Effects of generic types specialization on program behavior

Jeremy Bleichner, Nolan Nordlund, Elena Machkasova
Computer Science Discipline
University of Minnesota Morris
Morris MN, 56267
blei0015, nordl056, elenam@morris.umn.edu

Abstract

Generic types are a language feature of Java that allows writing program code that is parameterized over types. The Java implementation of generic types uses a mechanism called type erasure. Type erasure creates a single instance of a generic class, removing all type-specific usage information (e.g. whether it is used with strings or integers). The compiler inserts typecasts to guarantee type-safe calls to instances of the generic class. Since most of the optimizations in the Java programming language are performed at run-time, Java programs using generic classes are slower than they would have been under an implementation that uses type specialized classes.

Our research centers around specialization of generic types – a technique that we developed for optimizing Java programs. The key idea of the optimization is to replace generic classes with copies that are modified to use the actual type parameters that appear in a program. For instance, if a program uses a generic stack of strings, a copy of the stack class that is specialized for strings is created. We have observed program speed-ups of up to 20% due to type specialization.

In this paper we present the overview of the specialization, focus on the challenging issue of its dynamic effects, and present detailed results of the six versions of the code (the original one and five different variations of specialization) for three methods of a generic `ArrayList` – a class in the Java collections library.

1 Introduction and background

1.1 Java Compilation Model

Many programming languages are compiled into a set of platform-specific CPU instructions, but Java is unique in that its compiler converts a program into platform-independent sets of instructions called *bytecode*. A Java interpreter, known as the *Java virtual machine*

or *JVM*, then runs the bytecode. This part of the Java compilation process is called *static compilation*. Platform-independent bytecode guarantees the same behavior of a Java program across different platforms and runtime safety.
The drawback of this system is that interpreting the bytecode is generally slower than a compiled approach. To improve the runtime, *just-in-time or JIT compilers* can be included in the JVM. At runtime, JIT compilers will compile and optimize parts of the bytecode to native machine code to achieve greater performance than static compilation on its own; this is known as a *dynamic compilation*.
A particular JVM which we focus on is the HotSpot™JVM by Sun Microsystems. The HotSpot JVM reduces the overhead introduced by JIT compilers by profiling the program being run to selectively compile only frequently called methods. This approach avoids slowdowns due to dynamic compilation for code fragments that are not frequently used.
The HotSpot JVM can run in *client* or *server* mode. *Client* mode focuses on loading programs faster and uses a simple JIT compiler that will not make heavy optimizations to the code. *Server* mode is slower loading but optimizes the code more effectively. The performance increases in the server mode can be noticeable in longer running programs although this is not always the case.

1.1.1 Overview of Java Bytecode

A Java bytecode instruction consists of an *operation code* (also known as *opcode*) followed by zero or more operands. Operands often refer to elements of a class' *run-time constant pool*. A constant pool contains constants and references to objects and method used in a given class. Constant pool elements are represented in bytecode instructions by a # followed by a number, such as #24. See [4] for more details.
The following bytecode opcodes are important for the further discussion:

- `invokevirtual` denotes a method call that gets resolved dynamically (i.e. at run time). Since methods of a superclass can be overwritten in a subclass, a dynamic (or virtual) method lookup is performed to determine the right method to call.
- `checkcast` - dynamic typecast, e.g. `(String) obj`. If successful, returns an object that can be referenced via the new type. For instance, if `obj` is indeed a `String` then `(String) obj` can be referenced via a `String` variable. If the cast fails at runtime, a `ClassCastException` is thrown.

For some program examples in this paper we show the corresponding bytecode. We use *jclasslib bytecode viewer* [2] to view bytecode. While bytecode generated for a program varies between different compilers (e.g. `javac` and eclipse compiler), the elements essential for our research, such as method calls and typecast operations, are the same for all the compilers that we have tried.

1.2 Generics in Java

Java, like many other programming languages, is *strongly typed*. This means that it requires all variables to have explicitly declared types. One feature that many of these

languages share is called *generic types* or *generics*. Java generics, added in Java 1.5, allow containers such as an `ArrayList` to have *type parameters*. A type parameter forces a container to only hold objects of that type. This allows for code such as:

```
ArrayList<Shape> arr = new ArrayList<Shape>();
arr.add(new Shape(...));
for(int i = 0; i<arr.length; i++) {
  ...
  temp = arr.get(i).shapeMethod();
  ... } // end for loop
```

where `shapeMethod` is any method that can be used by the Shape class. Note that the Generics allows a method usable only by a Shape object to be called on a member of *arr*. Also, an attempt to add a non-Shape object to `arr` would result in a compilation error.
When a generic class, such as `ArrayList`, is declared, it is possible to restrict the range of acceptable type parameters. Such a restriction is called an *upper bound* or *bound* of the generic class. For instance, a priority queue may be restricted to take only `Comparable` objects:

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

If a class that does not implement `Comparable` interface is used as an instance of the type parameter, the program would not compile[1] Any class or an interface may serve as a bound. If no bound is specified, it is assumed to be `Object`.
It should be noted that type parameters in Java generics cannot be primitive types, such as `int`. It is possible to write Java code that looks as if integers are directly added to an `ArrayList`, for instance:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
arr.add(1);
int n = arr.get(0);
```

However, what is actually added to `ArrayList` is an object of a class `Integer`, a so-called *wrapper* class for a primitive type `int`. `Integer` contains the `int` value as an instance variable. The process of converting from an `int` to `Integer` performed by `add` in the example above involves a constructor invocation and is called *boxing*. The reverse process is performed at the call to `get` and is called *unboxing*. In our examples we explicitly use `Integer` instead of `int` to avoid overhead of boxing and unboxing and to make object creation explicit.
Generics in Java are implemented using an approach called *type erasure*. This approach creates only one copy of a generic type: the one in which the type parameter is replaced by the bound of the type. For instance, even if a program uses `ArrayList<Integer>` and `ArrayList<String>`, only one copy of `ArrayList` actually exists, and that is an array list of `Object`. The compiler inserts necessary typecasts (for instance, the `get`

[1] Strictly speaking, the bound should be declared as `T extends Comparable<T>` to guarantee that the elements of the priority queue are comparable *to each other* and not to any other objects, but we will omit these details for now.

method of ArrayList returns an Object so its result is cast to the actual parameter type, such as Integer or String). This is done to ensure that all Java code is backwards compatible with older Java code and applications.

1.3 Overview of Specialization of Generics

1.3.1 General Idea of Specialization

We propose and study an optimization of Java programs that we call *specialization of generic types*. The optimization creates a copy of a generic type with the type bound replaced by a more specific type based on the actual type parameter. For instance, if a program uses ArrayList<Integer>, a separate type ArrayListInteger is created. The bound of the original ArrayList is Object, the bound of the newly created ArrayListInteger is Integer. The rest of the code of ArrayListInteger is exactly the same as in ArrayList, except for a few changes explained in section 2. References to ArrayList in the original program are then replaced by references to ArrayListInteger. If a program additionally uses ArrayList with a different actual type parameter, e.g. ArrayList<String>, then a separate type ArrayListString with the bound String will be created. The specialization may also be applied to interfaces and parameterized static methods. The optimization is performed at the Java source code level.

The type specialization improves program performance in two major ways: it leads to elimination of unnecessary type casts and provides opportunities for the JVM to better optimize the code, utilizing more precise type information.

Typecast elimination occurs, for example, when an object is retrieved from a generic data collection, such as ArrayList. Consider the following simple code fragment. In our examples we are using Integer instead of int for the element added to the array and for the type of variable n because implicit boxing/unboxing of integers would add unnecessary complexity to our code.

```
ArrayList<Integer> arr = new ArrayList<Integer>();
arr.add(new Integer(5));
Integer n = arr.get(0);
```

The assignment on the last line requires a typecasting to Integer since the get method of ArrayList returns an Object. The bytecode instructions generated by this fragment confirm this (see section 1.1.1 for details on Java bytecode):

```
invokevirtual #36 <research/util/ArrayList.get>
checkcast #27 <java/lang/Integer>
astore_1
```

Only the bytecode for the last line of the Java code is shown. The checkcast operation is needed before the assignment can take place (the astore_1 instruction performs the assignment).

If the ArrayList is replaced by its specialized version ArrayListInteger, the checkcast is no longer needed. Below is the Java code and the bytecode fragment for the last line (the call to get and the assignment statement):

```
ArrayListInteger<Integer> arr = new ArrayListInteger<Integer>();
arr.add(new Integer(5));
Integer n = arr.get(0);
```

```
invokevirtual #52 <research/util/ArrayListInteger.get>
astore_1
```

Differences in constant pool numberings between the two examples are not important.
The bytecode instruction `invokevirtual` performs *dynamic method lookup* (also referred to as *virtual method lookup*). For instance, if a method `equals` is called on an object, the run-time system has to determine the correct method based on the actual type of the object. For instance, since `equals` in `Integer` overwrites `equals` in `Object`, the `equals` method of `Integer` will be called on an `Integer` object.
Dynamic method lookup, however, can be optimized if the method to be called can be uniquely determined by the type hierarchy. Such an optimization is called *devirtualization*. For instance, consider a static method[2] that takes an ArrayList and calls an `equals` method on its element:

```
public static <T> void originalMethod(ArrayList<T> arr) {
   boolean test = arr.get(0).equals(arr.get(1));
   System.out.println(test);
}
```

The method is called as follows:

```
ArrayList<Integer> arr1 = new ArrayList<Integer>();
arr1.add(new Integer(5));
arr1.add(new Integer(5));
originalMethod(arr1);
```

The method prints `true` which proves that the `equals` method of `Integer` is called (the `equals` in `Object` would have returned `false` because it compares memory addresses only, and the two integers in the array have different addresses). However, the `invokevirtual` bytecode instruction calls the `Object` method (the two calls to `get` are included to provide the context of the call):

```
invokevirtual #89 <research/util/ArrayList.get>
aload_0
iconst_1
invokevirtual #89 <research/util/ArrayList.get>
invokevirtual #104 <java/lang/Object.equals>
```

The call to `equals` gets resolved to the overwriting method in `Integer` only at runtime, based on the actual class of the `ArrayList` element which is an `Integer`. Thus the dynamic method lookup is non-trivial.

[2]Static methods in Java are methods defined for an entire class, not for individual objects. Methods in Java can be parameterized separately from the class that surrounds them. For details see [1].

If the method is known to take `ArrayListInteger` instead of `ArrayList` then the target of the call to `equals` is known to be in the `Integer` class right when the program is loaded. Note that the change from `ArrayList` to `ArrayListInteger` as the method parameter also leads to the change of the type bound for the method (from `<T>` to `<T extends Integer>`), otherwise `ArrayListInteger` would not be a valid parameter to the method:

```
public static <T extends Integer> void
      optimizedMethod(ArrayListInteger<T> arr) {
  boolean test = arr.get(0).equals(arr.get(1));
  System.out.println(test);
}
```

The method call is completely analogous to the one above for `ArrayList`. The bytecode changes to the following:

```
invokevirtual #96 <research/util/ArrayListInteger.get>
aload_0
iconst_1
invokevirtual #96 <research/util/ArrayListInteger.get>
invokevirtual #99 <java/lang/Integer.equals>
```

Here the `equals` method is of class `Integer`. The method target is unique (it can only be in the class `Integer` since nothing overwrites it) and the method call can be replaced by a direct ”jump” to that target, i.e. devirtualized. Thus more precise type information saves time for method lookups.
Note that in the first example with `ArrayList` the call can still be devirtualized at runtime based on statistical evidence that the `Integer` method gets called the most frequently. However, hundreds of calls will be needed to gather the statistics, and such an optimization would have to be able to ”roll back” if the call target changes [3]. Thus it is less efficient than the devirtualization when the target of the method call is known precisely when the program is loaded.
In our tests we have observed program speed-ups of up to 20% (see [5]). Note, however, that sometimes a specialization may also increase the running time. This happens, for instance, when a generic data structure stores its elements as an array of `Object` type, but the specialized version uses an array of `Integer` instead. Then an additional typecheck may be needed before an element is added to the array, whereas the non-specialized version could safely add any object without a typecheck. The exact cases when this happens depend on types of method parameters.

1.3.2 Partial and Complete Specializations

Interestingly, in Java a generic type that inherits from a generic interface may have a more restricted type bound than the interface. For instance, suppose an interface is defined with the default `Object` bound for its type parameter `T`:

```
interface List<T>
```

It is possible for a class to implement `List<T>` but to restrict the parameter to `Integer`:

```
class ArrayListInteger<T extends Integer> implements List<T>
```

This feature allows us to specialize only a part of a class's hierarchy, not the entire hierarchy. For instance, in one optimization version we specialize `ArrayList` but not `List` (very similar to the example above, except the example omits several classes in the `ArrayList` hierarchy). We refer to specializations that specialize only a part of a class's hierarchy as *partial specializations*, in contrast with *complete specializations* that specialize the entire generic hierarchy of a class, including all classes and interfaces that it inherits from.

The advantage of a partial specialization is that fewer classes need to be changed since classes that use higher-order interfaces would not be affected by specialization of classes that implement them.

Our tests show mixed efficiency results for partial specializations: some partial specializations actually perform worse than the original code for some tests, but some perform just as well as the complete specialization, and in some cases even better. We found out that in the `ArrayList` hierarchy specializing `List` interface but not interfaces above it gives the optimal behavior. We call this specialization *interface specialization*. For more details on our specializations see section 2.2, for the results and discussion see section 3.

1.3.3 Dynamic Effects of the Specialization

Typecast elimination and change in types in `invokevirtual` are the most straightforward effects of specialization of generics and are easy to detect in the bytecode. Such changes in the code account for a substantial percent of running time decrease. However, there are also some changes in behavior that cannot be directly explained by changes in the bytecode. They include method devirtualization that happens dynamically based on statistical data about method calls, method inlining, dynamic typecast elimination, and similar optimizations. They also contribute to runtime improvement for some specializations, although in some cases they may cause a slowdown.

Understanding these less obvious dynamic effects is a necessary but challenging task. This paper focuses on such effects, using tests for `set`, `add`, and `get` method of the `ArrayList` class in the Java collection framework (JCF).

2 ArrayList Example

2.1 JCF ArrayList Hierarchy and our Examples

The *Java Collections Framework, or JCF,* is an architecture that allows us to represent and manipulate collections in a quicker and easier way. The JCF include *interfaces, implementations and algorithms*. Interfaces specify general behavior of a collection. For instance, `List` interface specifies methods that any list must have, such as `add` and `get`, but their implementation may differ in different classes that implement `List`, e.g. `ArrayList` and `PriorityQueue`. JCF also provides algorithms for common operations, such as sorting. As of Java 1.5, most classes in JCF use generics.

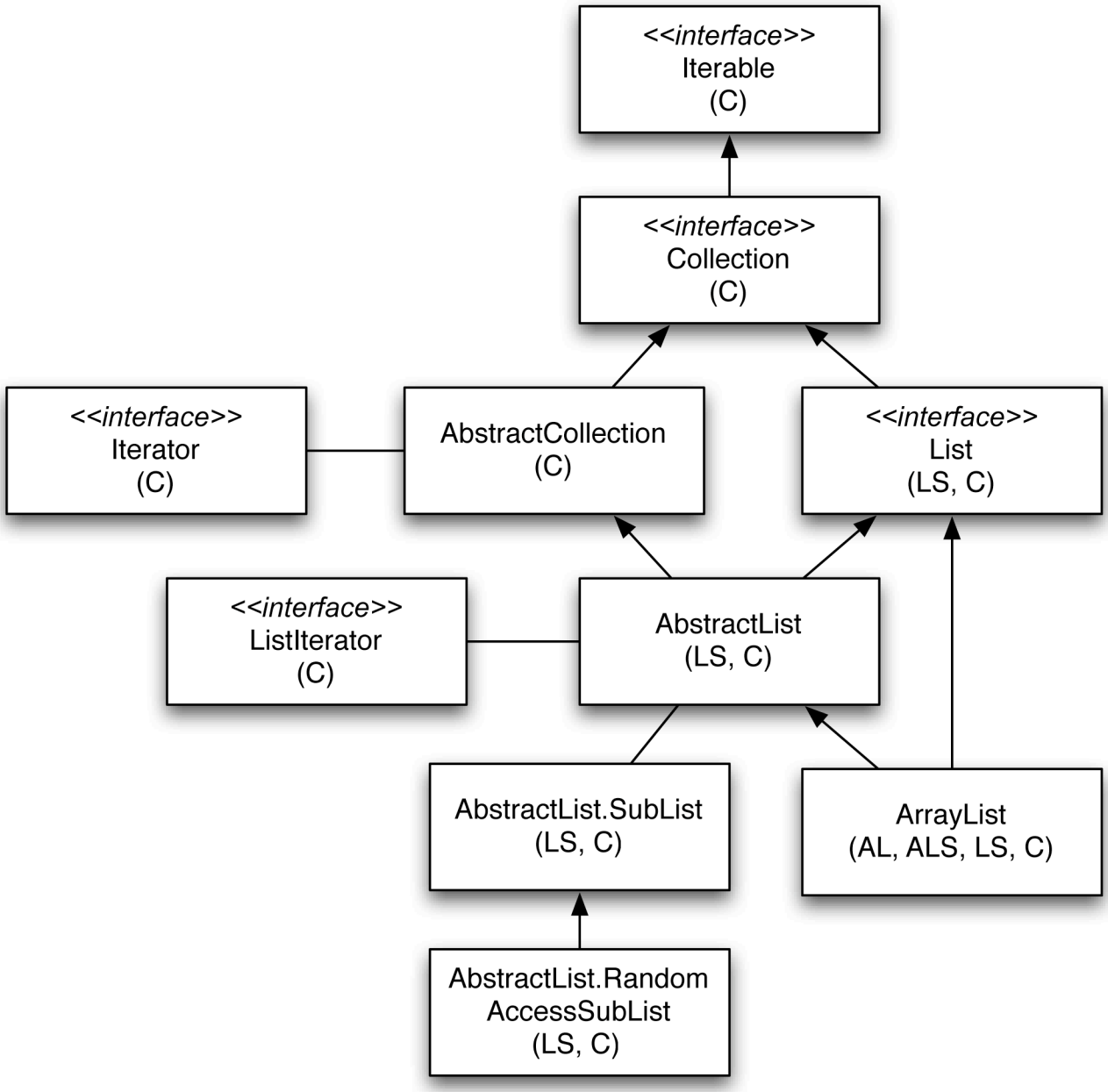

Figure 1: The hierarchy for ArrayList

To study the effects of generics specialization (see section 1.3.1) we looked at different specializations of the source code of `ArrayList`. We have copied the entire `ArrayList` hierarchy. To ensure that we are not accidentally referring to the standard `ArrayList` and other JCF classes, we simply renamed our copy to include an R before the name, e.g. `RArrayList` instead of `ArrayList`. We also used our own class `TestInteger` instead of `Integer` to guarantee that there are no inadvertent boxing/unboxing effects. Just like `Integer`, `TestInteger` stores a single `int` value and implements `Comparable` interface. It also has a static counter that counts the number of times `compareTo` was called; we use it in some tests.
As shown in diagram 1, in JCF `ArrayList` implements a `List` interface and extends an `AbstractList` class (for simplicity the diagram uses the original names in JCF without the R prefix; ignore the letters in parentheses, they will be explained in section 2.2). `AbstractList` in turn extends `AbstractCollection`. `Collection` interface is a supertype for both `List` and `AbstractCollection`. The hierarchy also includes iterators and the `Iterable` interface.
Diagram in figure 2 shows the structure of our tests. The left-hand side shows types of variables for various objects that we use and the method calls. The right-hand side shows the corresponding objects, those created by the constructor and (in the last line) the actual call of `add` on the `ArrayList` (note that the copy of `ArrayList` that we use is `RArrayList`). The diagram only shows a call of `add`, but the same sequence is used for other method tests as well. The rectangular block of the diagram shows the type bound of the class or interface. For instance, the constructor of `ArrayList` runs in the context where the type parameter `T` is bound by `Object`.
Our test starts by creating a new `RArrayList` parameterized with `TestInteger`. We refer to the `RArrayList` via an `RList` variable.

