[i]);
28          }
29          ...
30      }
31}
```

(b) Glue class **ListCombo** that does profiling and makes replacement decisions (automatically generated by the CoCo static compiler)

Fig. 2. Examples of a CoCo-optimizable `LinkedList` obtained by manually modifying `java.util.LinkedList` and a combo class generated automatically by the CoCo static compiler

these (manually modified and compiler generated) Java classes at the application level, not in the run-time system.

2.2 Creating CoCo-optimizable Container Classes

Lines 1–25 in Figure 2 (a) show a skeleton of the CoCo-optimizable `LinkedList`, obtained from modifying the original `LinkedList` class in the Java Collections Framework. For ease of presentation, all examples shown in this section are simplified from the real container classes that CoCo uses. In addition, these examples are created only to illustrate how CoCo performs online container replacement, without considering how much overhead the code could incur. We will discuss various overhead reduction techniques in Section 5.

In order to be optimized by CoCo, the modified `LinkedList` must implement the CoCo-provided interface `OptimizableList`, whose skeleton is shown in lines 26–30. This interface declares a set of `X$CoCo` methods, each of which corresponds to a method `X` declared in interface `List` (where `X` can be `get`, `add`, `remove`, etc.). It additionally declares two methods (`addAbstractElement` and `replaceConcrete-WithAbstract`) that will be implemented to perform abstraction-concretization operations. An `X$CoCo` method has exactly the same signature as method `X`. For each `X` in the original version of `LinkedList`, we move its entire body into `X$CoCo`, which, thus, becomes the method that implements the functionality of `X`. Method `X` will be used to choose container implementations at run time.

Container Combo. Each CoCo-optimizable container class has an associated `combo` object of type `ListCombo` (line 4 in Figure 2 (a)), responsible for profiling container operations and making replacement decisions. Figure 2 (b) shows a skeleton of the glue class `ListCombo`, which is generated automatically by the static CoCo compiler, given a container interface (e.g., `List`) and a selected set of its implementing classes to be optimized (e.g., `ArrayList` and `LinkedList`).

`ListCombo` also implements `List`, and thus, it contains all methods (`X`) declared in `List`. Each `X` in `ListCombo` is implemented as a *method dispatcher*. To illustrate, consider method `add` (line 13 in Figure 2 (b)). It first records that this is an ADD operation (line 14), and then invokes method `add$CoCo` on the currently active list `active` (line 15). Note that method `add$CoCo` in different container implementations can be invoked by simply updating the field `active` during execution.

Example. Consider again class `LinkedList` in Figure 2 (a). For each constructor in the original version of `LinkedList`, we add a new constructor that has one additional parameter of type `ListCombo` (line 6 in Figure 2 (a)). In a client program that calls the original constructor (at line 5), our JVM will replace this call site with a call to the new constructor and pass `null` as the argument. Upon receiving `null`, this constructor creates a new `ListCombo` (line 8) that, in turn, creates a group of other container objects (line 8-12 in Figure 2 (b)) to which this `LinkedList` object may be switched at run time. The `LinkedList` object becomes the first active `List` in this combo (line 5 in Figure 2 (b)). When an inactive container is created, the combo object is also passed into its constructor (line 11 in Figure 2 (b)), and thus, this combo object will be shared among all the candidate containers.

The combo is essentially a central method dispatcher—whenever method `add` (e.g., in `LinkedList`) is invoked from a client, it calls method `add` on the combo object (line 14 in Figure 2 (a)), which, in turn, invokes method `add$CoCo` on the currently active `List` object (line 15 in Figure 2 (b)). If, at a certain point during execution, the profiling method (line 21-30 in Figure 2 (b)) observes that the active `List` is a `LinkedList` and the total number of `get(`i`)` operations invoked is greater than a threshold value (line 23), it changes the active container to `ArrayList`—the only thing that needs to be done is to find the `ArrayList` object in the inactive container array and make it active (line 27 in Figure 2 (b)). An illustration of the combo structure can be found in Figure 3.

Preserving Semantics. From a client's perspective, using this combination of containers has no influence on the behaviors of the original `LinkedList`. The client code always interfaces with the `LinkedList` methods, although the actual "service" could be provided by a different container object. This design guarantees type safety of the forward execution—we make the modified `LinkedList` a subclass of the original Java `LinkedList` (line 2 in Figure 2 (a)), and thus, no failure would result from

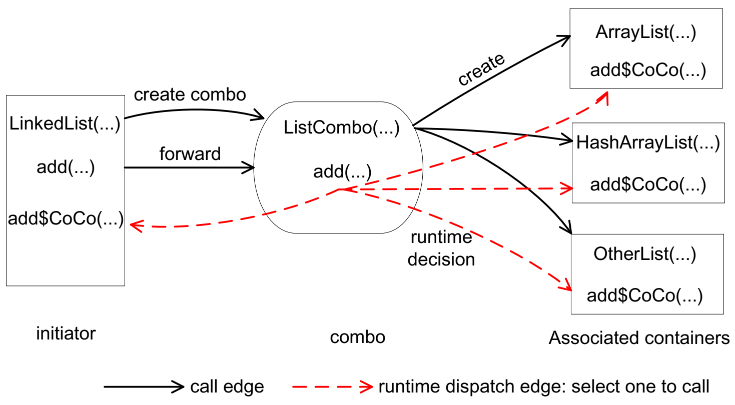

Fig. 3. Graphical illustration of a container combo

future operations that rely on the original type `LinkedList`, such as `instanceof` and the invocation of `LinkedList`-specific methods (e.g., `getLast`).

While manual work is needed to turn a Java (or user-defined) container into a CoCo-optimizable container, much of the original code can be reused and the user needs to add only a few methods, such as `X$CoCo`. The entire class `ListCombo` (except `doProfiling`) is generated automatically by the CoCo compiler. Furthermore, the intended users of CoCo are library designers and implementers, and thus, no extra effort is needed from the end developers. For method `doProfiling` that makes run-time replacement decisions, the condition code (e.g., lines 23-28) can be either generated from a set of user-specified rules (such as those used by Chameleon [7]), or filled in by human experts manually. The current version of CoCo uses the latter approach, while it is easy to develop a rule engine to translate user-defined rules into code predicates. If the designer wishes to combine implementations with conflicting specifications (e.g., some accept `null` values while others do not), she may need to explicitly insert checks in the combo code to direct the dispatch. For example, a `null` value cannot be inserted into a container that does not handle it.

2.3 Using Abstraction and Concretization to Ensure Safety

When an inactive container is activated, all container operations will be performed on it. Key to providing soundness guarantee is, thus, to make sure that any container, if becoming active, can provide service correctly. A natural idea is to move all elements from one container to another upon a switch. However, significant run-time overhead may result from regularly moving elements around containers. If the switch decision is inappropriate and we need to switch back to the original container, moving all elements back and forth can create a great amount of redundant work. To achieve efficiency, we propose an *on-demand* approach—an element is moved from an inactive container to an active container only when it is requested by the client. If the switch decision

```
1 class LinkedList implements OptimizableList {
2       Entry first, last;
3         ...
4       void add$CoCo(Object o){
5           insert(o);
6           for(OptimizableList l : combo.inactiveLists()){
7                 l.addAbstract(o, ID_LINKED_LIST);
8           }
9       }
10      Object get$CoCo(int i) {
11          Object o = findEntry(i);
12          if(o instanceof AbstractElement) {
13              AbstractElement ae = (AbstractElement)o;
14              Object[] concreteObjs =
                          ae.concretize(ID_LINKED_LIST);
15              replaceAndExpand(i, concreteObjs);
16               o = findEntry(i);
17          }
18          return o;
19      }
20      void addAbstractElement
                    (Object o, int containerID){
21          AbstractElement ae = createAbstraction
                                        (o, containerID);
22          insert(ae);
23      }
24      ...
