Effects of Static Type Specialization on Java Generic Collections (Technical Report) UMM Working Papers Series Work in progress

Elena Machkasova, Elijah Mayfield, Nathan Dahlberg, J. Kyle Roth

University of Minnesota, Morris

Last updated on: June 4, 2008.

Abstract. Generic types in the Java programming language provide the convenience of writing generic code and perform compilation-time type checking. However, the implementation based on type erasure discards type instance information before run time, complicating dynamic optimizations. We propose a specialization of generic types - a source-to-source transformation that creates specialized copies of a subset of generic classes, replacing their type bounds with specific instance types. Making instance type information available at run time removes unnecessary typecasting and enables the JVM to perform optimizations, such as method inlining. However, interactions of type specialization with the JVM are complex. Specializing more classes may introduce inefficiencies due to extra typechecks, array operations, or JVM warmup. We study trade-offs between effects of type specialization and suggest criteria for selecting groups of classes for performance improvements.
As an example, we compare different ways of specializing subsets of the Java collections library and present an algorithm for specializing such classes. Both the client and the server modes of Sun's HotSpot JVM are studied. We show that generic type specialization enables a program speed up of up-to 20% by specializing only a few classes.

1 Introduction

Generic types in the Java^{TM}programming language are a much-needed software development tool that allow programmers to combine the convenience of generic programming with the advantages of static typechecking. Java generics are implemented using a *type erasure* approach: after performing typechecking, a static compiler replaces the type parameter with its upper bound (either the one specified in the class declaration, or, if none is specified, by `Object`). This approach provides a convenient uniform representation of a generic type, but information about the actual type parameter for a given instance of a generic class is "erased" by the time the program is compiled to bytecode. It is very hard, and in many cases impossible, to dynamically restore information about the actual type parameters during run time. Since Java optimizations are traditionally performed at run time by a Java Virtual Machine (JVM), this type information cannot be used to optimize programs. This means that type erasure generally produces less efficient code than what would be produced from a type-specialized program.

We propose and study a transformation of Java generic types that we call *specialization of generic types*. This optimization creates a copy of a generic type with the type bound replaced by a more specific type, which is based on the actual type parameter. This leads to elimination of unnecessary type casts and provides opportunities for the JVM to better optimize the code, utilizing more precise type information. This optimization is performed as a source-to-source transformation. It is intended for computationally heavy programs that use generic data structures intensively.

The specialization algorithm is presented in section 4, which also discusses correctness of the algorithm and points out the need for an interactive phase. Currently the algorithm is performed by hand. Extending the algorithm to more complex situations and implementation of the algorithm is a part of our future work; see Section 10 for details.

Using the Java Collections library as a source of generic code examples, we compared variations of the optimization that specialize different subsets of classes. We observed that it is possible to specialize only a subset of a program's generic class hierarchy and still get substantial benefits (up to 20%). Thus this approach does not cause as much code duplication as a heterogeneous translation, such as C++ templates, would.

Our studies show that a certain group of specializations that we call *interface* specializations provides maximal efficiency with limited code duplication. In an example of an *interface* specialization using the `ArrayList` hierarchy, only six types out of eleven are specialized. As a result, we obtain the same efficiency as for a completely specialized program (see Section 6.2 for an overview of tests and the summary of results).

Different choices of subsets of classes to be specialized result in very different program behavior patterns, due to intricate interactions between the type systems and the dynamic optimizations performed by the JVM. The program behavior also greatly depends on JVM implementation. In our tests we use Sun's HotSpot^{TM} JVM and study both the *client* and the *server* mode. In Section 9 we discuss factors that contribute to program efficiency for different subsets of specialized classes in both these modes of the JVM.

The specialization would be the most beneficial for computationally heavy programs. It would allow programmers to take advantage of generic types and not pay a high efficiency price due to the overhead of type erasure. Additionally, this work may be of interest to programming language designers and JVM engineers, as it discusses trade-offs between implementation of parameterized types in dynamically interpreted languages and points out concrete inefficiencies of implementations of Java generics, some of which may be approached by adjusting JVM implementations. Software developers can also benefit from this study, because it suggests how inefficiencies due to type erasure may be mitigated by alternative design decisions.

2 Related work

Earlier proposals [6] and [3] laid out principles of implementing generics in Java. Specific implementation proposals include, among others, GJ (e.g. [7]) which proposed the homogeneous (i.e type erasure) approach later adopted in Java, NextGen (e.g. [1]) which proposed a combination of a heterogeneous (i.e. a separate code copy for each type instantiation) and a homogeneous approach, and [15] with an approach based on Java core reflection which requires an extension to the JVM. Unlike C# generic types [13], generic types in Java were added without modifying the existing JVM. This leads to inability to provide full support for operations on type variables, such as typecasting or array creation.

While recent research on static optimization of Java is not as widespread as that of dynamic optimization, there are several works that discuss static transformations of Java code. [17] proposes a specialization based on partial evaluation as a way of eliminating overhead of generic programming (not specifically generic types). [5] proposes a static inlining optimization of Java program at the level of bytecode, but does not specifically deal with generic types either. [18] proposes a source code transformation that facilitates load-time handling of generic types using Java core reflection with the goal of full support of operations on type parameters of generic types. A recent work [16] presents a compiler that accomplishes such integration of generic type parameters within a standard JVM. However, in both papers the focus is on full integration, and not on efficiency.

Our work is related to that of [1] in that it proposes a heterogeneous translation. However, it is unique in a sense that it explores a possibility of a selective compile-time specialization of Java generic types in the framework of the current Java generics implementation with the goal of increasing efficiency of the program. While it adopts the heterogeneous approach which may lead to code duplication, the transformation is intended to be performed only when it is beneficial, and thus creates copies only of those classes and methods that are promising for efficiency increase.

3 Overview of Specialization

A declaration of a generic class or a method specifies *bounds* for each type parameter. For instance,

```
public class Sort<T extends Comparable<T>> {...}
```

specifies `Comparable<T>` as the bound for the type parameter T. This means that every valid instantiation of T must be a subtype of `Comparable<X>`. If no bound is specified then the bound is assumed to be `Object`.

Generic types in Java are implemented using *type erasure*: after checking type-correctness of the generic code, the compiler replaces all occurrences of a type parameter within its scope by its type bound. This creates a uniform representation of each generic class or method. Type casts are inserted automatically so that at run time an attempt to assign an object of a wrong instance type to a variable results in a `ClassCastException`. The generated bytecode instructions are not specific to the type parameters[1], so the type information is said to be *erased*.

There are two main sources of overhead due to type erasure: unnecessary type casts and the inability of the JVM to perform type-specific optimizations based on the actual type parameter. As an example, consider the following code:

```
ArrayList<String> l = new ArrayList<String>();
.....
String s = l.get(0);
```

Since `ArrayList` is implemented as a "type-erased" copy, the type returned from `get` is `Object`, and the compiler inserts the cast to `String`, making the assignment equivalent to

```
String s = (String) l.get(0);
```

which requires a typecheck. An implementation of `ArrayList` that is specific to strings would return a `String` from `get` and thus would not require a typecheck.

Type erasure may also introduce an overhead of dynamic method lookup. For instance, if `ArrayList<String>` is passed to a method that takes a parameter of the type `ArrayList<T>`, where `T` is bound by `Object`, the call `l.get(0).toString()` would produce a non-trivial method lookup. Since `get` returns an Object, the target of the call to the method `toString` is unknown and requires a dynamic resolution.

Providing more precise type information opens opportunities for method call optimization. For instance, in the above example if a dynamic compiler can deduce that a method call is on a string, there is only one target for the call. Then the compiler may devirtualize the call, replacing it with a direct call to `String`'s `toString` method, or even inline the method (e.g. [12]).

Method call optimization is also possible when a call is made on a variable of an interface type. In this case a compiler first may perform a *virtualization* of an interface call, i.e. replace `invokeinterface` bytecode instruction by `invokevirtual` ([2]), and then devirtualize or inline the call.

4 Specialization Algorithm

The essence of our proposed transformation which we call *type bound specialization* is to create a copy of a generic type with the type bound replaced by the actual instance of the type used in the program. For instance, if a class `ArrayList` is declared as

```
public class ArrayList<E>
```

[1] Some information about type parameters is preserved in method signatures, but not in bytecode instructions themselves

then its bound is `Object`. If there is an object of the type `ArrayList<Integer>` used in the program then the transformation creates a specialized copy of `ArrayList` with an `Integer` bound:

```
public class ArrayListInteger<E extends Integer>
```

All references to `ArrayList<Integer>` are then replaced by the specialized class `ArrayListInteger<Integer>`.

Different versions of the transformation may specialize different subsets of the type hierarchy. In the remainder of this section we present an algorithm for optimizing a program given a "target" type - the highest point in a program's type hierarchy that we would like to specialize. We discuss correctness of the transformation. In Section 5 we compare efficiency of specializations with different target classes using classes from the the Java Collections library as an example.

4.1 General applicability

Type specialization is a whole-program transformation: it requires the entire source of the program is available. Since type names are modified in the process of specialization and references to old classes are replaced by references to their specialized copies, dynamic class reloading or reflection of specialized classes or classes that reference them may have undesired effects.

Both of these requirements, however, may be relaxed if it is known that only a subset of a program uses types that are going to be specialized so the source code of the remaining classes is not required and they may be dynamically reloaded. If only a small portion of the hierarchy will be specialized, fewer classes are subject to these restrictions. We also note that the optimization is the most beneficial for computationally heavy non-interactive programs which are unlikely to use dynamic loading or reflection.

4.2 Complete and Partial Specializations

A straightforward specialization algorithm would specialize every parameterized type in a program to all type instances that the type or its subtypes are used with. We call such a specialization *complete*. However, this would result in a substantial code duplication and may be undesirable for the reasons explained in Section 4.1. Additionally it may interfere with code encapsulation since it requires specializing a large number of interfaces and is likely to affect the client code. It may also introduce efficiency problems due to additional typecasting and possible lengthening of JVM warmup time (see section 9 for details). Therefore we would like to limit the number of classes affected by the specialization while still gaining efficiency.

Java allows a subtype to have a more specific bound than that of its supertype, opening a possibility of *partial* specializations, i.e. those that specialize only the subset of a program's type hierarchy that relates to modified classes. For instance, consider the following type declaration

```
class ArrayList<E> extends AbstractList<E>
                       implements List<E>
```

(the bound of both `AbstractList` and `List` is `Object`). It is possible to create a version of `ArrayList` specialized to `Integer` without specializing the other two types[2]:

```
class ArrayListInteger<E extends Integer>
        extends AbstractList<E> implements List<E>
```

We show that specializing only a small number of types may be just as efficient as a complete specialization.

4.3 Notations and Assumptions

Java has a single class inheritance, but allows a type to be a subtype of multiple interfaces. Thus a type hierarchy forms a lattice structure.

We assume that all variables referring to parameterized types, including formal method parameters, are properly parameterized upon declaration. For instance a variable for an `ArrayList` of `Integer`s is declared as `ArrayList<Integer>`, not as `ArrayList`.

For simplicity we do not specialize *nested* generic type instances, i.e. type parameters that are in turn generic types, such as `List<List<String>>`. While the algorithm may replace the inner list to a specialized version `ListString`, the original non-specialized version of `List` would be used for the outer list, resulting in `List<ListString<String>>`. Instances with nested type parameters are not likely to be heavy in method calls on the type parameter, and thus are not promising candidates for specialization in any case.

We also assume that classes that we attempt to specialize do not appear as bounds in other generic types. For instance, `T extends Comparable<T>` is a valid bound in our framework, but `T extends ArrayList<T>` is not.