```
RList<TestInteger> special = new RArrayList<TestInteger>();
```

The `RArrayList` is what we are going to be performing our adding, setting, getting, etc. operations on. Our "client" code, i.e. the code that uses the `RArrayList`, is located in a class `ListReader`. In the examples considered here the "client" code consists of various test methods (we tests each `ArrayList` method individually). The `ListReader`class is generic and its bound is `Comparable`:

```
public class ListReader<T extends Comparable<T>>
```

The `Comparable` bound is chosen because we wanted to model a sorting algorithm which calls `compareTo` on `RArrayList` elements to sort them. In this paper we focus on effects of `ArrayList` methods and do not discuss `compareTo`. As seen in diagram 2, we create `ListReader` with an `Integer` type parameter. We are adding a small number of elements to the `RArrayList` (the default is 2, but it can be changed by passing an extra flag to the program) and iterate over these elements. This way the heap space is small and there is no garbage collection.
Our tests are repeatable which means that once a test is written and ran, it is never changed. Tests are chosen based on a flag passed to the program via command line. We can also control the number of loops performed by the program from the command line by passing

a corresponding parameter that is stored in the variable `numLoops`. Since the goal of the tests is to detect time differences between the optimized and non-optimized code, we call the method that we are testing a very large number of times: the parameter passed on the command line is usually between 20,000 and 60,000 and it indicates the number of times we iterate over a loop that calls the corresponding method of `RArrayList` 10,00 times. After the flags are read, a new `ListReader<Integer>` called `reader` is created and an appropriate test method is called (in this example, `set` method):

```
reader.testSet(special, numLoops);
```

Below is the `testSet` method in the `ListReader` class. The `RArrayList` is passed to the method as a variable of type `RList`:

```
public int testSet(RList<T> special, int numLoops) {
     T element = special.get(0);
        int size = special.size();
        for(int i = 0; i < numLoops; i++){
            for(int j = 0; j < 10000; j++){
                special.set(j%size,element);
            }
        }
        return special.get(0).hashCode();
    }
```

The return of the method is needed to prevent the JIT from applying dead-code elimination to the method.

2.2 Six Optimizations of ArrayList Example

Our optimization creates specialized copies of classes for the actual type parameters used in the program. For this group of tests the actual parameter is `TestInteger`. For instance, `RArrayListInteger` is a specialized copy of `RArrayList` defined as follows (omitting the "extend" and "implement" clauses):

```
public class RArrayListInteger<E extends TestInteger>
```

The body of the class is exactly the same as `RArrayList` except that every time a new `Object[]` is created in the original code, a `TestInteger[]` is created in the specialized version. The reason for this is that if we store elements in an array of `Object`, we would lose most benefits of the optimization since there will still be typecasting when elements are retrieved by a `get` method.
The six versions of the program that we compare differ in the classes and interfaces that are specialized. Note that the "client" class `ListReader` is specialized in some versions but not in others. Its non-specialized version has `Comparable` as its bound, and the specialized version has `TestInteger`. Figure 1 marks every class with a list of versions of the optimization in which it is specialized. The six versions are as follows:

- **O** is the original code with no classes specialized.

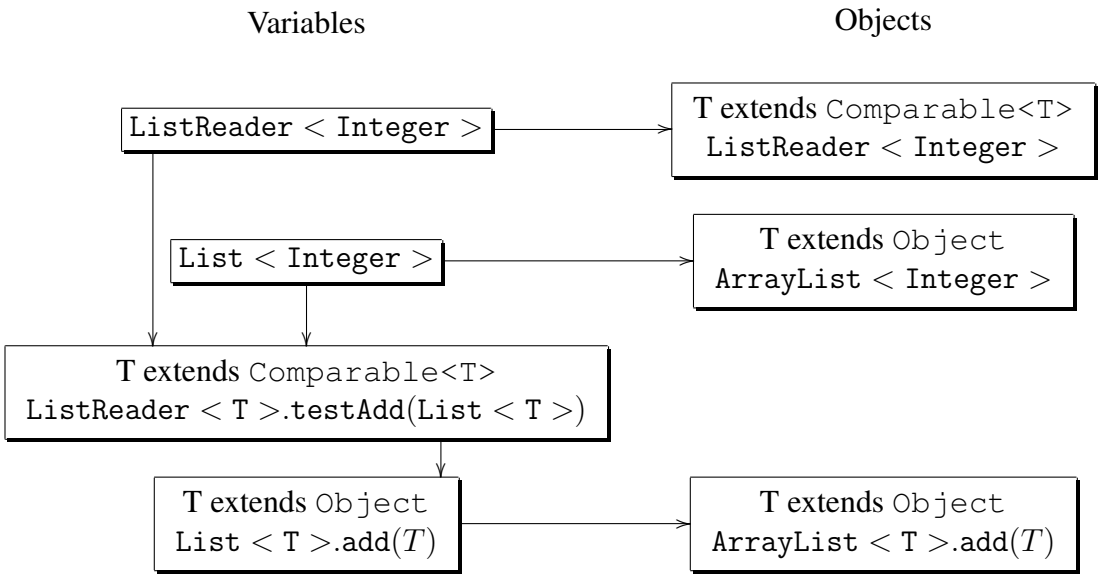

Figure 2: Method calls and types for our test example

- In **S** only the "client" `ListReader` class is specialized.
- In **AL** the only specialized class is `RArrayList`.
- In **ALS** combines AL and S: `RArrayList` and `ListReader` are specialized.
- In **LS** the interface `RList` and all classes below it in the hierarchy are specialized. This causes specializing `ListReader, RArrayList, RAbstractList`, and some inner classes. This is an interface specialization (see 1.3.2).
- **C** is a complete specialization: the entire `ArrayList` hierarchy is specialized and so is `ListReader`.

Note that AL, ALS, and LS are partial specializations since they only modify a part of the `ArrayList` hierarchy.

2.3 Testing Methodology, Framework, and Challenges

In order to make sure our tests are accurate we run each test 20 times in each of the client and the server modes. This process ensures that we can identify outliers and come close to the true mean of the run times. We use automated scripts to recompile the program, run all versions of it with the same flags, and record the statistical data. We measure the run-times using the system *time* command. While this includes the JVM startup and loading time, the large number of loops in our test runs ensure that the startup is not a significant factor.
The JVM's lack of transparency makes it difficult to design tests that clearly test only one aspect of a program. This means that any change in the code must be meticulously replicated across the code. Surprising amounts of variance in efficiency can come from aspects of programs that are not commonly associated with efficiency such as return types, public versus private accessibility, and similar minor changes.

Since the tests were not ran on a dedicated machine, they were subject to interference from operating system processes running, CPU limitations, memory limitations and many more factors. We take a number of precautions (discussed in section 2.3.1), the most significant of which is running the tests many times. It is this repetition that makes any results obtained statistically significant. Some tests, however, are more vulnerable to these interferences and it is made obvious by the variance in their results. More memory-intensive tests, such as `add`, belong to this category.

2.3.1 Test Setup

When dealing with containers, such as `ArrayList`, there are three operations that are almost always the most called: get, set, and add. When optimizing programs these three operations provide their own unique challenges. For instance, in `add` tests we have to make sure that the array size does not grow to the point when the garbage collection becomes an issue. In order to achieve consistency in our tests we chose to test each of these three operations separately. This is discussed further in section 3.
One precaution that is taken to obtain consistency in the tests is that all of our tests are run on the same computer. The specifications for the computer we used are:
AMD Athlon™64 Processor 3200+
512MB DDR RAM
Fedora Core 7
Kernel: 2.6.23.17-88.fc7 SMP i686
Java Version: Sun JDK 1.6.0_04
Time Binary: GNU time 1.7
glibc version: 2.6
It is important to note that this computer is part of a public lab environment where the user accounts are stored on a central server. To keep this from impacting the tests, the testing script is run from the /tmp directory on the computer to avoid interference from the server file system. Once the script has finished the results are used to produce statistical data and graphs. The results are committed to version control system (cvs).

3 Tests, Results, and Observations

3.1 Tests for Get

Lets look at the code in `ListReader` that tests the `get` method:

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