25 }
26 class AbstractElement{
27      ListCombo combo;  int targetContainerID;
28      Predicate p;
29      Object[] concretize(int newContainerID) {
30        OptimizableList l = combo.getRuntimeList
                                  (targetContainerID);
31        Object[] objs = new Object[p.size()];
32        for(Integer index : p.iterateAll())
33        {  objs[...] = l.get$CoCo(index);  }
34        l.replaceConcreteWithAbstract
                          (p, newContainerID);
35        return objs;
36      } ...
37 }
```

Fig. 4. An abstraction-concretization example in LinkedList

is appropriate, the active container will remain active for a relatively long time and concrete elements will be gradually moved to this container as more operations are performed.

To do this, when an ADD operation is performed, it is insufficient to add the incoming object only into the active container—we have to additionally record some information of this element in all inactive containers, so that if any one of them becomes active later and this element is requested by a client, the container would have a way to find it and return it to the client. This information is referred to as the *abstraction* of this concrete element (or sometimes *abstract element*). If an abstraction is found during the retrieval of an element, this abstraction is *concretized* to provide the concrete element(s) it represents. For different types of containers (e.g., `Map`, `Set`, `List`, etc.), we may need to create different types of abstractions. Section 3 discusses a general methodology to create abstractions.

Example. Figure 4 shows part of the modified `LinkedList` responsible for element abstraction and concretization. Highlighted methods are declared in interface `OptimizableList`. After inserting a concrete element *o* into the list (line 5), method `add$CoCo` adds an abstraction of *o* into all inactive lists (lines 6–8). Once an abstraction is found in a retrieval (lines 12–17), it is concretized to get an array of concrete elements it represents (line 14), which are then inserted into the list to replace this abstraction.

Lines 26–36 of Figure 4 show class `AbstractElement`, which is the abstraction that CoCo uses for `List`. Here an abstraction has three pieces of information (lines 27–28): the ID of the container object where the concrete element(s) represented by this abstraction are located (i.e., `targetContainerID`), the combo object from which this container object can be obtained (i.e., `combo`), and a `predicate` (i.e., `p`) that specifies the exact locations of the concrete elements in their host container.

Predicate. A predicate is defined based on how elements are added into and retrieved from a container. For example, for `List`, elements are added and retrieved based on their *indices*, and thus, the predicate in each element abstraction is defined over the domain of indices (integers), that is, each predicate represents a range of indices (e.g., [0, 15]). When an abstraction is concretized (line 14 in Figure 4), all the elements whose indices satisfy the predicate of the abstraction (lines 32–33) are retrieved from their host container in order to fulfill the client's request. These concrete elements are then moved into this active `List` to avoid future concretization operations (line 15). In addition, we need to remove these concrete elements from their old host container by creating an abstraction there to represent them (line 34). If CoCo switches back to this (old) container later, the same (concretization) process will be performed to retrieve the elements.

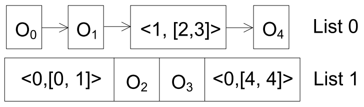

Fig. 5. A `List` combo in which both containers have abstractions

Figure 5 shows an example `List` combo resulting from the code shown in Figure 4. In this combo, both the `LinkedList` (List 0) and the `ArrayList` (List 1) have abstractions (i.e., placeholders) and concrete elements. For example, the abstraction $\langle 1, [2, 3] \rangle$ is the placeholder of the No.2 and No.3 elements and it specifies that these two elements are currently in List 1. Different abstractions can be defined by creating different `AbstractElement` classes (in Figure 4). Section 4 discusses the abstractions that CoCo currently uses for `List`, `Map`, and `Set`, while the general methodology proposed in this paper can be applied to create abstractions for all (Java built-in or user-defined) data structures in order to enable this optimization.

Calls to methods that are specific to each container implementation (i.e., that are not declared in the container Interface) cannot be forwarded to the combo object. For example, if method `getLast` is invoked on a `LinkedList` object after the active container in the combo becomes an `ArrayList`, this call cannot be performed on the `ArrayList` because `getLast` does not exist in `ArrayList`. In this case, `getLast` is still invoked on the `LinkedList` object, but we need to modify this method and concretize an abstraction to bring the last concrete element back to the `LinkedList` to fulfill the request (e.g., if the last element is not the `LinkedList`). It is subject to the developer how the concretization should be performed.

3 Formal Descriptions

This section formalizes the notion of a container, and in that context, formally describes CoCo's core idea on run-time container replacement. Although CoCo currently supports only a few Java built-in containers, this formal framework defines a methodology that can be used to switch arbitrary container implementations. Any instantiation of this framework is guaranteed to be sound.

Definition 1. *(*Container*) A container Σ is a triple $\Sigma = \langle \mathsf{X}, \prec, f \rangle$, where X is a set of concrete elements and $\prec$ is a partial order on X. $\prec$ is determined by a property encoding function $f(x)$ that maps each $x \in \mathsf{X}$ to an integer i_x that encodes a certain property of x. For any two elements $x_1, x_2 \in \mathsf{X}$, $x_1 \prec x_2$ iff. $f(x_1) < f(x_2)$.*

Partial order $\prec$ determines how elements are stored in a container. x_1 precedes x_2 in the underlying data structure of the container if $x_1 \prec x_2$. The position of each element depends on a certain property of this element used by the container, which is encoded by function f. For example, for `List`, f maps each x to a unique index; for `HashMap`, f is a hash function that maps each x to the index of the hash bucket where x should be stored based on x's hashcode; for `TreeMap`, f maps each x to the index of x in a list of all elements in the map sorted by the results of their `compareTo` method. This index determines when x will be reached in an in-order traversal of the binary search tree used by `TreeMap`. If neither $x_1 \prec x_2$ nor $x_2 \prec x_1$, there is no constraint regarding the positions of x_1 and x_2. For example, if the hash function (in a `HashMap`) maps both x_1 and x_2 to the same bucket slot, the way they are stored is subject to the specific implementation of `HashMap`.

Definition 2. *(*CoCo-optimizable container*) A CoCo-optimizable container* Γ *is a six-tuple* $\Gamma = \langle \mathsf{X}, \mathsf{A}, \prec, f, \alpha, \gamma \rangle$*, where* X *is a set of concrete elements,* A *is a set of abstract elements,* $\prec$ *is a partial order on* $\mathsf{X} \cup \mathsf{A}$*,* f *is a property encoding function,* α*:* $\mathsf{X} \rightarrow \mathsf{A}$ *is an abstraction function that maps each concrete element to an abstraction, and* γ*:* $\mathsf{A} \rightarrow 2^{\mathsf{X}}$ *is a concretization function that maps each abstract element to a subset of concrete elements it represents. Partial order* $\prec$ *is redefined as follows:*

(a) $\forall x_1, x_2 \in \mathsf{X}$: $x_1 \prec x_2 \Leftrightarrow f(x_1) < f(x_2)$
(b) $\forall a_1, a_2 \in \mathsf{A}$: $a_1 \prec a_2 \Leftrightarrow \mathsf{max}_{x \in \gamma(a_1)}(f(x))$
$< \mathsf{min}_{y \in \gamma(a_2)}(f(y))$
(c) $\forall a \in \mathsf{A}, x \in \mathsf{X}$: $a \prec x \Leftrightarrow \mathsf{max}_{y \in \gamma(a)}(f(y)) < f(x)$
(d) $\forall a \in \mathsf{A}, x \in \mathsf{X}$: $x \prec a \Leftrightarrow f(x) < \mathsf{min}_{y \in \gamma(a)}(f(y))$

Each CoCo-optimizable container (such as the `LinkedList` shown in Figure 2 (a)) is a mixture of concrete and abstract elements. The (new) partial order $\prec$ is defined on the union of these two groups of elements. The definition of $\prec$ does not change when two concrete elements are compared (i.e., shown in condition (a)). An abstraction $a_1 \prec$ another abstraction a_2 only when the "greatest" concrete element abstracted by $a_1 \prec$ the "smallest" concrete element abstracted by a_2 (i.e., shown in (b)). (c) and (d) show comparisons between an abstraction a and a concrete element x: $a \prec x$ only if the "greatest" concrete element a abstracts $\prec x$. The abstraction function α and the concretization function γ are user-defined and are specific to each type of container optimized by CoCo.