By a *program* we mean the source code for all classes and interfaces used in a given program. We assume that there is no dynamic loading (neither reloading of classes nor reflection). As we pointed out in Section 4.1, these restrictions may be relaxed in practice.

We use the following notations:

- T for any object (i.e. non-primitive) type.
- G for a name of a generic type.
- B for a *type bound* of a generic type. For instance, in the declaration `class ArrayList<E>` the bound type is `Object`. In most cases the type bound is a *concrete type*, i.e. a type that does not have any type variables. Another common bound in our example is `Comparable<T>`. By the assumption above types that we attempt to specialize do not appear as bounds of other types.

[2] This approach does not work directly if the bound of `List` is `Comparable<T>` since `Integer` is a not a valid substitute for G in this case. However, it still would be possible to specialize the subclass by directly replacing all occurrences of G within the class by `Integer`

- X for an *instance type* that may be a concrete type, a type that contains type variables, or a type variable from an enclosing class or method scope.
- $<:$ is used as a *subtype* notation: $T_1 <: T_2$ means that T_1 is a subtype of T_2. $T <: T$ holds for any T.

A scope of a class (or an interface) type variable is the entire declaration of the class, excluding sections where the variable name is overshadowed by nested declarations. Such declarations may be those of generic methods (a scope of a method's type variable is the entire method) or of inner classes. For more details see Section 8 of [10].

By a *statically-bound type* of a variable or a method parameter or a constructor we mean the type derived for it by the static compiler with the type parameters replaced by their type bounds in the current scope. For instance, a statically-bound type of a variable declared as `List<T>` or of a method parameter `List<T>`, where `T` is in the scope that defines it as an `Object`, is `List<Object>`. The statically-bound type of the form $G\langle B\rangle$ (such as `ArrayList<Integer>`) is that type itself.

The algorithm uses *type/bound pairs* to record types that are candidates for specialization and bounds that they would be specialized to. For instance, a pair $\langle$`ArrayList,Integer`$\rangle$ indicates that the class `ArrayList` would be specialized to the bound `Integer`. The definition below uses such pairs.

Definition 1. *Given a program P and two types T_1, T_2 in P such that $T_2 <: T_1$, a downward closure of T_1 to T_2 in P, denoted $D_P(T_1, T_2)$, is a set of all types T such that $T <: T_1$ and $T_2 <: T$.*

Given a program P and a generic type G, a downward closure of G, denoted $D_P(G)$, is the union of all sets of pairs $\langle \widetilde{G}, B\rangle$ such that $\widetilde{G} \in D_P(G, G_i)$, where $G_i <: G$ such that there is a call to the constructor of a statically-bound type of the type $G_i\langle B\rangle$ in P.

We omit the subscript P when it is unimportant and just write $D(T_1, T_2)$ and $D(G)$, respectively.

The definition allows us to find all descendants of a given generic type G used in a program and all the bounds they are used with. If a descendant of G is not used in the program, there is no need to specialize it. Since by our assumption there is no reflection in the part of the type hierarchy relevant to the generic types under consideration, the only way to create an instance of a type is via a constructor.

As an example, consider the interface `List`. If in a program its descendant `ArrayList` is used with `Integer` and `String` actual type parameters then the downward closure includes 6 pairs: `List`, `AbstractList`, `ArrayList`, each paired with `Integer` and `String`. The abstract class `AbstractList` is between `List` and `ArrayList` in the type hierarchy.

We do not specialize generic methods, i.e. methods that declare their own type parameter: a type of a generic method is determined via type inference which complicates the algorithm. Studying effects of specializing individual generic methods is a part of our future work. Our approach can handle wildcards whose bounds are type parameters in the specialized classes, but other uses of wildcards are not yet supported. A full treatment of wildcards is future work (see Section 10).

For simplicity we assume that each generic type has only one type parameter. The algorithm can be easily generalized to a type with multiple parameters.

4.4 Specialization Algorithm.

Our presentation of the algorithm is somewhat informal. There are points in the algorithm where multiple decisions are possible. These points may require a user's intervention. In section 4.5 we discuss possible user choices and their implications.

The algorithm is given a program P and a target generic class or an interface G. It is also possible to specify multiple independent (i.e one not a subset of another) targets; the algorithm then specializes the program one target at a time. We assume that the given program successfully compiles (possibly with warnings).

The algorithm determines the minimal set of types that need to be specialized in order to specialize the given *target type* to the bounds used in the program. The target is a parameter for the algorithm. For instance, for the *quicksort* example we get different results when we specify `ArrayList` and `List` as target bounds. See Section 6.1 for details.

The algorithm checks whether the specialization guarantees to preserve the behavior of the program. If some changes may potentially alter behavior, the algorithm would proceed only if these changes are approved by the user, otherwise it would stop and output *failure* (see Section 4.5 for details). If the analysis part is successful (possibly with some user-approved changes), the algorithm produces specialized copies of generic types, replaces references to non-specialized types by those to their specialized copies, and performs other necessary changes.

The algorithm takes a program P and a target generic type G_0 as an input. Let B_0 be the bound of G_0. The algorithm performs the following steps:

1. **Downward closure.** Construct $\mathcal{T} = D(G_0)$ according to Definition 1.
2. **Propagation.** Let $\langle G, B\rangle \in \mathcal{T}$ and let $\widetilde{G}$ be a generic class or an interface for which at least one of the following holds:
   - $\widetilde{G}$ contains an instance (i.e. a non-static) variable of the statically-bound type $G\langle B\rangle$[3].
   - $\widetilde{G}$ has a method with a local variable of the statically-bound type $G\langle B\rangle$.
   - $\widetilde{G}$ has a method m such that in the program there is a call to m that takes an actual parameter of the statically-bound type $G\langle B\rangle$ or returns an object of the statically-bound type $G\langle B\rangle$, where a statically-bound type of the return value is determined in the context where the method is called.

[3] The variable may also appear as an array type or a type of a parameter to another generic type.

Then $\mathcal{T} = \mathcal{T} \cup \{\langle \widehat{G}, B\rangle \mid \langle \widehat{G}, B\rangle \in D(\widetilde{G})\}$, i.e we add all elements of the downward closure of $\widetilde{G}$ paired up with the bound B.

Note that G may be positioned anywhere in a type hierarchy, relative to classes already included in $\mathcal{T}$. In particular, it may be below such classes, above such classes, be an inner class of one of such classes, or be completely unrelated to any of them.

Repeat step 2 until no more types can be added to $\mathcal{T}$.

3. **Correctness Analysis.** The procedure for determining whether an optimization preserves program behavior and for keeping track of user choices is given at the end of Section 4.5. If the procedure returns *failure*, the optimization is halted. Otherwise the three lists returned by the procedure are passed to step 4.
4. **Specializing the program.** The code transformation proceeds as follows:
   (a) For each pair $\langle G, B\rangle$ in $\mathcal{T}$:
      i. create a copy of G with the name that is a concatenation of the names of the type itself and of the bound (we refer to it as G_B); replace the bound of G by B. Change the names of constructors accordingly.
      ii. Change the names of types that G_B inherits from to their copies specialized to B (if these types have been specialized).
      iii. for each method m in G if there is a call to m anywhere in the program with a formal parameter or a return value whose statically-bound type is $G\langle B\rangle$, replace this method's parameter type by $G_B\langle B\rangle$. Note that the calls may originate in the class G itself.
      iv. If there are any static variables, ambiguous operations on raw types, or any ambiguous references to other specialized types, follow decisions made by the user in phase 3.
   (b) In the rest of the program:
      i. replace every call to a constructor of a non-specialized type $G\langle B\rangle$ by a call to the corresponding constructor of $G_B\langle B\rangle$.
      ii. In each scope where $\langle G, B\rangle \in \mathcal{T}$ and the statically-bound type of X is B, replace each variable and each method parameter of type $G\langle X\rangle$ by the type $G_B\langle B\rangle$.

Section 6.1 shows the work of the algorithm for two specializations of `ArrayList` type hierarchy.

4.5 Correctness Issues and User's Choices

The algorithm given in Section 4.4 detects cases when automatic changes to the program are insufficient to specialize generic classes. In such cases the user needs to know the intent of the program's designer in order to chose the right specialization strategy. Different user decisions in these cases may lead to different program behavior. In most cases the difference would manifest itself only in rare circumstances.

In this section we give a list of conditions when this may happen. More items may be added to the list as our research covers more features (such as generic methods or wildcards).

Features that may cause a behavior difference are:

1. Static variables are shared by all instances of a class. Specializing classes separates their static data into multiple classes so it is no longer shared. For instance, a class may have a static variable `serialVersionUID` for serialization. Specializing the class duplicates the variable. This is not a problem for our examples since the program does not use serialization. However, in general it is impossible for the algorithm to guess the purpose of each static variable so the user needs to decide whether the specialization should proceed.
2. Sometimes type erasure implementation forces use of raw (erased) types instead of a type parameter. Two examples illustrate the issue:
   - A collection with a formal type parameter T cannot use an array of type $T[]$ to store the data it contains. Instead, the array is declared to be of the bound type, for instance `Object`. Specializing this code would require a user input, since an algorithm cannot tell whether an array is meant to be of type T or of `Object`.
     This problem can manifest itself in two ways. An array may be declared as type T. Alternatively, an array of the bound type may be used. If the array is not declared as type T, it is changed to be so. Once the array variable is of type T, a typecast is necessary in the array assignment:
     ```
     this.elementData = (T[]) new
                            Object[initialCapacity];
     ```
     Here `elementData` is of type $T[]$. Since an array of type T (a type variable) cannot be created directly, the declaration of the array must then be changed to the new bound type. Additional typecasts to the new bound type may need to be inserted for assignments to the array.
   - Use of explicit typechecking or typecasting in methods such as `equals` creates similar problems. Consider the following method in `AbstractList` (this is an abstract class that implements interface `List`):
     ```
     public boolean equals(Object o) {...
         if (! (o instanceof List))
           return false;
     ...}
     ```
     If `List` is specialized as both `ListInteger` and `ListString` then instances of `AbstractList` that are specialized to `Integer` and `String` in the propagation phase are no longer in the same class, and are not instances of the non-specialized type `List`. A program that relies on a particular result of `equals` when comparing objects in different instantiations of `List` may change its behavior. It is up to the user to decide whether the `List` in the specialized `AbstractList` should be left as it is or replaced by a specialized type. A similar issue arises with bounded wildcards whose bound is the same as the bound of a generic type that contains them.

3. If a program uses both a specialized and a non-specialized copy of a class G and a method in another class $\widetilde{G}$ takes an object of type G as a parameter, then the question arises whether the method should take an object of the specialized or non-specialized class.
   Note that if there are calls to this method then often the type of the parameter would be determined during the propagation phase and $\widetilde{G}$ itself will be specialized. If there are no calls (either direct or via dynamic method dispatch) to the method then any decision would do since no actual calls are affected.
   The problematic situation arises when the actual parameter types for the method are obscured by using variable types higher in the type hierarchy (for instance, the specialization target is `ArrayList` but the method call is via a `List` interface parameter. In this case the user may decide, based on the knowledge of the program's design, to do one of the following:
   – specialize the entire class $\widetilde{G}$,
   – create specialized copies of the method itself (currently our optimization has not encountered cases like this, but it is an option for the future),
   – decide to halt the specialization altogether (and possibly try for a higher target).

Based on the above considerations, the *correctness check* procedure will return one of the following:

– a token *failure* when the user decides not to optimize.
– A list *ArrayTypes* which contains the list of arrays whose type needs to be changed from $B_{old}[]$ to $B_{new}[]$, where B_{old} is the original bound of G, as described in part 2 in the above discussion; a list *TypeCasts* that lists extra type casts that need to be inserted because of the array type changes; and a list *Types* that lists user-chosen ways of resolving type ambiguities described in the second case of part 2 and in part 3.