```
        }
        return gadgets[r.nextInt(size)];
    }
```

The method itself in `ArrayList` (or, equivalently, in `RArrayList`), looks like this:

```
    public T get(int index) {
        RangeCheck(index);

        return elementData[index];
    }
```

For `get` `numLoops` is usually about 1000-5000 (significantly less than `set` or `add`). This method returns a randomly selected element. This return is used to help avoid any dead code elimination (an optimization done by JIT compilers). The results for the server

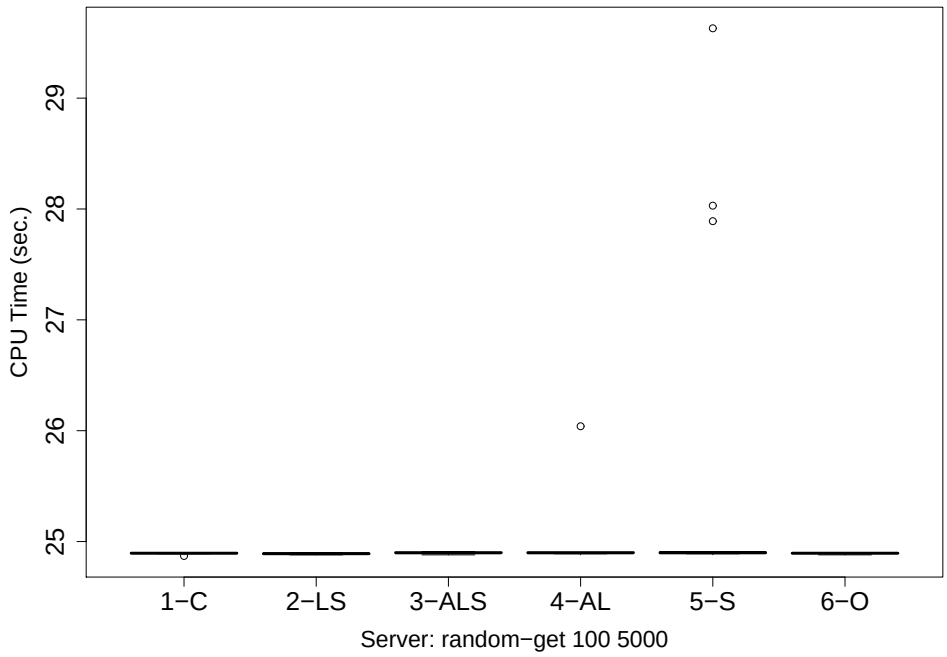

Figure 3: A test run using the get method with 5000 loops on the server JVM

version of the JVM are shown in figure 3. They show no significant difference between any of the specializations and the original unspecialized code. The same test in the client JVM shows similar results. This shows that the specialization does not affect `get` method if its result is not cast to a more specific type. Note that if there is a typecast to `TestInteger` in the testing method then the specialized versions would benefit from typecast elimination. However, our goal is to study behavior of the `ArrayList` methods themselves, and not their interactions with other code.

3.2 Tests for Set

Now since the code for `testSet` was covered in section 2.1. Below is the code for `set`; `RangeCheck` is a private method that checks whether the index is within the valid range.