Here the concrete element set X and the abstract element set A are separated for ease of presentation and formal development. In our implementation, they are stored together in the underlying data structure (e.g., the data array for `ArrayList`, the bucket array for `HashMap`, etc.) of a container that used to store only concrete elements. An abstract element in a container is stored in the location where its corresponding concrete element(s) would have been stored if they were in this container.

Definition 3. *(*Contiguous abstraction*) An abstraction* $a \in \mathsf{A}$ *in a container object is a contiguous abstraction iff.* $\forall x_1, x_2 \in \gamma(a)$: $(\nexists x \in \mathsf{X}: x_1 \prec x \prec x_2) \wedge (\forall a' \in \mathsf{A} : a' \neq a \Rightarrow \nexists y \in \gamma(a') : x_1 \prec y \prec x_2)$.

A contiguous abstraction represents a range of concrete elements that should be stored contiguously in the container. No other concrete element either directly in the container or abstracted by another abstraction can be put in the middle of them. Allowing only contiguous abstractions in a container simplifies significantly the implementations of the abstraction function α and the concretization function γ. This definition also implies that different contiguous abstractions do not overlap.

Definition 4. *(*Well-formed container*) A CoCo-optimizable container* $\mathsf{\Gamma} = \langle \mathsf{X}, \mathsf{A}, \prec, f, \alpha, \gamma\rangle$ *is a well-formed container, if (1)* $\forall a \in \mathsf{A}$*: a is a contiguous abstraction and (2)* $\forall x \in \mathsf{X}$*:* $\nexists a \in \mathsf{A}$*:* $x \in \gamma(a)$*.*

Well-formedness has two important aspects: all abstractions are contiguous abstractions and no abstraction abstracts a concrete element that exists in the container (i.e., an abstraction abstracts only concrete elements located in other containers). It is easy to see that a container with no abstraction is a well-formed container; so is a container with one single abstraction that abstracts all elements.

Definition 5. *(*Well-formed container combo*) A container combo* C *is an n-tuple* $\mathsf{C} = \langle \mathsf{\Gamma}_1, \mathsf{\Gamma}_2, \ldots, \mathsf{\Gamma}_{n-1}, i\rangle$*, where* $\mathsf{\Gamma}_1, \ldots, \mathsf{\Gamma}_{n-1}$ *are CoCo-optimizable containers and* $i \in [1, n\text{ -}1]$ *is the index of the active container.*

A container combo is well-formed iff.

- $\forall j \in [1, n \text{ - } 1]$, $\mathsf{\Gamma}_j$ *is well-formed; and*
- $\forall j, k \in [1, n \text{ - } 1]$, $j \neq k$*: (*$|\mathsf{X}_j| + \sum_{a \in \mathsf{A}_j} |\gamma_j(a)| = |\mathsf{X}_k| + \sum_{a \in \mathsf{A}_k} |\gamma_k(a)|$*); and*
- $\forall j \in [1, n \text{ - } 1]$*:* $\forall x \in \mathsf{X}_j$*:* $\forall k \in [1, n \text{ - } 1]$*:* $k \neq j \Rightarrow \exists a \in \mathsf{A}_k$*:* $x \in \gamma(a)$

A well-formed container combo must first have all its containers well-formed. The last two conditions concern the relationships between the containers in the combo: each container must have the same number of elements if all abstractions are concretized; and for any concrete element x in a container Γ_j, there must exist an abstraction in each container Γ_k ($k \neq j$) that abstracts x.

Definition 6. *(*ADD, GET, and SWITCH*) An ADD(x) operation performed on a container combo* $\langle \mathsf{\Gamma}_1, \ldots, \mathsf{\Gamma}_{n-1}, i\rangle$ *adds a concrete element* x *to set* X_i *of the active container* $\mathsf{\Gamma}_i$*, and appropriately updates abstract element set* A_j *in each inactive container* $\mathsf{\Gamma}_j$ *to add* x*'s abstraction.*

A GET operation first looks for the target element in X_i *of the active container* $\mathsf{\Gamma}_i$*. If the element does not exist in* X_i*, it finds the abstraction* $a \in \mathsf{A}_i$*, uses concretization function* γ_i *to retrieve all elements* $\gamma_i(a)$*, appropriately updates set* A_j *in each container* $\mathsf{\Gamma}_j$ *, and returns the target element* $x \in \gamma_i(a)$*.*

A SWITCH operation causes a container combo $\langle \mathsf{\Gamma}_1, \ldots, \mathsf{\Gamma}_{n-1}, i\rangle$ *to become* $\langle \mathsf{\Gamma}_1, \ldots, \mathsf{\Gamma}_{n-1}, j\rangle$ *(*$i \neq j$*), without changing the internal state of each container.*

These definitions specify the behaviors of the three most important container operations. Section 4 discusses how these operations are performed for different types of containers such as `List`, `Map`, and `Set`.

Table 1. Containers currently supported by CoCo

Combo type	*Participating containers*	*Condition*	*Switch To*
java.util.List	ArrayList	#contains $>$ X $\wedge$ size $>$ Y	HashArrayList
	LinkedList	#get $>$ X $\wedge$ size $>$ Y	ArrayList
	HashArrayList	#contains $<$ X $\vee$ size $<$ Y	ArrayList
java.util.Map	HashMap	size $<$ Y	ArrayMap
	ArrayMap	size $>$ Y	HashMap
java.util.Set	HashSet	size $<$ Y	ArraySet
	ArraySet	size $>$ Y	HashSet

Definition 7. *(*Soundness of container switch*) A SWITCH operation that causes a container combo* C *to become* C' *is a sound switch if any element added by an ADD operation on* C *can be successfully retrieved by a corresponding GET operation on* C'.

Lemma 1. *(*Well-formedness implies soundness*) If the well-formedness of a container combo* C *is preserved by each (ADD or GET) operation, any SWITCH operation that occurs on* C *is sound.*

Proof Sketch. If C is always well-formed during the execution, an element added to C is either in the concrete element set X_i of the active container Γ_i, or in set X_j of an inactive container Γ_j and appropriately abstracted by an abstraction $a \in \mathsf{A}_i$. In the former case, it can be found directly in X_i, while in the latter case, it can be found by using γ_i to concretize abstraction a (as long as γ_i is sound). □

This section presents a methodology for creating sound container combos. Optimizations of any container implementations are guaranteed to be sound if they are instances of this formal framework. The library designers are responsible for creating appropriate optimizations for their containers and make them strictly follow the CoCo methodology. The next section describes the 7 containers CoCo currently supports and our implementations of their combos. We will also demonstrate, for each type of combo, how our specific implementations of abstraction and concretization preserve its well-formedness.

4 Implementation

We have modified four Java built-in container classes and implemented another three from scratch to instantiate the framework discussed in Section 3. Table 1 shows the container combos, when switches are performed, and switch destinations. Note that a subset of the switch conditions is adapted from the rules discovered and used in Chameleon [7]. Additional rules can be easily added by modifying the method `doProfiling` in each combo class (shown in Figure 2 (b)). Among the seven containers, `HashArrayList`, `ArrayMap`, and `ArraySet` are implemented by ourselves; the rest of them are modified from their original versions in the GNU classpath library [1].