5 Applications of the Algorithm to Java Collections

The Java Collections library provides a variety of parameterized classes. From an implementation standpoint, these classes present intricate interconnections via inheritance, interfaces, and inner classes. This makes specialization quite challenging but instructive. We focus on specializing some frequently used classes in the library: `ArrayList`, `PriorityQueue`, and `TreeMap`.

5.1 Test Data and Naming Conventions

Our examples use two user-defined classes, `TestInteger` and `TestString` which implement the `Comparable` interface as data that is stored in the collections. These classes are similar to `Integer` and `String`, with the primary difference that each of them includes a static counter for counting the number of times `compareTo` is called. Keeping track of the number of calls to `compareTo` is helpful for some of our tests and also makes the method `compareTo` less prone to being eliminated by unrelated run-time optimizations.

Names of specialized copies of classes are formed by concatenating the the name of the new bound (sometimes abbreviated) at the end of the class name. For instance, our copy of `ArrayList` which is specialized to `TestInteger` would be named `ArrayListInteger`.

5.2 Partial Specializations: Classification and Notations.

The following conventions are used in the rest of the paper to denote different kinds of partial type specializations:

- Original non-specialized programs are denoted by O.
- S stands for program with specialized “client” classes, i.e. classes that use a given collection; the collection class itself is unspecialized.
- For referring to optimizations that specialize collections classes we use an abbreviation of the highest type that is specialized. For instance, AL stands for the transformation that specializes `ArrayList` class. If the subset has more than one highest type, we include all highest types in the abbreviation. For instance, the TME specialization in the `TreeMap` example specializes unrelated `TreeMap` and `Entry` classes.
- Some specializations combine the two previous kinds: they specialize a part of the collections hierarchy and also the class that uses the collections. In these cases we add S to the specialization name.
  Some specializations of collection classes require the “client” class to be specialized as well. In this case we add S at the end of the name to stress that the class that uses the collections is specialized, although there is no counterpart of such a specialization without an S.
- A complete optimization, i.e. the one that specializes every parameterized class, is denoted by C.

Our examples follow a traditional software design practice of accessing data structures via interfaces in the “client” code. For instance, `ArrayList` is accessed via its interface `List`. A specialization that uses such an interface as a target type in the algorithm is called an *interface* specialization. Since the propagation step in the algorithm requires that the “client” class is specialized as well, the interface specialization specialization for `ArrayList` is denoted by LS and not by L. Other interface specializations considered here are QS for `PriorityQueue` and MS for `TreeMap`.

5.3 Testing Methodology and Run-time Environment.

We use the HotSpot^{TM}JVM in both the *client* and the *server* modes. The client mode is intended for shorter programs; it minimizes the startup time and memory footprint, but performs a smaller set of optimizations. The server mode is intended for generating fast optimized code, but requires more time and memory to optimize the program.

We observed significant differences in code behavior in the two modes. For some versions of the specialization, the client JVM produced a slowdown of an optimized program, compared to the original one, while the server JVM showed improvement. In our work we mainly focus on optimizing for the server mode, since it generally gives better total running times for long programs. However, we also would like to at least avoid a slowdown in the client mode. We show that certain versions of the specialization lead to a speed up in both modes.

The HotSpot JVM compiles frequently called methods to native code. Method selection is based on profiling data collected during the run. Initially all methods start executing in an interpreted mode. Once the number of calls to a method reaches a certain *threshold*, the method is compiled. The client mode of the JVM by default uses 500 calls as a threshold while in the server mode the default is 10,000 calls. The time that a JVM takes before it compiles a method to native code is called *warmup* [11]. Since the time spent on dynamic compilation is included in a program running time and is different for different specializations, we measure total running times, including the JVM warmup, for fair comparison between specializations.

We ran all tests under Fedora Core 7 on AMD Athlon^{TM}64-bit 2GHz processor. Both the compiler and the HotSpot JVM are version 1.6.0_04. We ran each test in the client and the server modes 20 times each. We recorded the total time the program spent in the user mode using the system *time* command and plotted the results using a statistical package R [14].

5.4 Notations and code sample conventions.

In hierarchy diagrams we use UML notations. When a class is shown as "Name1.Name2", this represents Name2 as an inner class of Name1. Each object lists in parentheses which specializations it is changed in. On the diagrams and in examples in the rest of the paper we ignore non-generic interfaces that some collections implement, such as `RandomAccess, Serializable`, and `Cloneable`.

In the code samples we use our own copies of the `ArrayList` hierarchy. To make sure that we do not accidentally use a library copy instead of our own, we prefix names in this hierarchy with `R`, such as `RArrayList` instead of `ArrayList`. Our tests sorted both `TestInteger`s and `TestString`s. Consequently, the classes in the `ArrayList` hierarchy had two specialized copies - one for `TestInteger` and one for `TestString`. For simplicity in code samples below we show only types specialized to `TestInteger`; `TestString` specialization is completely analogous.

6 A Quicksort Example Using `ArrayList`

We used `ArrayList` as data storage for a quicksort program. Since sorting is a common component of many real-life programs and quicksort is a widely-used implementation of sorting, this example emulates a typical use of generics in real-life programs. Type hierarchy of `ArrayList` is represented in Figure 1 (see section 5.4 for notations).

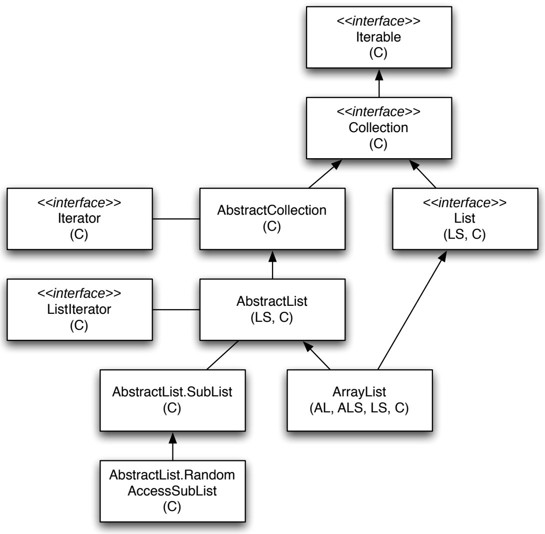

Fig. 1. ArrayList Class Hierarchy

In the quicksort example, we call our test code via a main class, Measure-Runtimes. This class creates the test data and a client `Quicksort` class we are using. The program takes, as input, a number of arrays to create, a number of elements to create and place into those arrays, and a number of times to sort this data. We construct the given number of `ArrayLists` and fill these with randomly generated `TestIntegers` and `TestStrings`, using a constant seed for repeatability. We then call the sorting method of the quicksort class on this randomly generated data.

```
public class MeasureRuntimes {
     private static int ARRAY_SIZE, NUM_LOOPS, NUM_SORTS;

     private static Random randomFactory;
private static RList<RList<TestInteger>> testIntegerCases =
                    new RArrayList<RList<TestInteger>>();

public static void main(String[] args) {
//      [...]
    time1 = ((double) System.currentTimeMillis()) / 1000.0;
    insertItems(randomFactory);
    sortCases();
    time2 = ((double) System.currentTimeMillis()) / 1000.0;
    System.out.println(time2 - time1);
//      [...]
private static void insertItems(Random r) {
    for (int i = 0; i < NUM_LOOPS; i++) {
        RList<TestInteger> integers = new RArrayList<TestInteger>();
        testIntegerCases.add(integers);
    }

    for (int j = 0; j < ARRAY_SIZE; j++) {
        int rand = r.nextInt();
        TestInteger tInt = new TestInteger(rand);
        for (int i = 0; i < NUM_LOOPS; i++) {
            testIntegerCases.get(i).add(tInt);
        }
    }
}

private static void sortCases() {
    Quicksort<TestInteger> intSorter = new Quicksort<TestInteger>();
    for (int s = 0; s < NUM_SORTS; s++) {
        for (int i = 0; i < testIntegerCases.size(); i++) {
            intSorter.sort(testIntegerCases.get(i));
        }
    }
}

//      [...]
}
```

The quicksort example program we use contains a `sort` method which takes in an interface `List<T>`. It sorts the data contained in this list according to the quicksort algorithm as defined in [8]. These methods include calls to methods such as `set`, `get`, and `compareTo`.

```
public class Quicksort<T extends Comparable<T>> {

    public void sort(RList<T> inputList) {

//      [...]

    }

//      [...]

}
```

Our first optimization is labelled S. This changes only the `Quicksort` class, changing the type bound to the following:

```
public class QuicksortInteger<T extends TestInteger> {

private static void sortCases() {
       QuicksortInteger<TestInteger> intSorter =
           new QuicksortInteger<TestInteger>();
//      [...]
}
```

This change alters the type parameter T throughout the class to be more specific. The performance improvement comes from devirtualization of `compareTo` which is enabled since the exact target of the method is now known.

Our optimization AL changes the ArrayList class, from the old type bound:

```
public class RArrayList<E> extends RAbstractList<E>
implements RList<E> {
```

to a newer, more specific bound:

```
public class RArrayListInteger<E extends TestInteger>
extends RAbstractList<E> implements RList<E> {
```

Once again, methods can call the class directly and retain type information, such as in the following method:

```
private static void insertItems(Random r) {
     for (int i = 0; i < NUM_LOOPS; i++) {
         RList<TestInteger> integers =
             new RArrayListInteger<TestInteger>();
//      [...]
      }
//      [...]
}
```

Our third optimization is referred to as ALS, and combines both the changes from AL and from S, specializing the two classes.

Our fourth optimization, LS, combines the changes from ALS with an optimized interface List. This changes the following code:

```
public class RArrayList<E> extends RAbstractList<E>
implements RList<E> {
public abstract class RAbstractList<E> extends RAbstractCollection<E>
implements RList<E> {
public interface RList<E> extends RCollection<E> {
```

to

```
public class RArrayListInteger<E extends TestInteger> extends
RAbstractListInteger<E> implements RListInteger<E> {
public abstract class RAbstractListInteger<E extends TestInteger>
extends RAbstractCollection<E> implements RListInteger<E> {
public interface RListInteger<E extends TestInteger>
extends RCollection<E> {
```

This change also affects many methods, most of which pass data as the interface class. For instance, the following sections of the `MeasureRuntimes` class are altered:

```
private static RList<RList<TestInteger>> testIntegerCases =
new RArrayList<RList<TestInteger>>();

 private static void insertItems(Random r) {
        RList<TestInteger> integers = new RArrayList<TestInteger>();
 }

    private static void sortCases() {
        Quicksort<TestInteger> intSorter = new Quicksort<TestInteger>();
    }
```

to:

```
private static RList<RListInteger<TestInteger>> testIntegerCases =
new RArrayList<RListInteger<TestInteger>>();

    private static void insertItems(Random r) {
            RListInteger<TestInteger> integers =
new RArrayListInteger<TestInteger>();
    }

    private static void sortCases() {
           QuicksortInteger<TestInteger> intSorter =
new QuicksortInteger<TestInteger>();
    }
}
```

The final optimization that we consider is a complete optimization, C. In addition to the LS changes, the following classes were changed:

```
public class RArrayList<E> extends RAbstractList<E>
    implements RList<E> {
public abstract class RAbstractList<E>
    extends RAbstractCollection<E> implements RList<E> {
public abstract class RAbstractCollection<E>
```

```
    implements RCollection<E> {
public interface RCollection<E> extends
    research.lang.RIterable<E> {
public interface RIterable<T> {
public interface RList<E> extends RCollection<E> {
public interface RListIterator<E> extends RIterator<E> {
public interface RIterator<E> {
```

These classes appear, in the complete optimization, as follows:

```
public class RArrayListInteger<E extends TestInteger> extends
    RAbstractListInteger<E> implements RListInteger<E> {
public abstract class RAbstractListInteger<E extends TestInteger>
    extends RAbstractCollectionInteger<E> implements RListInteger<E> {
public abstract class RAbstractCollectionInteger<E extends TestInteger>
    implements RCollectionInteger<E> {
public interface RListInteger<E extends TestInteger> extends
    RCollectionInteger<E> {
public interface RCollectionInteger<E extends TestInteger>
    extends research.lang.RIterableInteger<E> {
public interface RIterableInteger<T extends TestInteger> {
public interface RListIteratorInteger<E extends TestInteger>
    extends RIteratorInteger<E> {
public interface RIteratorInteger<E extends TestInteger> {
```

We also studied the effects of passing parameters to methods in different forms. Specifically, we examine the difference between passing a parameter of the actual type ArrayList to a method with the formal parameter of an interface type `List` and passing it to a method with a formal parameter `ArrayList`. We call the latter mechanism a `direct parameter passing` since the parameter does not change its type in the process. The code is altered such that whenever a List would be used in the original example, an ArrayList is used instead. This caused significant differences in performance. The following fragment from the measurement class shows the differences:

```
private static RArrayList<RArrayList<TestInteger>> al_IntegerCases =
    new RArrayList<RArrayList<TestInteger>>();

    private static RArrayList<RArrayList<TestString>> al_StringCases =
    new RArrayList<RArrayList<TestString>>();

    public static void main(String[] args) {
            time1 = ((double) System.currentTimeMillis()) / 1000.0;
            insertItemsAL(randomFactory);
            sortCasesAL();
            time2 = ((double) System.currentTimeMillis()) / 1000.0;
            System.out.println(time2 - time1);
    }

    private static void insertItemsAL(Random r) {
        for (int i = 0; i < NUM_LOOPS; i++) {
            al_IntegerCases.add(new RArrayList<TestInteger>());
        }

        for (int j = 0; j < ARRAY_SIZE; j++) {
            int rand = r.nextInt();
            TestInteger tInt = new TestInteger(rand);
            for (int i = 0; i < NUM_LOOPS; i++) {
                al_IntegerCases.get(i).add(tInt);
            }
        }
    }

    private static void sortCasesAL() {
        Quicksort<TestInteger> intSorter = new Quicksort<TestInteger>();
        Quicksort<TestString> stringSorter = new Quicksort<TestString>();
        for (int s = 0; s < NUM_SORTS; s++) {
            for (int i = 0; i < al_IntegerCases.size(); i++) {
                intSorter.sortAL(al_IntegerCases.get(i));
            }
            for (int i = 0; i < al_StringCases.size(); i++) {
                stringSorter.sortAL(al_StringCases.get(i));
            }
        }
    }

    public void sortAL(RArrayList<T> inputList) {
        quicksortAL(inputList, 0, inputList.size() - 1);
    }
```

6.1 Applications of the Algorithm to Quicksort Example

In this section we consider our original set of examples with a list parameter passed as via an interface formal parameter, not via direct passing. The work of the algorithm with `ArrayList` as the target (AL case) is as follows:

1. $\mathcal{T} = \{$<`ArrayList, TestInteger`>, <`ArrayList, TestString`>$\}$. The downward closure only contains `ArrayList` since no other type extends it.
2. Propagation: no generic classes reference `ArrayList`.
3. Decisions that the user made: ignore the static variable `serialVersionUID`. The list *ArrayTypes* returned from the correctness-checking procedure contains `Object[]` arrays created in `ArrayList` constructors and several other methods. These arrays are assigned to the variable of the type `E[]`, where `E` is

the type parameter. Thus in the specialized classes they need to be changed to the new bound type: `TestInteger[]` or `TestString[]` respectively. There are no additional typecasts or type ambiguities in this case.

4. Create the specialized classes `ArrayListInteger` and `ArrayListString` with the bounds set to `TestInteger` and `TestString`, respectively. In the main class change the calls to the constructors accordingly. In the new specialized classes perform the array type changes as specified in *ArrayTypes*.

The work of the algorithm with `List` as the target (LS case) is as follows:

1. $\mathcal{T}$ = {<`ArrayList, TestInteger`>, <`ArrayList, TestString`>, <`AbstractList, TestInteger`>, <`AbstractList, TestString`>, <`List, TestInteger`>, <`List, TestString`>}. `ArrayList` class is included based on calls to the constructor and `AbstractList` is on the path from `List` to `ArrayList`.
2. Add to $\mathcal{T}$ classes with methods that take `AbstractList` as a parameter: `SubList`, `RandomAccessSubList`; classes that take `List` as a method parameter: `Quicksort`. All of these classes are added with both `TestInteger` and `TestString` bounds.
3. The same issues as for AL specialization. Additionally there is an ambiguity in constructors of the two sublist classes that take `AbstractList` as a parameter, and there is no call in the program to determine the correct type; we decided to specialize the parameter.
4. The same changes to `ArrayList` as in AL specialization; analogous changes to `AbstractList, List, Quicksort`, and to the two sublists; references in the main class are changed to the corresponding specialized classes, e.g. `ListInteger<TestInteger>` and `QuicksortInteger<TestInteger>`.

6.2 Tests and Results

The results of the test runs are presented in Figures 2 (server mode) and 3 (client mode) and are summarized in the Table 1. Thick black lines on the graphs denote the median, quartiles are marked by the thin lines, and the circles denote outliers.

In server mode, LS and C have the best performance, improving over O by 10-15%. C is slightly slower than LS because of the JVM warmup of constructors. The main sources of their speed-up are the static factors described in Section 9 and the fact that a dynamic check of a parameter for `set` method is eliminated because the parameter is passed as `TestInteger`.

As it is easy to notice, AL specialization causes a very significant slowdown (by 25%). The reason for this is that the parameter (`ArrayListInteger` object) is passed to methods `sort`, etc. of the non-specialized `Quicksort` class as a non-specialized `List`. Therefore `compareTo` is called via `invokeinterface`. Moreover, passing a `TestInteger` to `set` method prevents elimination of dynamic method lookup since the `set` method of `List` takes any `Comparable`, and finding the

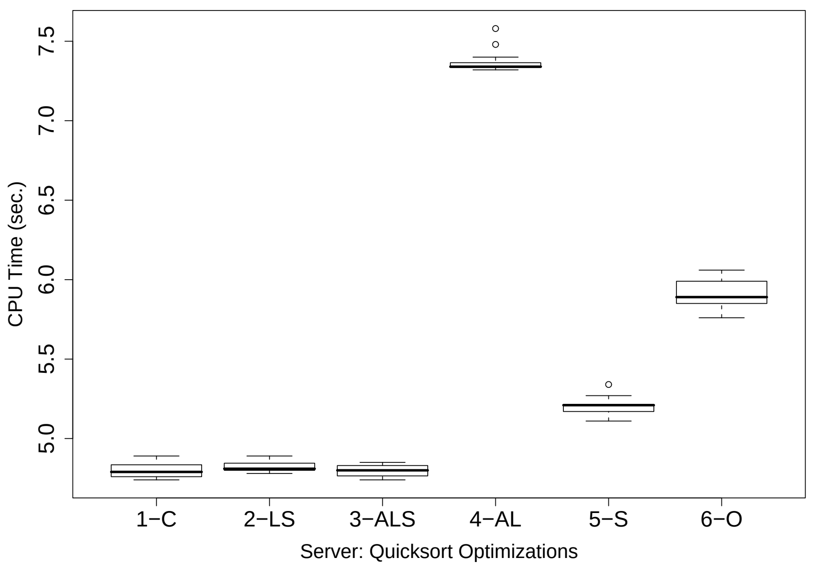

Fig. 2. Results of Quicksort specializations, server mode

right method requires a type check. Iterating over just `set` or `add` methods (without any other calls) produces a similar slowdown for AL, whereas iterating over just `get` (which has no object parameters) does not. ALS performs better than AL because of static devirtualization of `compareTo` and the fact that the type parameter to `set` is known to be a `TestInteger`. In fact, in server mode `set` in ALS performs significantly better than in any of the other specializations. See section 9.1 for details.

S has advantages of static devirtualization of `compareTo` and of passing a parameter to `set` known to be a `TestInteger`. However, it does not have static

Name	Server Time (sec)	Server % Speedup	Client Time (sec)	Client % Speedup
O	5.91	-	12.49	-
S	5.19	12.13	11.53	7.65
AL	7.38	-24.96	14.59	-16.80
ALS	4.80	18.83	11.63	6.90
LS	4.83	18.29	11.15	10.71
C	4.80	18.83	11.16	10.68

Table 1. Quicksort Results; percentage time decrease relative to Original

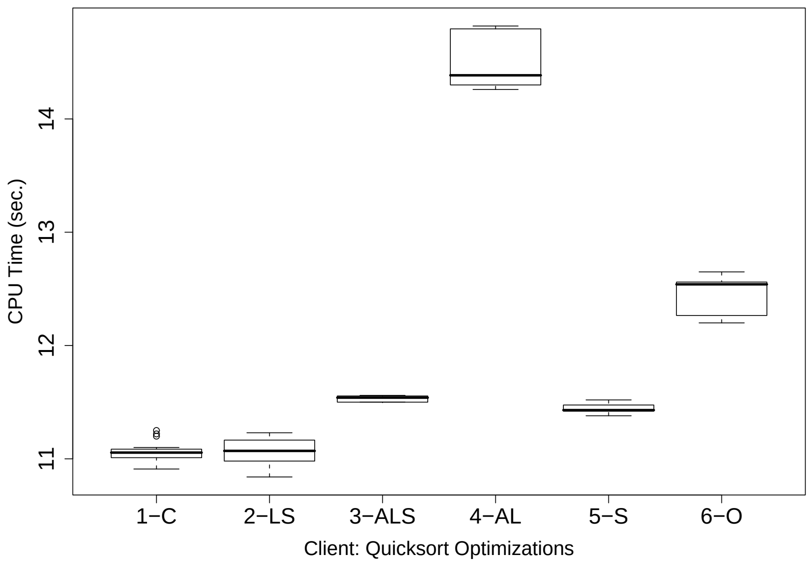

Fig. 3. Results of Quicksort specializations, client mode

typecasting elimination for the result of `get`, and thus performs worse than C and LS.

In client mode, LS and C still have the best performance, improving over O by close to 11%. AL causes a slowdown similar to that in the server mode for the same reasons. Both S and ALS give a modest improvement close to 7%.