These containers are selected because (1) they are all designed for general use, and (2) each container has clear time efficiency in a certain usage scenario. Future work may

[1] `www.gnu.org/s/classpath/`

Algorithm 1. List ADD that inserts object x at the index $index$ into a combo whose active container is i.

```
Input: List combo $\langle \Gamma_1, \ldots, \Gamma_{n-1}, i \rangle$, Index $index$, Element $x$
Suppose $L_j$ is the underlying data structure in List $\Gamma_j$ that stores $X_j$ and $A_j$
foreach $\Gamma_j$, $j \in [1, n-1]$ do
    $newIndex \leftarrow$ recalculateIndex($L_j$, $index$)
    if $j = i$ then
        /* update the active List */
        if $L_j[newIndex]$ is an abstract element $a = \langle k, [index_m, index_n] \rangle$ then
            /* $k$ is the ID of the List that contains the concrete element,
            $[index_m, index_n]$ is the predicate specifying a region*/
            split this abstraction into $a_1 = \langle k, [index_m, index - 1] \rangle$ and $a_2 =$
            $\langle k, [index + 1, index_n + 1] \rangle$
            replace $a$ with $[a_1, x, a_2]$ in $L_j$
            foreach abstract element $\langle t, [index_q, index_r] \rangle$ following $a_2$ in $L_j$ do
                update predicate $[index_q, index_r]$ to $[index_q + 1, index_r + 1]$
        else
            /*This is a concrete element*/
            insert $x$ into $L_j$ at the index $newIndex$
    else
        /*update an inactive List*/
        /*Initially each inactive List has one abstraction $\langle i, [0, 0] \rangle$ */
        if $L_j[newIndex]$ is an abstraction $a = \langle k, [index_m, index_n] \rangle$ then
            if $k = i$ then
                update $a$ to be $\langle k, [index_m, index_n + 1] \rangle$
            else
                create new abstraction $a' = \langle i, [index, index] \rangle$
                split $a$ into $a_1 = \langle k, [index_m, index - 1] \rangle$ and $a_2 = \langle k, [index + 1, index_n + 1] \rangle$
                replace $a$ with $[a_1, a', a_2]$ in $L_j$
                $a \leftarrow a_2$
        else
            /*$L_j[newIndex]$ is a concrete element*/
            create new abstraction $a' = \langle i, [index, index] \rangle$
            insert $a'$ into $L_j$ at the index $newIndex$
            $a \leftarrow a'$
        foreach abstract element $\langle t, [index_q, index_r] \rangle$ following $a$ in $L_j$ do
            update predicate $[index_q, index_r]$ to $[index_q + 1, index_r + 1]$
return
Function recalculateIndex($L$, $index$)
Input: data structure $L$, the original index given by the client $index$
Output: The new index in the presence of abstractions
$count \leftarrow 0$
foreach $k \in [0, |L| - 1]$ do
    if $L[k]$ is an abstract element $\langle t, [index_m, index_n] \rangle$ then
        if $index_m \leq index \leq index_n$ then
            return $k$
        $count \leftarrow count + (index_n - index_m + 1)$
    else
        if $count = index$ then
            return $k$
        $count \leftarrow count + 1$
return $k$
```

Algorithm 2. List GET that retrieves an element at the index $index$ from a combo whose active container is i.

```
Input: List combo ⟨$\Gamma_1, \ldots, \Gamma_{n-1}, i$⟩, Index $index$
Output: Element $x$
1  $newIndex \leftarrow$ recalculateIndex($L_i$, $index$)
2  if $L[newIndex]$ is an abstract element $a = \langle k, [index_m, index_n]\rangle$ then
3      $newIndex_m \leftarrow$ recalculateIndex($L_k$, $index_m$)
4      $newIndex_n \leftarrow$ recalculateIndex($L_k$, $index_n$)
5      /*Concretize this abstraction to get the range of elements it represents*/ Array $r \leftarrow$ getElements($L_k$,
       $newIndex_m$, $newIndex_n$)
6      replace $a$ with $r$ in $L_i$
7      create abstraction $a' = \langle i, [index_m, index_n]\rangle$
8      replace $r$ with $a'$ in $L_k$
9      $size \leftarrow index_n - index_m + 1$
10     foreach $\Gamma_j, j \in [1, n-1] \wedge j \neq i \wedge j \neq k$ do
11         $index_j \leftarrow$ recalculateIndex($L_j$, $index$)
12         $L_j[index_j]$ must be an abstraction $a'' = \langle k, [index_p, index_q]\rangle$
13         if $index_q - index_p + 1 = size$ then
14             $L_j[index_j] \leftarrow \langle i, [index_p, index_q]\rangle$
15         else
16             split $a''$ into $a_1 = \langle k, [index_p, index_m - 1]\rangle$ and $a_2 = \langle k, [index_n + 1, index_q]\rangle$
17             create new abstraction $aa = \langle i, [index_m, index_n]\rangle$
18             replace $a''$ with $[a_1, aa, a_2]$ in $L_j$
19     $newIndex \leftarrow$ recalculateIndex($L_i$, $index$)
20 return $L[newIndex]$
```

investigate container replacement based on other performance metrics (such as space efficiency and concurrency). In this section, we show only the ADD and GET operations for each combo, while our implementation supports all operations of the containers shown in Table 1. The implementations of the CoCo `Map`, `List`, and `Set` combos are publicly available at `www.ics.uci.edu/~guoqingx/tools/coco.jar`.

4.1 List Combo

The structure of the `List` combo has been shown in Figure 2. Here we discuss only how abstractions and concretizations are performed. Algorithm 1 illustrates the ADD operation of the `List` combo. As mentioned in Section 3, for each `List` in the combo, concrete elements and abstract elements are stored together in its underlying data structure that used to contain only concrete elements. Suppose L is such a data structure (e.g., the data array for `ArrayList`, the linked structure for `LinkedList`, etc.). We use $L[k]$ to denote the k-th element in L, regardless of the type of L. Note that the sequence of elements in L is determined by the partial order $\prec$ of the container, which is, in turn, determined by the property encoding function f. For all `List` containers, f maps each element to a unique index, which is used to determine the position of this element in L.

For `List`, each abstraction has the form $\langle k, [m, n]\rangle$, where k is the ID of the container that has the concrete element and $[m, n]$ is the predicate that specifies a range of indices of the concrete elements represented by this abstraction. Note that CoCo currently allows only contiguous abstractions, and thus, a range is sufficient to represent a predicate. Richer predicates can be used if the contiguousness requirement is relaxed in the future. Figure 5 has illustrated a simple (well-formed) `List` combo where both participating containers have abstractions.

Implementation of List ADD. Given an index $index$, we first compute a new index $newIndex$ that corresponds to $index$ in the presence of abstractions (line 3 in Algorithm 1). Function `recalculateIndex` is shown in lines 35–46. Next, we add this incoming object x into the active container Γ_i (lines 4–15). If the element at $newIndex$ is an abstract element (line 6), this abstraction is split into two separate abstractions (line 9) and then x is inserted between them (line 10). In addition, as the addition of x increases the index of each existing element following x by 1, we need to update the predicates in all the abstractions following a_2 in L to make them reflect the new indices (lines 11–12) (note that a_2 itself has been updated by line 9). In our implementation, abstractions in each L are indexed by a separate array, which allows quick update of the predicates.