The results for the case of a **direct parameter passing** are summarized in table 2 and figures 4 and 5. They show a performance improvement in all specializations. In the server mode, ALS, LS, and C give a drastic improvement of approximately 22%. S does not have advantage of dynamic optimizations within the `ArrayList` class, such as inlining, resulting in a moderate 6% time

Name	Server Time (sec)	Server % Speedup	Client Time (sec)	Client % Speedup
O	5.74	-	10.45	-
S	5.37	6.54	8.94	14.42
ALS	4.46	22.36	8.61	17.60
LS	4.48	21.97	8.64	17.36
C	4.45	22.44	8.70	16.77

Table 2. Quicksort Results with Direct Parameter Passing; percentage time decrease relative to Original

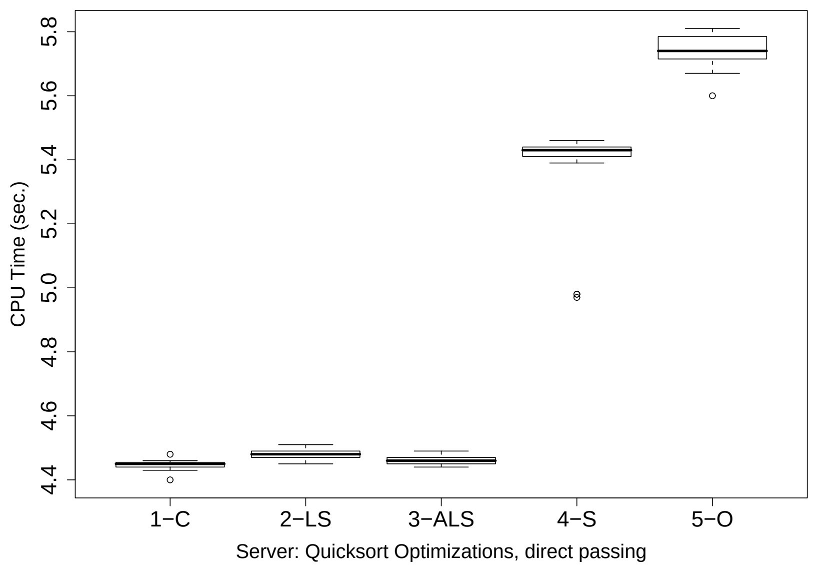

Fig. 4. Results of Quicksort specializations with direct parameter passing, server mode

decrease. Similar results are seen in the client mode: ALS, LS, and C all improve over O by approximately 17% and the S specialization improves performance by 14%.

7 Optimizations of the PriorityQueue Example

We now present a specialization example utilizing `PriorityQueue`. The type hierarchy for the `PriorityQueue` class is shown in Figure 6.

We tested the PriorityQueue classes by passing in three parameters: the number of queues to create, the number of elements to place in each queue, and the number of times to iterate over these queues. We created copies of queues parameterized to both `TestInteger` and `TestString`:

```
public class MeasureRuntimes {

private static RList<RQueue<TestInteger>> testIntegerCases
                    = new RArrayList<RQueue<TestInteger>>();
private static RList<RQueue<TestString>> testStringCases
                    = new RArrayList<RQueue<TestString>>();
```

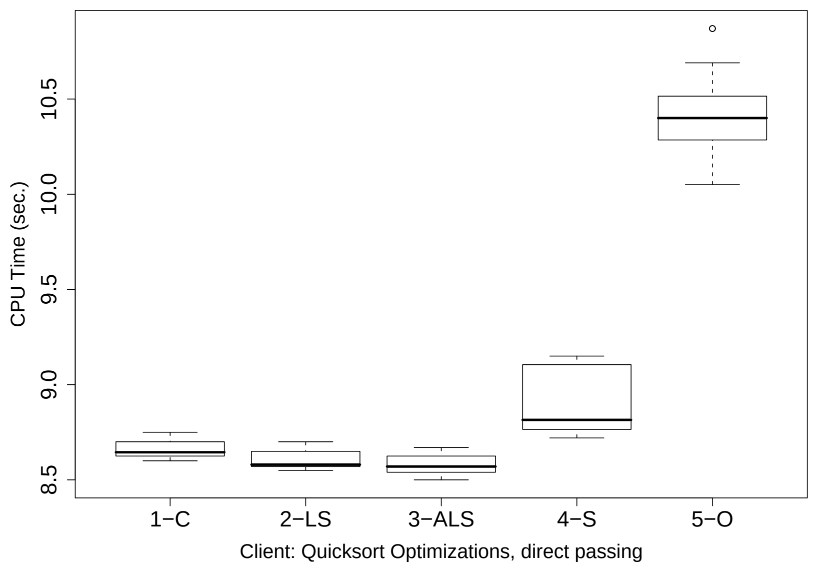

Fig. 5. Results of Quicksort specializations with direct parameter passing, client mode

```
private static TestInteger[][] intContent;
private static TestString[][] stringContent;

private static int numLoops;
private static int numQueues;
private static int queueSize;

public static void testQueues(RList<RQueue<TestInteger>> intQueue,
       RList<RQueue<TestString>> strQueue, int numLoops){
  createContentArrays();
    QueueTester<TestInteger> intTester = new QueueTester<TestInteger>();
    QueueTester<TestString> strTester = new QueueTester<TestString>();
    for (int s = 0; s < numLoops; s++) {
        resetQueues();
        for(int i = 0; i < intQueue.size(); i += 2){
            intTester.compareQueues(intQueue.get(i), intQueue.get(i+1));
        }
        for(int i = 0; i < strQueue.size(); i += 2){
            strTester.compareQueues(strQueue.get(i), strQueue.get(i+1));
        }
    }
```

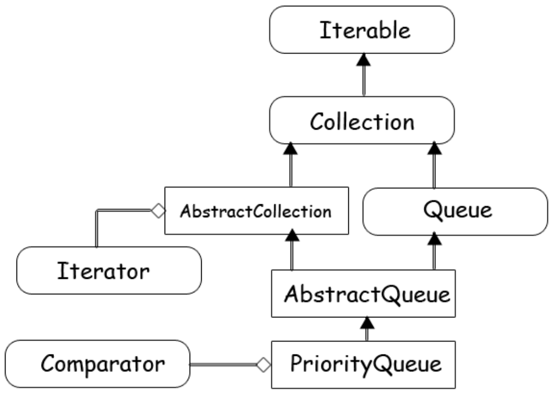

Fig. 6. PriorityQueue Class Hierarchy

```
}

public static void createContentArrays(){
    intContent = new TestInteger[2][queueSize];
    stringContent = new TestString[2][queueSize];
    for(int i = 0; i < queueSize; i++){
        int rand = r.nextInt();
        intContent[i%2][i] = new TestInteger(rand);
        stringContent[i%2][i] = new TestString("" + rand);
        rand = r.nextInt();
        intContent[(i+1)%2][i] = new TestInteger(rand);
        stringContent[(i+1)%2][i] = new TestString("" + rand);
    }

public static void resetQueues(){
    testIntegerCases.clear();
//      [...]
    int rand;
    for(int i = 0; i < numQueues/2; i++){
        RQueue<TestInteger> intQueueZero = new RPriorityQueue<TestInteger>();
        RQueue<TestString> strQueueZero = new RPriorityQueue<TestString>();
        for(int j = 0; j < queueSize; j++){
            intQueueZero.add(intContent[0][j]);
            strQueueZero.add(stringContent[0][j]);
//      [...]
        }
//      [...]
        testIntegerCases.add(intQueueZero);
        testStringCases.add(strQueueZero);
    }
}
}
```

PriorityQueue in the original form stores data in an Object[]:

```
private transient Object[] queue;
```

When using queues, this class takes queues two at a time and compares the top element of each. It then removes the element with the higher value. This is repeated until one of the queues becomes empty:

```
public RQueue<T> compareQueues(RQueue<T> first, RQueue<T> second){
    while(!first.isEmpty() && !second.isEmpty()){
        if(first.peek().compareTo(second.peek()) > 0){
            firstCounter++;
            first.remove();
        }else {
            secondCounter++;
            second.remove();
        }
    }
    return (firstCounter > secondCounter) ? first : second ;
}
```

This example uses the PriorityQueue methods `add` for inserting elements and `isEmpty`, `peek`, and `remove` in the client class, along with calling `compareTo` on the elements in each queue.

There are several other classes that are also important for the PriorityQueue example:

```
public abstract class RAbstractCollection<E> implements RCollection<E>
public abstract class RAbstractQueue<E> extends RAbstractCollection<E>
    implements RQueue<E>
public interface RCollection<E> extends RIterable<E>
public interface RComparator<T>
public interface RIterable<T>
public interface RIterator<E>
public interface RQueue<E> extends RCollection<E>
public interface RSet<E> extends RCollection<E>
public interface RSortedSet<E> extends RSet<E>
```

We compare four different optimizations of this example given below. All the optimizations (except O) involve changing the storage array type from `Object[]` to `E[]`, as described at the end of Section 4.5. The unchanged optimization, **O**, is identical to the code described above.

The **PQ** optimization changes the PriorityQueue class only:

```
public class RPriorityQueueInteger<E extends TestInteger>
    extends RAbstractQueue<E>
private transient E[] queue;

public static void resetQueues(){
    for(int i = 0; i < numQueues/2; i++){
        RQueue<TestInteger> intQueueZero = new
            RPriorityQueueInteger<TestInteger>();
        RQueue<TestInteger> intQueueOne = new
            RPriorityQueueInteger<TestInteger>();
    }
}
```

The **QS** specialization is an interface specialization with a target interface `Queue`. The client class is specialized as well. The following changes are made:

```
    public class QueueTesterInteger<T extends TestInteger> {
    public RQueueInteger<T> compareQueues(RQueueInteger<T> first,
        RQueueInteger<T> second)
    public abstract class RAbstractQueueInteger<E extends TestInteger>
    extends RAbstractCollection<E> implements RQueueInteger<E>
    public class RPriorityQueueInteger<E extends TestInteger> extends
        RAbstractQueueInteger<E>
    public interface RQueueInteger<E extends TestInteger>
        extends RCollection<E>

public class MeasureRuntimes {
    private static RList<RQueueInteger<TestInteger>> testIntegerCases = new
        RArrayList<RQueueInteger<TestInteger>>();
    private static RList<RQueueString<TestString>> testStringCases = new
        RArrayList<RQueueString<TestString>>();

    public static void testQueues(RList<RQueueInteger<TestInteger>> intQueue,
        RList<RQueueString<TestString>> strQueue, int numLoops){
     createContentArrays();
     QueueTesterInteger<TestInteger> intTester = new
         QueueTesterInteger<TestInteger>();
        QueueTesterString<TestString> strTester = new
            QueueTesterString<TestString>();
//      [...]
    }

    public static void resetQueues(){
//      [...]
        for(int i = 0; i < numQueues/2; i++){
            RQueueInteger<TestInteger> intQueueZero = new
```

```
                RPriorityQueueInteger<TestInteger>();
            RQueueInteger<TestInteger> intQueueOne = new
                RPriorityQueueInteger<TestInteger>();
        }
//      [...]
    }
//      [...]
}
```

The complete specialization, **C**, makes the same changes to the testing and client classes as in **QS** and additionally specializes the following Collections classes:

```
public abstract class RAbstractCollectionInteger<E extends TestInteger>
    implements RCollectionInteger<E>
public abstract class RAbstractQueueInteger<E extends TestInteger>
    extends RAbstractCollectionInteger<E> implements RQueueInteger<E>
public interface RCollectionInteger<E extends TestInteger> extends
    RIterableInteger<E>
public interface RComparatorInteger<T extends TestInteger>
public interface RIterableInteger<T extends TestInteger>
public interface RIteratorInteger<E extends TestInteger>
public class RPriorityQueueInteger<E extends TestInteger> extends
    RAbstractQueueInteger<E>
public interface RQueueInteger<E extends TestInteger>
    extends RCollectionInteger<E>
public interface RSetInteger<E extends TestInteger>
    extends RCollectionInteger<E>
public interface RSortedSetInteger<E extends TestInteger>
    extends RSetInteger<E>
```

The results are given in Figures 7 and 8 and summarized in Table 3.

Name	Server Time (sec)	Server % Speedup	Client Time (sec)	Client % Speedup
O	13.31	-	18.69	-
PQ	11.17	16.04	15.03	19.53
QS	10.87	18.3	14.38	23.05
C	10.88	18.27	14.40	22.92

Table 3. PriorityQueue Testing

In both modes all three optimizations benefit from typecasting elimination and devirtualization of `compareTo` within the `PriorityQueue` class. PQ performs extra typechecking in the non-specialized client class which makes it slightly

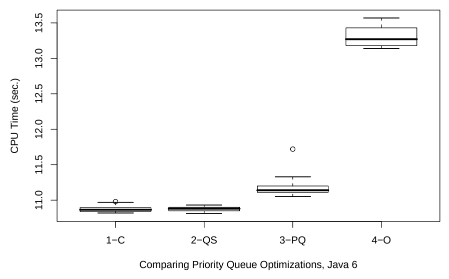

Fig. 7. Results of PriorityQueue specializations, server mode

slower than PQS and QS. QS and C show nearly identical results which shows that an interface optimization may be sufficient to achieve the full potential of specialization without a large code duplication. A very slight slowdown of C compared to QS may be due to a JVM warmup, but the time differences are not significant enough to make conclusions.

8 Optimizations of the TreeMap Example

In this section we present a specialization example using `TreeMap`. The type hierarchy for the `TreeMap` class is shown in Figure 9. `TreeMap` stores its elements as a red-black tree. The `TreeMap` class is parameterized to `<K, V>`, where an inner class `Entry` stores keys of type K and values of type V:

```
public class RTreeMap<K,V> extends RAbstractMap<K,V>
implements RSortedMap<K,V>
```

`Entry` is parameterized (independently of `TreeMap`) to `<K, V>`:

```
static class REntry<K,V> implements RMap.REntry<K,V>
```

Elements are inserted into `TreeMap` through the `put` method. The method rebalances the tree if needed by calling `K.compareTo` and several private methods.

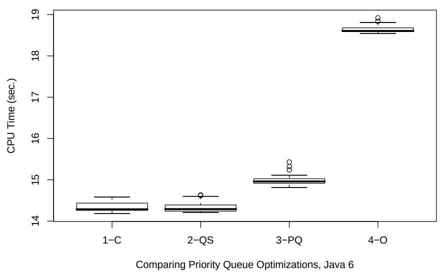

Fig. 8. Results PriorityQueue specializations, client mode

As in the previous examples, we used a variety of classes and methods in the testing code and observed a variety of different effects of the optimizations.

We pass into the test program parameters listing the number of trees, the number of elements in each tree, and the number of times to iterate over the trees when testing performance. The times we measure are the time it takes to insert these values into trees and then retrieve those values repeatedly:

```
public class MeasureRuntimes {

private static int numLoops, numMaps, mapSize;
private static TestInteger[] keys;
private static TestString[] values;
private static List<RMap<TestInteger, TestString>> maps;
private static TreeReader<TestInteger, TestString> mapReader;
//      [...]
public static void main(String[] args) {
//      [...]
    mapReader = new TreeReader<TestInteger, TestString>();
    createContents();
//      [...]
    time1 = ((double) System.currentTimeMillis() / 1000.0);
```

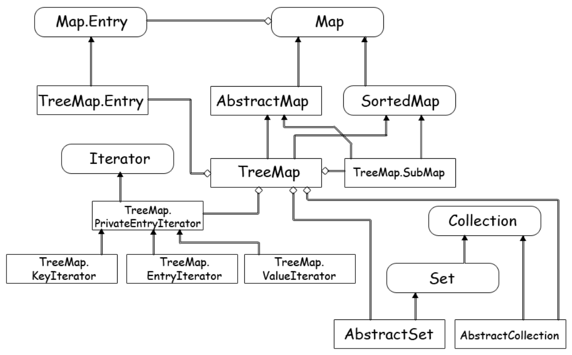

Fig. 9. TreeMap Class Hierarchy

```
    testFull();
    time2 = ((double) System.currentTimeMillis() / 1000.0);
    System.out.println(time2-time1);
//      [...]
}
```

We create the set number of keys, values, and trees before beginning timing. We use `TestIntegers` as keys and `TestStrings` as values:

```
public static void createContents(){
    keys = new TestInteger[mapSize];
    for(int i = 0; i < mapSize; i++){
        keys[i] = new TestInteger(i);
    }
    values = new TestString[mapSize];
    for(int i = 0; i < mapSize; i++){
        int val = r.nextInt();
        values[i] = new TestString(val + "");
    }
    maps = new ArrayList<RMap<TestInteger, TestString>>();
    for(int i = 0; i < numMaps; i++){
        maps.add(new RTreeMap());
    }
}
```

We time how long it takes to fill all trees and then iterate over them:

```
public static void testFull(){
    fillMaps();
    for(int i = 0; i < numLoops; i++){
        for(int j = 0; j < numMaps; j++){
            mapReader.chartMap(maps.get(j));
        }
    }
}

public static void fillMaps(){
    for(int i = 0; i < numMaps; i++){
        for(int j = 0; j < mapSize; j++){
            maps.get(i).put(keys[j], values[((j*7919)/(i+5))%mapSize]);
        }
    }
}
```

The test code uses `Set` and `Iterator` classes to iterate over keys. The method `get` is used to obtain values from a `TreeMap`, and `compareTo` method is called on the values. The client class is originally parameterized as `<K, V extends Comparable<V>>`:

```
public class TreeReader<K, V extends Comparable<V>> {

public static int sailors;
//      [...]

public Comparable<V> chartMap(RMap<K, V> map){
    sailors++;
    RSet<K> keys = map.keySet();
    RIterator<K> keyIterator = keys.iterator();
    V buriedTreasure =  map.get(keyIterator.next());
    while(keyIterator.hasNext()){
        V value = map.get(keyIterator.next());
        if(value.compareTo(buriedTreasure) > 0){
            buriedTreasure = value;
        }
    }
    return buriedTreasure;
}
}
```

Since the class hierarchy of `TreeMap` includes many different classes, a complete specialization of it would be quite complex so it was not performed. There are numerous possibilities for combining different optimizations; we selected a subset of them. We tested several optimizations that specialize subsets of this hierarchy. In total, we tested seven different versions of the program. The original (referred to as O) uses all code identical to that listed above. There are several other important classes as well:

```
public abstract class RAbstractMap<K,V> implements RMap<K,V>
public interface RSortedMap<K,V> extends RMap<K,V>
public interface RMap<K,V>
    interface REntry<K,V>
public interface RSet<E> extends RCollection<E>
public abstract class RAbstractSet<E> extends RAbstractCollection<E>
    implements RSet<E>
```

In our first optimization, S, we only change the `TreeReader` class, changing the type bound to `TestInteger` for keys and `TestString` for values:

```
public class TreeReaderIS<K extends TestInteger, V extends TestString
private static TreeReader<TestInteger, TestString> mapReader;
```

For the TMS optimization, we change the type bound in both the `TreeReader` and `TreeMap` classes, as can be seen in the following samples:

```
public class TreeReaderIS<K extends TestInteger, V extends TestString>
public class RTreeMapIS<K extends TestInteger,V extends TestString>
extends RAbstractMap<K,V> implements RSortedMap<K,V>
private static TreeReaderIS<TestInteger, TestString> mapReader;
    public static void createContents(){
//      [...]
        for(int i = 0; i < numMaps; i++){
            maps.add(new RTreeMapIS());
        }
//      [...]
    }
```

Changing from O to TMES requires the same changes to `TreeReader` and TreeMap as TMS. Addtionally, we modify the class `REntry` to `TestInteger` and `TestString` bounds:

```
public class TreeReaderIS<K extends TestInteger, V extends TestString>
public class RTreeMapIS<K extends TestInteger,V extends TestString>
extends RAbstractMap<K,V> implements RSortedMap<K,V>
private static TreeReaderIS<TestInteger, TestString> mapReader;
    public static void createContents(){
//      [...]
        for(int i = 0; i < numMaps; i++){
            maps.add(new RTreeMapIS());
        }
//      [...]
    }
static class REntry<K extends TestInteger,V extends TestString>
implements RMap.REntry<K,V> {
```

Optimizing more of the hierarchy than TMES, SMS specializes the interface `RSortedMapIS` as well as the abstract class `RAbstractMapIS`. This is in addition to the rest of the optimization that TMES does.

```
public class TreeReaderIS<K extends TestInteger, V extends TestString>
public class RTreeMapIS<K extends TestInteger,V extends TestString>
extends RAbstractMapIS<K,V> implements RSortedMapIS<K,V>
public interface RSortedMapIS<K extends TestInteger,V extends TestString>
extends RMap<K,V>
public abstract class RAbstractMapIS<K extends TestInteger,V extends TestString>
implements RMap<K,V>
private static TreeReaderIS<TestInteger, TestString> mapReader;
    public static void createContents(){
//      [...]
        for(int i = 0; i < numMaps; i++){
            maps.add(new RTreeMapIS());
        }
//      [...]
    }
static class REntry<K extends TestInteger,V extends TestString>
implements RMap.REntry<K,V> {
```

MS optimizes the `TreeReader` class, much like S does. However, it also modifies the `MeasureRuntimes` class to use `RMapIS`, which has been optimized to the `TestInteger` and `TestString` bounds.

```
public class TreeReaderIS<K extends TestInteger, V extends TestString> {
   public int chartMap(RMapIS<K, V> map){
//      [...]
   }
}

public class MeasureRuntimes {
    private static List<RMapIS<TestInteger, TestString>> maps;
        mapReader = new TreeReaderIS<TestInteger, TestString>();

    public static void createContents(){
       maps = new ArrayList<RMapIS<TestInteger, TestString>>();
       for(int i = 0; i < numMaps; i++){
            maps.add(new RTreeMapIS());
       }
    }
}

public class RTreeMapIS<K extends TestInteger,V extends TestString>
extends RAbstractMapIS<K,V> implements RSortedMapIS<K,V>
```

```
    static class REntry<K extends TestInteger,V extends TestString>
implements RMapIS.REntry<K,V> {
public abstract class RAbstractMapIS<K extends TestInteger,V
extends TestString> implements RMapIS<K,V>
public interface RSortedMapIS<K extends TestInteger,V extends TestString>
extends RMapIS<K,V>
public interface RMapIS<K extends TestInteger,V extends TestString>
    interface REntry<K extends TestInteger,V extends TestString> {
```

The final optimization MSetS takes everything MS has optimized as well as optimizing the interface RSetInteger and abstract class RAbstractSetInteger.

```
public class TreeReaderIS<K extends TestInteger, V extends TestString> {
   public int chartMap(RMapIS<K, V> map){
//      [...]
   }
}

public class MeasureRuntimes {
    private static List<RMapIS<TestInteger, TestString>> maps;
        mapReader = new TreeReaderIS<TestInteger, TestString>();

    public static void createContents(){
       maps = new ArrayList<RMapIS<TestInteger, TestString>>();
       for(int i = 0; i < numMaps; i++){
            maps.add(new RTreeMapIS());
       }
    }
}

public class RTreeMapIS<K extends TestInteger,V extends TestString>
extends RAbstractMapIS<K,V> implements RSortedMapIS<K,V>
    public RSetInteger<K> keySet() {
    static class REntry<K extends TestInteger,V extends TestString>
implements RMapIS.REntry<K,V> {
public abstract class RAbstractMapIS<K extends TestInteger,V
extends TestString> implements RMapIS<K,V>
public interface RSortedMapIS<K extends TestInteger,V
extends TestString> extends RMapIS<K,V>
public interface RMapIS<K extends TestInteger,V extends TestString>
    interface REntry<K extends TestInteger,V extends TestString>
public interface RSetInteger<E extends TestInteger> extends RCollection<E>
public abstract class RAbstractSetInteger<E extends TestInteger>
extends RAbstractCollection<E> implements RSetInteger<E>
```

– **O** - original, non-optimized program. All bounds (for both `K` and `V`) are `Object`. The `TestTM` class is bound by `<K, V extends Comparable<V>>`.