Abstractions in all the inactive `Lists` need to be updated in a similar manner (lines 16–33) to account for this new element. In each inactive `List` Γ_j (where $j \neq i$), we first find the element at the index $newIndex$. If it is an abstraction and the host container in this abstraction happens to be the active `List` (line 20), we simply grow the range by 1 (line 21). Otherwise, we have to split this abstraction into two separate abstractions and insert a new abstraction a' (whose host container is the active container) between them (lines 23–25). If $\mathsf{L}_j[newIndex]$ is a concrete element, a new abstraction a' is created and inserted into L_j (lines 28-30). Eventually, the predicate in each abstraction following a is updated with a new index range (lines 32–33).

It is clear that if no switch is performed on the combo, the active `List` has all concrete elements and there is only one abstraction in each inactive `List` that abstracts all of them.

Implementation of List GET. Shown in Algorithm 2 is the GET operation implemented in CoCo. If the element at the index $newIndex$ is a concrete element, it is returned immediately (line 20). Otherwise, we retrieve all the concrete elements represented by the abstraction (lines 3–5) and bring them into the active container (line 6). An abstraction is then created to replace them in their original host container Γ_k (lines 7–8). As the host container of these elements is changed from Γ_k to Γ_i, code at lines 9–18 updates their corresponding abstractions in other inactive containers (not Γ_k or Γ_i) with the new host information. If the abstraction abstracts exactly the same range of concrete elements (line 13), we simply change its host container from k to i (line 14); otherwise, this abstraction needs to be split into two a_1 and a_2, and a new abstraction aa is created to represent this range (line 17). aa is inserted between a_1 and a_2 in L_j.

Note that once an abstraction a is encountered during a retrieval, we concretize the abstraction and move the entire range of concrete elements into the active container (lines 5–8). This is conceptually similar to a cache line fill. An alternative is to split a into three abstract elements $\langle k, [index_m, index]\rangle$, $\langle k, [index, index]\rangle$, and $\langle k, [index + 1, index_n]\rangle$, and then concretize only $\langle k, [index, index]\rangle$ to get the exact element requested. We have also implemented this approach for the general GET operation but found that it is much less effective than the one shown in Algorithm 2. It is primarily because elements in a contiguous region are often visited together (e.g., using an Iterator) and getting them one at a time can increase cache misses significantly, especially for array-based containers.

However, we do use this alternative approach to implement methods that are specific to each container implementation (i.e., that are not declared in the common Interface). For example, method `getLast` (that is specific to `LinkedList`) needs to retrieve only the last element. If this last element in the `LinkedList` is a concrete element, it is directly returned; if it is an abstraction ($\langle k, [start, |\mathsf{L}| - 1]\rangle$), we first split it into two abstractions ($\langle k, [start, |\mathsf{L}| - 2]\rangle$, $\langle k, [|\mathsf{L}| - 1, |\mathsf{L}| - 1]\rangle$), and concretize only ($\langle k, [|\mathsf{L}| - 1, |\mathsf{L}| - 1]\rangle$) that represents the last element. The insight here is that in most cases where these implementation-specific methods are invoked, their containers are in the inactive state. We should avoid bringing many concrete elements from an active container back to an inactive one, which may lead to significant performance degradation.

Implementation of HashArrayList. Container `HashArrayList` is an `ArrayList`-based data structure. It maintains an `ArrayList` and a `HashMap` internally, and duplicates concrete elements between them. The `HashMap` contains only concrete elements and is used to perform `contains`-related operations. Abstractions are only allowed to be added into the `ArrayList`. When a `HashArrayList` is activated, all concrete elements in the combo are added into its `HashMap`. Abstraction/concretization operations are performed only between the "list" part of the `HashArrayList` and other lists in the combo; its "map" part always contains concrete elements as long as the `HashArrayList` is active.

Theorem 1. *(*List combo soundness*) Any SWITCH operation performed on the List combo (whose ADD and GET are shown in Algorithm 1 and Algorithm 2, respectively) is a sound switch.*

Proof Sketch. To prove this, it is important to show that each ADD and GET preserves the well-formedness of the combo. This can be easily seen from Algorithm 1 and Algorithm 2: (1) each element added to the active list (lines 4–15 in Algorithm 1) is well abstracted in each inactive list (lines 17–33 in Algorithm 1); and (2) each concretization replaces an abstract element in the active list with the concrete elements it represents (lines 5–6 in Algorithm 2). These concrete elements (in their old host) are replaced with a new abstract element (lines 7–8 in Algorithm 2). Abstractions corresponding to these elements in all the other inactive lists are updated to point to their new locations (lines 10–18 in Algorithm 2). Hence, well-formedness is preserved by both the element addition and the concretization. □

4.2 Map and Set Combos

The general algorithms (in Algorithm 1 and Algorithm 2) used for `List` can be naturally adapted to create `Map` and `Set` combos. For example, for a `Map` combo, when a pair of objects is added into the `HashMap` object (which is active), we can create an abstraction in the `ArrayMap` whose predicate records the bucket index of this pair in the `HashMap`. However, unlike the `List` combo where all participating `Lists` have the same property encoding function f (thus we can easily merge adjacent abstractions as they represent adjacent concrete elements), `HashMap` and `ArrayMap` have different f. Elements in a `HashMap` are ordered based on their hashcodes while elements in an `ArrayMap` are ordered simply based on indices. As such, adjacent abstractions in the `ArrayMap` may represent concrete elements far away from each other in the `HashMap`. These abstractions may not be easily merged and we may end up with a great number of abstractions in each container, leading to increased space consumption and concretization overhead.

We develop a simpler approach for both `Map` and `Set` combos—for each combo, we maintain only one single abstraction in each inactive container that represents all concrete elements; upon a container switch, this abstraction is immediately concretized by moving the entire collection to the newly active container. We do not split abstractions because it may not be as beneficial for `Map` and `Set` as for `List`. Furthermore, as `ArraySet` and `ArrayMap` are designed to hold a very small number of elements, it is relatively inexpensive to perform MOVE_ALL upon each switch (e.g., much less costly than performing MOVE_ALL for a `List`).

Theorem 2. *(*Map and Set combo soundness*) Any SWITCH operation performed on the Map/Set combo is a sound switch.*

Proof Sketch. It is clear to see that the active container always has all the concrete elements. Each ADD or GET operation only updates the active container, and hence, the well-formedness of the combo is guaranteed. □

4.3 Discussion

Thread Safety. Our combo implementations (described in this section) are thread-safe. Because the client code always interfaces with methods in the original container, the concurrent behavior of the program is not influenced by any container switch. To illustrate, consider a switch from a `Hashtable` (which is thread-safe) to a `HashMap` (which is thread-unsafe). Because the client still invokes methods in `Hashtable` after the switch and these methods are appropriately synchronized, the fact that the actual service is provided by `HashMap` would not create any side effect. In addition, because the list of all associated containers is created inside the original container, whether or not these associated containers can be shared among threads depends on whether the original container is shared. No replacement will change the sharing property of the container (e.g., from being shared to thread-local or vice-versa). However, concurrent containers (e.g., those in `java.util.concurrent`) often use non-blocking algorithms (e.g., compare-and-swap). Forming a combo with both concurrent containers and non-concurrent, thread-unsafe containers may cause concurrency issues, which should not be allowed. Mixing only concurrent containers can be thread-safe as long as the abstraction and concretization operations are appropriately synchronized.

Handling of Operations with Incompatible Specifications. In some cases, implementations of the same operation (e.g., a method declared in a Java interface) in different containers may be incompatible, making it difficult for combo containers to provide the same service to the client. For example, different `Set` implementations may iterate elements in different orders; unexpected program behaviors may result from switching implementations and traversing elements in a different order. The easiest way to solve the problem is to select only containers whose common methods have the same pre- and post-conditions to form a combo. However, this approach can limit significantly the optimization capabilities.

An alternative is to switch back to the original (user-chosen) container immediately upon the invocation of a method that is incompatible with its corresponding method in the original container. Concretizations are subsequently performed to bring necessary elements back. While this approach preserves semantics, it may potentially cause frequent container switches and element copies, leading to increased overhead. One possible way to alleviate the problem is to disable CoCo optimizations if such incompatible methods are frequently invoked. For example, the entire combo can be dropped; a detailed description of this optimization is described in Section 5. Note that such a case may rarely happen in practice. If a method is declared in an interface, its behaviors are often specified by the interface and all its implementations should strictly comply to this specification. Even if an implementation may have a unique way to fulfill the specification, this uniqueness should not be exploited by the client.

5 Optimizations

Naively profiling all container objects is expensive. This section describes three optimizations to reduce the replacement overhead. We modify Jikes Research Virtual Machine (RVM) to replace each allocation site of the form `java.util.X` $x =$ new `java.util.X` (...) with a new allocation site of the form `coco.util.X` $x =$ new `coco.util.X` (..., `null`), where `X` is a container class in Table 1. The additional argument `null` is used to notify the constructor that a new `XCombo` object needs to be created (e.g., line 8 in Figure 2 (a)). Note that such program modification can also be done by bytecode rewriting (e.g., through the load-time instrumentation framework `java.lang.instrument`). Implementing it inside a JVM eliminates the need to perform a separate program transformation phase.

5.1 Dropping Combos

A large part of the overhead comes from the method call forwarding and dispatch. For example, for each element addition/retrieval, there are two additional calls made (e.g., `LinkedList.get` calls `ListCombo.get`, which then calls `ArrayList.get$CoCo`). To reduce this overhead, we maintain a counter in each `XCombo` object, which is incremented every time method `doProfiling` (e.g., line 21 in Figure 2 (b)) is executed but no switch occurs. It is reset to zero if the combo decides to perform a switch. When this counter exceeds a pre-set threshold value, we no longer profile the operations—the currently active container appears to be suitable for the execution.

At this point, there are two situations that might occur. First, no replacement has ever been performed on the combo. The active container is the original container created by the client. In this case, the combo object notifies the active container to drop the combo (and all inactive containers) by setting the field `combo` to `null`. The container goes back to normal and all operations afterwards will be performed directly on it. In the second case, switches have occurred and the currently active container is not the original (user-chosen) container. In this case, we cannot remove the combo because this current container does not directly communicate with the client. However, we can stop profiling the container operations to reduce overhead. Specifically, method `doProfiling` is no longer invoked for the future container operations.

While dropping combos is an important technique to reduce overhead, it may potentially lead to inappropriate switch decisions. For example, the usage of a container may change dramatically after its combo is dropped and the profiling is disabled, leaving the program with a suboptimal implementation. However, we have not found this problem affects the current implementations of the CoCo combos. In fact, many (long-lived) containers in a real-world program have strictly monotonic behaviors—they are more heavily used in a later stage of the execution (e.g., workload run) than in an earlier stage (e.g., warmup), making CoCo switch containers from an implementation that favors fewer elements/operations to another that favors more elements/operations.

5.2 Sampling

Invoking method `doProfiling` for each container operation can be quite expensive. A sampling-based profiling approach is employed to reduce this run-time cost. In addition, we do not profile a container until the first non-ADD operation is performed. This allows the container to have a start-up phase where it gets populated and stabilized; otherwise, the many ADD operations in the beginning may prevent CoCo from observing its real usage pattern and making appropriate switch decisions.

5.3 Lazy Creation of Inactive Containers

In the example shown in Figure 2, inactive containers are created immediately after the active container is created. However, if the combo never switches the container, it is completely redundant to create, initialize, and garbage collect these inactive containers. To make the implementation more efficient, we employ a lazy approach that does not allocate and initialize inactive containers until the first switch is about to be performed. This approach is sound, because, in any combo, each inactive container must have one single abstraction (that abstracts all existing concrete elements) before the first switch. We do not need to create and maintain this abstraction every time an element is added before a switch occurs.

6 Limitations

While the CoCo methodology is general enough to be applied to a variety of container implementations, it has the following three limitations. First, it improves application running time at the cost of introducing space overhead. While this overhead is relatively small (e.g., the detailed statistics are reported in Section 7) and acceptable for most applications (on machines with large memory space), the technique may not be suitable for optimizing memory-constrained programs.

The second limitation is that this technique does not preserve asymptotic complexity of the user-chosen containers. The containers used by CoCo at run time may have a different worst-case complexity than those intended to be used by the user and may thus perform worse in a later execution state when the CoCo profiling is disabled. However, as discussed earlier in this section, we found that the usage scenarios of most long-lived containers are quite consistent during the execution, which reduces the chance for CoCo to make inappropriate decisions. An approach similar to dynamic feedback [13] may be employed in the future work to run a program in a mixed profiling and production mode—the execution alternates between the profiling run and the production run so that profiling is periodically enabled to collect the most updated information.

Finally, although the framework presented in Section 3 provides soundness guarantee for combo implementations that comply with the CoCo methodology, there is no automated enforcement of this compliance. Library designers have to manually ensure that their combos are well-formed, implementations in each combo do not have conflicting specifications, and the replacement rules are appropriately placed. It is interesting to develop tool support, in the future, that can check the combo well-formedness to help developers write reliable optimizations.

7 Empirical Evaluation

We have evaluated CoCo on both a set of micro-benchmarks and a set of large-scale, real-world applications. All experiments are executed on a quad-core machine with an Intel Xeon E5620 2.40GHz processor, running Linux 2.6.18.

7.1 Micro-benchmarks

We have designed several micro-benchmarks (whose execution is dominated by container operations) in order to gain a deep understanding of what achieves efficiency and what incurs overhead. In fact, many of optimization techniques presented in Section 5 are motivated by our observations on the executions of these micro-benchmarks.

***LinkedList* $\rightarrow$ *ArrayList*.** This simple program creates 10 `LinkedList` objects. For each of them, 1000 elements are added and then 40,000 ADD/GET/REMOVE operations are performed. The original program finishes in 127.1 seconds. Using CoCo, its running time ranges from 35.2 to 38.8 seconds, depending on the threshold value X used to switch the `List`. Here X is the percentage of `GET` among all operations executed on each `LinkedList`. For this program, we have tried six different X (i.e., 0, 10%, ..., 50%) and found that the larger X is, the longer the running time is. `ArrayList` outperforms `LinkedList` in all kinds of operations (not just `get`(i)). Hence, we set X = 0 when we run experiments with large, real-world programs—`LinkedList` is immediately switched to `ArrayList` after the first $1/Y$ operations are performed on it (Y is the sampling rate).

***ArrayList* $\rightarrow$ *HashArrayList*.** We find that this switch is highly beneficial for programs with a large number of `contains` operations. We write a program that creates 100 `ArrayList` objects and populates each of them with 10,000 Integers. For each `ArrayList`, we generate 4,000 random Integers and test if this `List` contains these

Table 2. CoCo run-time statistics. Reported in section (a) and (b) is the run-time information of each program executed without and with CoCo, respectively; each section includes running time (*T*) in seconds, GC time (*GC*) in seconds, and peak memory consumption (*S*) in Megabytes; α in section (b) shows the standard deviation of the collected running times.