– **S** specializes only the client class `TestTM`.
– **TMS** specializes `TreeMap` and `TestTM`.
– **TMES** specializes `TreeMap, TreeMap.Entry,` and `TestTM`.
– **SMS** specializes `TestTM, TreeMap, SortedMap`, and `AbstractMap`.
– **MS** specializes `TestTM, TreeMap, AbstractMap, SortedMap`, and also `Map, Map.Entry`, and `TreeMap.Entry`.
– **MSetS** specializes the same classes as MS and additionally the `Set` class is specialized to `TestInteger`.

Both MS and MSetS are considered interface specializations. SMS also specializes an interface, but it is not considered an interface specialization since the interface is not the one used in the client class to access `TreeMap`.

The results of `TreeMap` specializations are shown in Figures 10 (server) and 11 (client) and summarized in Table 4.

By specializing the key type to `<K extends TestInteger>`, we enable devirtualization of the call to `TestInteger.compareTo`. However, at least two problems exist with this specialization. The call to `compareTo` in `TreeMap` typecasts the keys to `Comparable` before calling `compareTo`. The overhead of the virtual method call is optimized away by the server mode but not in the client mode. Additionally, many methods in `TreeMap` (such as `containsKey`, `containsValue`, and others) pass keys and values as `Object`, typecasting them back into a `K` or `V`. While this is a trivial cast in the non-specialized version of `TreeMap`, in a specialized the cast to `K` or `V` becomes non-trivial. Thus programs heavy in calls to `put` and similar methods benefit less from the specialization. As we observe, the results of a test with a lot of calls `get` produce speed-ups of up to 20% for optimizations that specialize the `TreeMap.Entry` class. We also observed that some optimizations have a wide range of running times in server mode. One possible explanation is memory allocation variations.

Name	Server Time (sec)	Server % Speedup	Client Time (sec)	Client % Speedup
O	6.64	-	10.88	-
S	6.55	1.39	10.98	-0.88
TMS	6.70	-0.87	9.29	17.18
TMES	5.76	15.44	8.58	26.86
SMS	6.75	-1.59	9.34	16.53
MS	5.89	12.83	8.55	27.29
MSetS	5.83	14.03	8.61	26.35

Table 4. TreeMap Testing

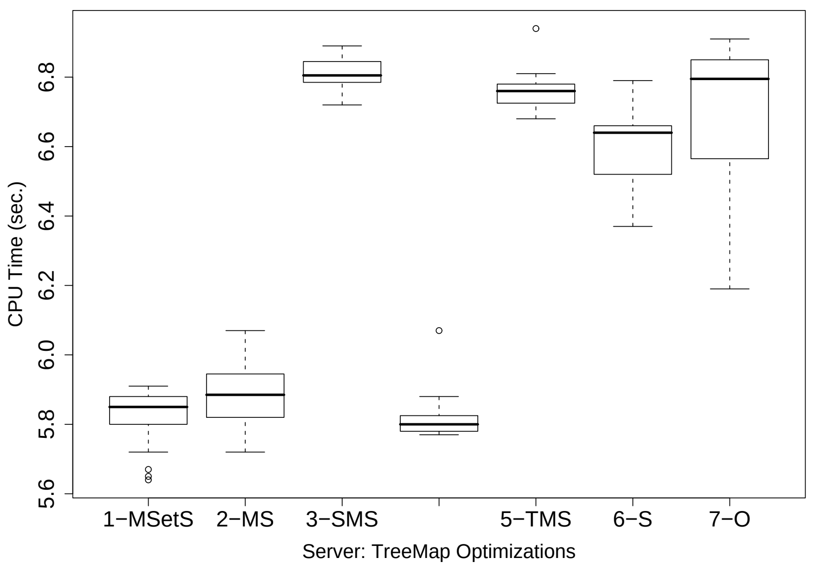

Fig. 10. Results of TreeMap specializations, server mode

9 Effects of Specialization

9.1 Testing Individual Methods.

An important goal of our research is to understand how various elements of the generic type hierarchy are affected by each kind of a specialization. In order to study such effects we created a simplified version of a program similar to the quicksort program.

The program tests the same five versions of the specialization (where the sixth version is the original program). In fact, the program uses the same specialized classes in the `ArrayList` hierarchy as the quicksort, does with the only difference that for simplicity for these tests we use only classes that are specialized to `Integer`, not to `String`. Our studies show that, as expected, classes specialized to `String` behave the same as those specialized to `Integer` in terms of relative performance of different versions of the specialization (although the absolute numbers differ). Also we have observed that JVM warmup and other overhead related to class duplication is insignificant and does not differ much for programs with both `Integer` and `String` versions of classes and for those with only `Integer`-specialized hierarchy. Thus limiting our testing to `Integer`-specialized classes does not affect the soundness of conclusions about relative effects of different specializations on methods.

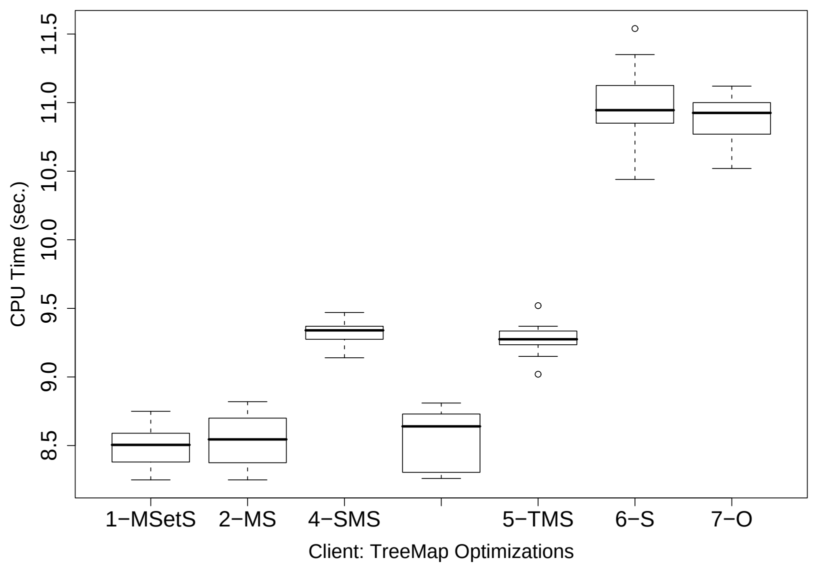

Fig. 11. Results of TreeMap specializations, client mode

The program consists of several test methods, each testing a particular method in the `ArrayList` class by looping over calls to that method. The methods being tested are `set, add`, and `get`. Some of the tests are designed to test effects of typecasting in addition to the effects on the methods. Just like with the quick-sort, we use two kinds of parameter passing: via an interface `List` and directly via `ArrayList` class. We also tested effects of inlining by setting the JVM flag to `-XX:-Inline`.

Since the three methods studied here differ in passing parameters, handling return values, and behavior within the `ArrayList` class, they give us important insights into the effects of the optimization.

9.2 Parameter Passing via an Interface

In this section we discuss the case when the container objects (`ArrayList` or `TreeMap`) are referred to in the client classes of a program via an interface variable. We discuss performance changes caused by each of the five optimizations for the three methods used in our sample runs: `add, get`, and `set`.

Effects on `set`. The `set` method takes an parameter of type E and an index. The method consists of a call to a private method `RangeCheck` and two assignment statements to objects of type E. The call returns a value of type E, although the return value is not used in the test.

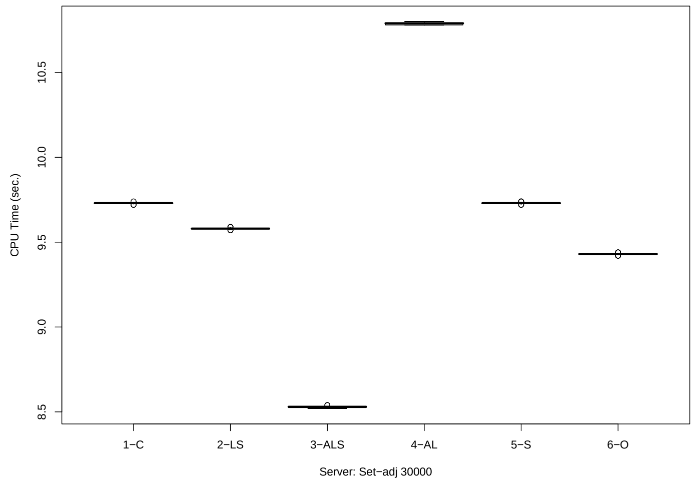

Fig. 12. Set method, server mode

The results are shown in Figures 12 and 13. In the server mode the only optimization that shows improvement over O in the server mode is ALS, with about 9.5% improvement. The other optimizations show different degrees of slowdowns: a slight slowdown (less than 3%) for S, LS, and C, with LS being the closest to O, and a larger slowdown for AL (on the order of 16%). The percent of slowdown or speedup between any pair of optimizations remains constant for the number of loops changing from 20000 to 30000 to 40000.

The hypothesis for explanation of the slowdown is that specialization introduces additional typecasting which cannot be eliminated at running time.

In the client mode `set` shows a very different pattern of behavior: the only optimization that runs faster than O is S which gives a modest speed up of about 2%. ALS is the worst-performing "optimization" resulting in 7% slowdown. AL is close to it with a slowdown of about 6%. LS and C show identical results with 3.5% slowdown. Just like for the server case, the percent of slowdown or speedup between any pair of optimizations remains constant for different number of loops.

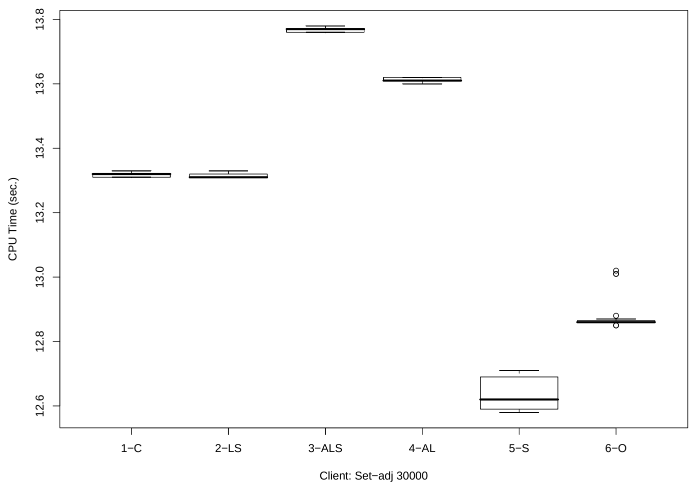

Fig. 13. Set method, client mode

Effects on `get`. The `get()` method takes an integer parameter (an index), calls the `rangeCheck` method (which may generate an opportunity for inlining) and returns an element at that index as an element of type E (the type parameter). The test for `get` assigns the result returned from `get` to a variable of type `Object`, but does not call any methods on it to avoid separate effects of typecasting or devirtualization:

```
public Object testGet(RList<T> list, int numLoops) {
     Random r = new Random();
     Object[] objects = new Object[10 + r.nextInt(100)];
     int size = objects.length;
     for (int i = 0; i < numLoops; i++) {
         for (int j = 0; j < 8675309; j++) {
             objects[j % size]= list.get(j % 2);
         }
     }
     return objects[r.nextInt(size)];
 }
```

The results of `get` testing are shown in Figures 14 and 15. In both modes results are nearly identical for all six optimizations (in the server mode the differences are in the hundredths of a second for a 25-second run and in the client mode they are in the tenths of a second in a 57-second run. Therefore returning an element whose type is the type parameter of the class is not affected by the type specialization if the element is not typecast to a more specific type after it is returned.