Bench	(a) Regular execution			(b) CoCo execution			
	$T_1(s)$	$GC_1(s)$	$S_1(Mb)$	$T_2(s)$	α (s)	$GC_2(s)$	$S_2(Mb)$
bloat	53.8	4.5	91.5	51.4(96%)	1.4 (2.7%)	5.9(131%)	165.2(181%)
chart	17.6	4.9	188.2	16.2(91%)	0.8 (4.9%)	7.4(151%)	228.2(121%)
fop	2.5	0.3	49.2	2.1(84%)	0.06 (2.9%)	0.4(133%)	50.1(102%)
lusearch	4.7	1.4	28.6	3.1(66%)	0.07 (2.2%)	1.4(100%)	28.6(100%)
avrora	23.6	1.5	62.6	20.8(89%)	1.1 (5.3%)	2.1(140%)	66.4(106%)
GeoMean				85.4%	3.4%	129.8%	118.8%

numbers. If an Integer is in the list, it is removed. This usage scenario of `ArrayList` is common in static analysis tools that make heavy use of worklist-based algorithms. The original running time is 149 seconds, while CoCo reduces it to 2.2 seconds (i.e., 74$\times$ speedup).

Set and Map Optimizations. The goal here is to investigate the borderline between `HashSet` (`HashMap`) and `ArraySet` (`ArrayMap`) (i.e., what is the appropriate size threshold under which `ArraySet`/`ArrayMap` would outperform `HashSet`/ `HashMap`). We write a few programs that make heavy use of `Sets` and `Maps`, and use different size threshold values to tune the performance. It appears that, for `Set`, this line is somewhere between 5 and 8—in general, we observed, in this test, that switching from `HashSet` to `ArraySet` when the number of elements it contains is smaller than 5 is always beneficial. For `Map`, this line is lower—clear running time reduction can only be seen when we set this threshold to 2. This may be because `HashMap` operations are less expensive than those of `HashSet`, as `HashSet` maintains an internal `HashMap` and delegates all work to it. Based on these observations, we use 5 and 2 as size threshold values for switching `Map` and `Set` when running large benchmarks.

Note that even if small programs are used to tune the parameters, the usage of containers in these programs is real and each container has a large number of elements. In addition, it is much easier to see the impact of the parameter adjustment on performance in such container-centric programs than real-world programs whose performance can often be influenced by many complicated factors.

7.2 Performance on Large Benchmarks

Our large-scale benchmark set contains five real-world applications: bloat, chart, fop, lusearch, and avrora. These applications are chosen because container bloat has been previously found in them (e.g., reported in [7] and [3]). The sampling rate is 1/50, meaning that the method `doProfiling` is invoked once per 50 container operations. We have tried several different sampling rates and 1/50 appears to lead to the best performance. The generational Immix [14] garbage collector is used for our experiments.

For each program, we run it 10 times with a maximum 1GB heap (each with two iterations) using the large workload. The median steady-state performance and the standard

Table 3. Statistics of containers and their run-time switches

Bench	(a) CoCo containers				(b) #Switches			
	#AL	*#LL*	*#HM*	*#HS*	*LL* $\rightarrow$ *AL*	*AL* $\leftrightarrow$ *HL*	*HM* $\leftrightarrow$ *AM*	*HS* $\leftrightarrow$ *AS*
bloat	3322.4K	1158.0K	67.0K	370.7K	108	76/0	645/81	13860/241
chart	25.7K	0	14.5K	0	0	96/0	204/0	2/0
fop	87.4K	0	93	0	0	10/0	17/0	0/0
lusearch	374	0	259	128	0	0/0	101/0	0/0
avrora	50	414.3K	787	14	0	0/0	22/0	4/0

deviation of the running times (α) are reported in Table 2. In addition to the execution information, section (b) in Table 2 also includes ratios between the numbers in each column of section (b) and those in its corresponding column of section (a). Percentages shown in parentheses are either overheads (if $> 100\%$) or improvements (if $< 100\%$). Overall, CoCo reduces program running time by 14.6%, at the cost of introducing a 18.8% overhead in memory space. For all applications, the running time benefit gained from using suitable algorithms can successfully offset the overhead caused by creating (and garbage collecting) extra objects and making extra calls. It is worth investigating, in the future work, how to further reduce the overhead. For example, using object inlining (that inlines the combo and other inactive container objects) may reduce the number of extra objects created, leading to lower GC overhead and smaller space consumption. Note that our optimization techniques (discussed in Section 5) are effective: without them, the average performance improvement for these 5 applications is 6.34%. Due to space limitations, the detailed performance comparison (with and without these optimizations) is omitted.

Table 3 shows, for each program, the number of instances of each container type (i.e., `ArrayList`, `LinkedList`, `HashMap`, and `HashSet`) that CoCo attempts to optimize (section (a)) and the number of container switches that CoCo actually performs (section (b)). All kinds of switches except `LL` $\rightarrow$ `AL` are bi-directional. Each column for a bi-directional switch `X` $\leftrightarrow$ `Y` reports pairs of numbers a/b: a in each pair is the number of switches from `X` to `Y` and b is the number of switches from `Y` to `X`. Note that switches in both directions have occurred during the execution of bloat. In addition, we find that more than half of `LinkedLists` in bloat continue to be switched to `HashArrayLists` after becoming `ArrayLists`. This observation shows that for many data structure objects, there do not exist single optimal solutions throughout the execution. Optimal implementations change as the execution progresses and, therefore, an online adaptive system is highly necessary for removing container inefficiencies.

Despite the many container objects that CoCo attempts to optimize (e.g., shown in section (a) of Table 2), there is only a small number of them for which optimizations are actually possible. Our combo dropping technique appears to be effective—when no optimization opportunity can be found, the overhead incurred by CoCo is negligible. The current version of CoCo focuses only on Java built-in containers, leading to a fairly limited pool of candidates that can form combos. Larger performance gains may be achieved if the technique can be employed to optimize user-defined, application-specific data structures. Another interesting future direction is to optimize only a selected subset of containers that are highly likely to be inefficiently used. This can be done by focusing

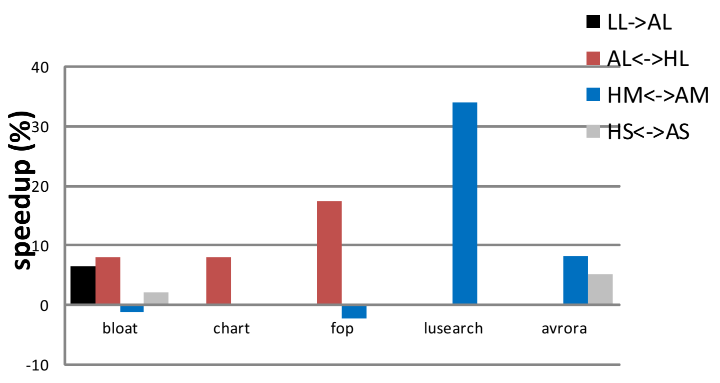

Fig. 6. Breakdown of the benefits from each type of container switch

on allocation sites. We may profile only a few instances (samples) of each allocation site and then use this information as feedback to guide the optimization of other instances created by the same allocation site.

7.3 Breakdown of the Benefits

To further understand the performance improvement that each type of switches contributes, an additional experiment is conducted: each application is run four times and for each time, only one type of switches (from Table 2) is enabled. The breakdown of the benefits is reported in Figure 6. In general, the effectiveness of each kind of replacement is application-specific and depends on the container usage in the application. However, it is clear that the switch from `ArrayList` to `HashArrayList` plays an important role in improving performance. In fact, we find an even larger benefit in fop (17.3%) when all the other types of switches are disabled. This is because `HashArrayList` has a clear algorithmic advantage for performing membership tests (e.g., `contains`) over other lists.

8 Related Work

Container Optimizations. Despite the body of work on container optimizations, CoCo is the first technique that can safely and automatically remove container inefficiencies.

Early work on the SETL framework [15,16,17] and recent work on data representation synthesis [18] attempt to generate appropriate data structure implementations from high-level abstractions at compile time. Our work addresses a different problem, which is to develop a system for Java that can safely switch implementations at run time. Recent attention has been paid to the container bloat problem [7,3,9]. Our previous work [3] proposes a static analysis to identify container inefficiencies. Work that is closest to our proposal is Chameleon [7] and Brainy [9]. Both of them can profile programs to make recommendations on appropriate container implementations that should be used. Recent work from [10] develops memory compaction techniques to reduce the footprint of Java collection objects. Our proposal differs from this category of research in two major aspects. First, the problem that these tools address is orthogonal to the problem that we propose to solve. Both Chameleon and Brainy try to understand precisely what container implementation is suitable for what execution scenario (i.e., *when* to switch implementations), and then identify inconsistency between users' choices and the actual execution scenarios to help developers perform manual tuning. Our work goes far beyond and attempts to develop techniques that can automatically and safely switch implementations (i.e., *how* to switch implementations).

The C# standard library contains a collection class called `HybridDictionary` (`www.dotnetperls.com/hybriddictionary`) that implements a linked list and a hash table, switching over to the second from the first when the number of elements increases past a certain threshold. While this optimization is similar to our `ArrayList` → `HashArrayList` optimization, CoCo is a much more general technique that can be used to optimize a variety of containers.

Software Bloat Analysis. Software bloat analysis [19,4,20,1,21,22,3,2,23,24] attempts to identify and remove performance problems due to run-time inefficiencies in the code execution and the use of memory. Mitchell *et al.* [5] propose a manual approach that detects bloat by structuring behavior according to the flow of information, and their later work [4] introduces a way to find data structures that consume excessive amounts of memory. Our work takes further step in performing online optimization of inappropriately-used data structures. Shankar *et al.* [22] attempt to improve performance by making aggressive method inlining decisions based on the identification of regions that make extensive use of temporary objects. Our previous work [1,2] detects memory bloat by profiling copy chains and copy graphs, and by measuring costs and benefits of object-oriented data structures, respectively. Recent work [25] encodes runtime data structures to identify those that can be reused for improved efficiency.

All these existing techniques detect bloat and provide diagnostic report by profiling semantic information of the program execution. In this paper, we use one type of such information (i.e., container semantics) to find and remove problems automatically. Future work may identify additional semantic bloat patterns that can be exploited to perform similar semantics-aware optimizations.

9 Conclusions and Future Work

This paper proposes an application-level optimization technique, called CoCo, that can safely and adaptively switch container implementations. At the core of this technique is an abstraction-concretization methodology that can be used to create optimizable containers among which container replacement is guaranteed to be safe. While this work focuses on Java containers, the methodology can also be employed to optimize general user-defined data structures. Although the current implementation of CoCo suffers from a number of limitations, this work is the first step towards achieving the goal of automating a range of semantic optimizations for object-oriented applications, and our experimental results already show its promise. Future work may address these limitations and consider to extend the CoCo methodology to optimize languages like Scala, where some data structures seemingly expose their internal implementation through pattern matching. It is also interesting to investigate the possibility of offloading expensive operations to idle cores (in a multicores architecture) to improve the replacement efficiency.

Acknowledgments. We would like to thank Per Larsen, Stefan Brunthaler, Dacong Yan, Brian Demsky, and the anonymous reviewers for their helpful comments.

References

1. Xu, G., Arnold, M., Mitchell, N., Rountev, A., Sevitsky, G.: Go with the flow: Profiling copies to find runtime bloat. In: PLDI, pp. 419–430 (2009)
2. Xu, G., Arnold, M., Mitchell, N., Rountev, A., Schonberg, E., Sevitsky, G.: Finding low-utility data structures. In: PLDI, pp. 174–186 (2010)
3. Xu, G., Rountev, A.: Detecting inefficiently-used containers to avoid bloat. In: PLDI, pp. 160–173 (2010)
4. Mitchell, N., Sevitsky, G.: The causes of bloat, the limits of health. In: OOPSLA, pp. 245–260 (2007)
5. Mitchell, N., Sevitsky, G., Srinivasan, H.: Modeling runtime behavior in framework-based applications. In: Thomas, D. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 429–451. Springer, Heidelberg (2006)
6. Mitchell, N., Schonberg, E., Sevitsky, G.: Four trends leading to Java runtime bloat. IEEE Software 27(1), 56–63 (2010)
7. Shacham, O., Vechev, M., Yahav, E.: Chameleon: Adaptive selection of collections. In: PLDI, pp. 408–418 (2009)
8. Xu, G., Rountev, A.: Precise memory leak detection for Java software using container profiling. In: ICSE, pp. 151–160 (2008)
9. Jung, C., Rus, S., Railing, B.P., Clark, N., Pande, S.: Brainy: effective selection of data structures. In: PLDI, pp. 86–97 (2011)
10. Gil, J., Shimron, Y.: Smaller footprint for java collections. In: Noble, J. (ed.) ECOOP 2012. LNCS, vol. 7313, pp. 356–382. Springer, Heidelberg (2012)
11. Ansel, J., Chan, C., Wong, Y.L., Olszewski, M., Zhao, Q., Edelman, A., Amarasinghe, S.: Petabricks: a language and compiler for algorithmic choice. In: PLDI, pp. 38–49 (2009)
12. Ansel, J., Chan, C., Wong, Y.L., Olszewski, M., Edelman, A., Amarasinghe, S.: Language and compiler support for auto-tuning variable-accuracy algorithms, pp. 85–96 (2011)
13. Diniz, P.C., Rinard, M.C.: Dynamic feedback: an effective technique for adaptive computing. In: PLDI, pp. 71–84 (1997)
14. Blackburn, S.M., McKinley, K.S.: Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. In: PLDI, pp. 22–32 (2008)
15. Schonberg, E., Schwartz, J.T., Sharir, M.: Automatic data structure selection in SETL. In: POPL, pp. 197–210 (1979)
16. Schonberg, E., Schwartz, J.T., Sharir, M.: An automatic technique for selection of data representations in SETL programs. TOPLAS 3, 126–143 (1981)
17. Freudenberger, S.M., Schwartz, J.T.: Experience with the SETL optimizer. TOPLAS 5, 26–45 (1983)
18. Hawkins, P., Aiken, A., Fisher, K., Rinard, M., Sagiv, M.: Data representation synthesis. In: PLDI, pp. 38–49 (2011)
19. Mitchell, N.: The runtime structure of object ownership. In: Thomas, D. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 74–98. Springer, Heidelberg (2006)
20. Dufour, B., Ryder, B.G., Sevitsky, G.: A scalable technique for characterizing the usage of temporaries in framework-intensive Java applications. In: FSE, pp. 59–70 (2008)

21. Mitchell, N., Schonberg, E., Sevitsky, G.: Making sense of large heaps. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 77–97. Springer, Heidelberg (2009)
22. Shankar, A., Arnold, M., Bodik, R.: JOLT: Lightweight dynamic analysis and removal of object churn. In: OOPSLA, pp. 127–142 (2008)
23. Bhattacharya, S., Nanda, M.G., Gopinath, K., Gupta, M.: Reuse, recycle to de-bloat software. In: Mezini, M. (ed.) ECOOP 2011. LNCS, vol. 6813, pp. 408–432. Springer, Heidelberg (2011)
24. Chis, A.E., Mitchell, N., Schonberg, E., Sevitsky, G., O'Sullivan, P., Parsons, T., Murphy, J.: Patterns of memory inefficiency. In: Mezini, M. (ed.) ECOOP 2011. LNCS, vol. 6813, pp. 383–407. Springer, Heidelberg (2011)
25. Xu, G.: Finding reusable data structures. In: OOPSLA, pp. 1017–1034 (2012)