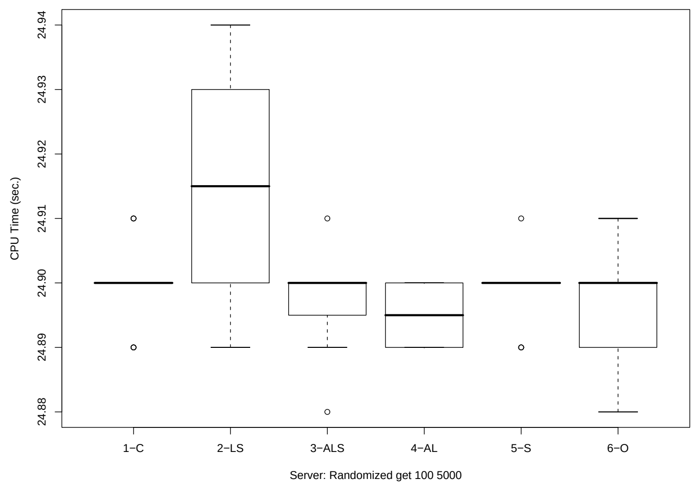

Fig. 14. Get method, server mode

Effects on `add`. `add` method takes a parameter of the type E and returns a boolean. It calls `EnsureCapacity` and then performs one assignment statement of a variable of type E. `EnsureCapacity`, if needed, allocates a new array of type Object and uses arrayCopy to copy the objects of type E into it.

The results are shown in Figures 16 and 17. In the server mode the only specialization that causes a performance decrease is AL. The decrease is about 10%. The other three specializations result in about 35% performance improvement.

The client mode shows a very different pattern. Firstly, we observe a very wide spread of running times. For instance, the complete optimization C takes anywhere between 7.5 and 11.5 seconds in our test runs. This wide spread is observed in several different runs of the same test. This is likely to be caused by unpredictability of memory allocation in `arraycopy`. It is difficult to judge the effects of different specializations based on results with so wide ranges. No consistent pattern of time increase or decrease due to specialization has been observed.

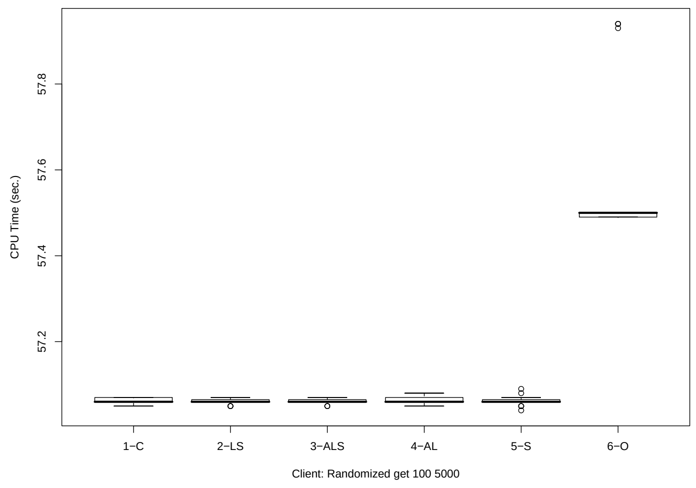

Fig. 15. Get method, client mode

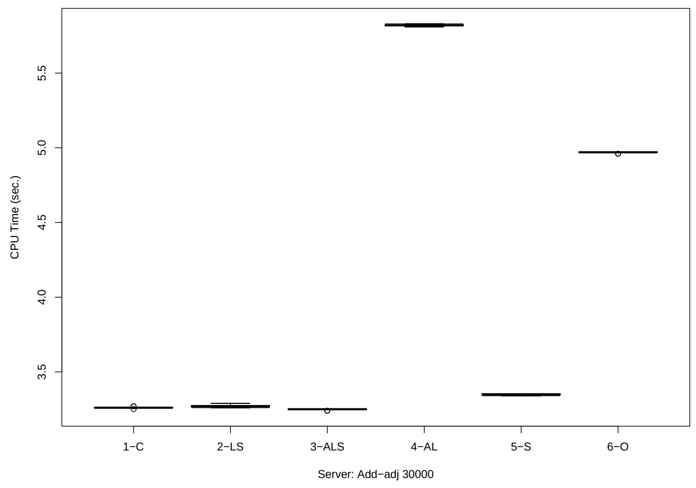

Fig. 16. Add method, server mode

9.3 Effects on Parameter Passing via a Class

Work in progress, to be added shortly

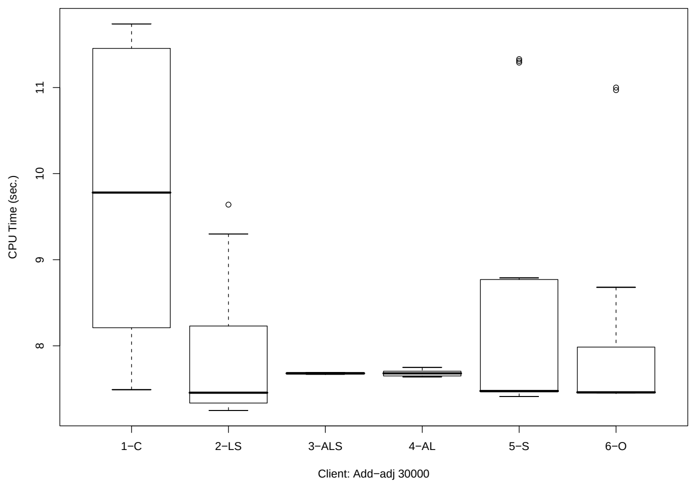

Fig. 17. Add method, client mode

9.4 Static Effects.

When considering effects of specialization, we take into account two kinds of factors: static changes to bytecode and dynamic effects of JVM optimization. Static effects can be observed by comparing bytecode generated for the original program and its specializations. For instance, consider the following line of code in the method `partition` in the `Quicksort` class:

```
inputList.get(i).compareTo(pivot)
```

This fragment generates the following bytecode sequence in the original `Quicksort` class:

```
invokeinterface #54 <research/util/List.get> count 2
checkcast #58 <java/lang/Comparable>
aload 4
invokeinterface #60 <java/lang/Comparable.compareTo> count 2
```

However, the same code fragment in a specialized `QuicksortInteger` class in Interfacespecialization generates more efficient bytecodes:

```
invokeinterface #54 <research/util/ListInteger.get> count 2
aload 4
invokevirtual #58 <research/main/TestInteger.compareTo>
```

The checkcast operation got eliminated and the call to `compareTo` via `invokeinterface` is replaced by `invokevirtual`. We observed that the typecast was eliminated in

LS and C, and a call to `compareTo()` was devirtualized in ALS, S, LS, and C. However, often static factors (such as those obvious in bytecode) do not explain all the time differences.

9.5 Dynamic Effects.

JVM warmup affects specialized programs where the same type is specialized to two different bounds or where a non-specialized version is used alongside the specialized one. Suppose a non-specialized type G has a method m that gets called a sufficient number of times to trigger dynamic compilation. The number of times the non-compiled version of m is called equals to the threshold of the JVM. If we use two different copies of G, each of them will have its own copy of m, and the warmup will take twice as long. Thus, in the quicksort example specialized copies of the class have a warmup delay because methods `set` and `get` in the `ArrayList` class (which perform the bulk of the work) have two copies, one for each specialized class. However are on the order of 0.1-0.2 sec. for 10,000 method calls in server.

Interestingly, constructors contribute to warmup delays greater than regular methods. Each constructor calls its super constructor, creating specialized copies of classes may result in a slowdown. When creating an `ArrayList` the following constructors are called: `ArrayList, AbstractList`, and `AbstractCollection`. Complete specialization C has specialized copies of constructors for all three classes (two specialized and one non-specialized). In LS `AbstractCollection` is shared between all classes. One may notice a small slowdown of C over LS specialization.

Another element that affects a program's performance is the use of `arraycopy` in specialized classes. Copying from an array referenced by `Object[]` variable to `TestInteger[]` can give a substantial program slowdown. However, the delay seems to be small in practice (below 0.3 sec) since `arraycopy` is called only when a larger array needs to be allocated, and since an array size doubles every time, with 1500 elements it is not a significant factor.

Dynamic factors that contribute positively to efficiency are dynamic devirtualization and inlining. However, passing parameters to a specialized class via a non-specialized interface leads to extra run-time typechecks causing overhead of concrete class specializations over interface ones.

10 Conclusions and Future Work

We observed that interface specializations (e.g. LS in the quicksort examples) performs very well in a variety of settings in client and server modes. Optimizations that specialize just the class itself but not the interface or the client code but not the classes are not good options and may occasionally cause a slowdown. Finally, complete specializations perform at about the same level as interface ones but have a much larger code duplication since all classes that reference the interface need to be modified. Thus we consider interface specialization a promising option and plan to further investigate its behavior.

Extending the algorithm to handle wildcards that use specialized types as bounds should not be difficult. These types should just be added in the propagation phase of the algorithm. This approach, however, still needs to be tested. Our future work also includes extending our algorithm to cover parameterized methods, multiple type parameters of classes, and other language features. Our goal is to implement the specialization algorithm and to apply it to larger programs, including standard benchmarks.

References

1. Eric Allen, Robert Cartwright: The case for run-time types in generic Java. Principles and Practices of Programming in Java (PPPJ), 2002, Intermediate Representation Engineering for Virtual Machines (IRE), 2002.
2. Bowen Alpern, Anthony Cocchi, Stephen J. Fink, David Grove and Derek Lieber: Efficient Implementation of Java Interfaces: Invokeinterface Considered Harmless. ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA), 2001.
3. Ole Agesen, Stephen N. Freund, John C. Mitchell: Adding type parameterization to the Java language. ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA), 1997.
4. Gilad Bracha: Generics in the Java Programming Language. Sun Microsystems, java.sun.com, 2004.
5. Francesco Bellotti, Riccardo Berta, Alessandro De Gloria: Evaluation and optimization of method calls in Java. Softw. Pract. Exper., **34** (2004), 395–431.
6. Joseph A. Bank, Andrew C. Myers, Barbara Liskov: Parameterized types for Java. SIGPLAN-SIGACT symposium on Principles of programming languages (POPL), 1997.
7. Gilad Bracha, Martin Odersky, David Stoutamire, Philip Wadler: Making the future safe for the past: adding genericity to the Java programming language. ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA), 1998
8. Thomas Cormen, Charles Leiserson, Ronald Rivest, Clifford Stein: Introduction to Algorithms. MIT Press, 2001.
9. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Design patterns: elements of reusable object-oriented software. Addison-Wesley, 1995.
10. James Gosling, Bill Joy, Guy Steele, Gilad Bracha: The Java Language Specification (3rd Edition). Prentice Hall, 2005.
11. Brian Goetz: Java theory and practice: Dynamic compilation and performance measurement. Java technology series at www.ibm.com, 2004.
12. Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Hideaki Komatsu, Toshio Nakatani: A study of devirtualization techniques for a Java Just-In-Time compiler. ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, 2000.
13. Andrew Kennedy, Don Syme: Design and implementation of generics for the .NET Common language runtime. ACM SIGPLAN conference on Programming language design and implementation, 2001.
14. R Development Core Team: R: A Language and Environment for Statistical Computing. http://www.R-project.org, 2006.

15. Jose H. Solorzano, Suad Alagić: Parametric polymorphism for Java: a reflective solution. ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA), 1998.
16. James Sasitorn, Robert Cartwright: Efficient first-class generics on stock Java virtual machines. Symposium on Applied Computing (SAC), 2006.
17. Ulrik P. Schultz, Julia L. Lawall, Charles Consel: Automatic program specialization for Java. ACM Trans. Program. Lang. Syst., **25** (2003) 452–499.
18. Mirko Viroli, Antonio Natali: Parametric polymorphism in Java: an approach to translation based on reflective features. ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA), 2000.