```
public T set(int index, T element) {
        RangeCheck(index);

        T oldValue = elementData[index];
        elementData[index] = element;
        return oldValue;
}
```

This method is similar to the `get` method except that `set` also stores the element in the array. Figure 4 shows that the ALS specialization performed significantly better than both

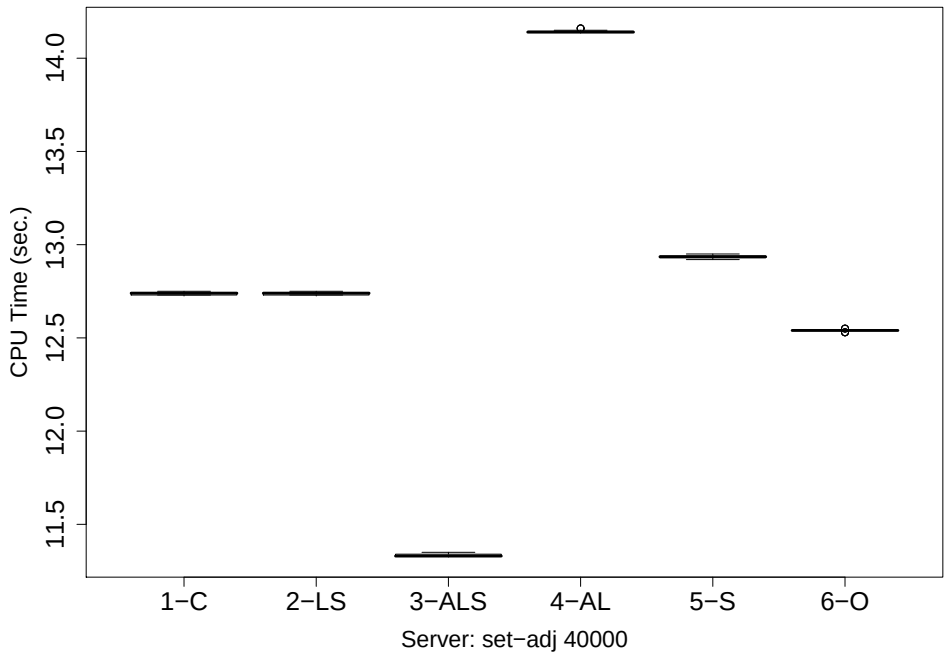

Figure 4: A test run using the set method with 40000 loops on the server JVM

the original code and the other specializations in the server JVM. In the client JVM all optimizations performed worse than the original code and (with the exception of the C specialization) also performed worse than that same specialization in server.

3.3 Tests for Add

While writing the tests for add we endeavored to keep the code as close to the code for `get` and `set` as possible:

```
public int testAdd(RList<T> special, int numLoops) {
     T element = special.get(0);
        for(int i = 0; i < numLoops; i++){
         special.clear(); // get rid of the old elements
            for(int j = 0; j < 10000; j++){
                special.add(element);
            }
        }
        return special.get(0).hashCode();
}
```

`add` calls a public method `ensureCapacity` to increase the size of the array if needed:

```
public boolean add(T o) {
        ensureCapacity(size + 1);
        elementData[size++] = o;
        return true;
}
```

The only difference between `testSet` and `testAdd` (besides calling different methods) is that `testAdd` removes all the old values before adding the new value. `set` and `add`, however, have more differences. Firstly, `add` calls `ensureCapacity` instead of `RangeCheck` because for add the size of the `ArrayList` may need to be increased (recall that `ArrayList` doubles the size of the storage array when the current array becomes insufficient). For this reason size is also incremented. `add` also doesn't store the old value of the index in the `ArrayList` because it returns `true` instead of the old value. Although different, the use of the return values in test code have the same purpose of avoiding dead code elimination. Figures 5 and 6 are good examples of the variation between client and server JVMs. In figure 5 all of the specializations are tightly grouped at about 35% faster than the original (unspecialized) code (except for the AL specialization, which is slower). In contrast to that, many of the client times in figure 6 have a much larger variance, and all are slower than the original code.

4 Conclusions and Future Work

Our tests show a large variety of behavior for very simple methods in `ArrayList` when we perform the same six versions of specialization. We have also observed drastic differences between the client and the server dynamic compilers. The server compiler seems to benefit more from our specializations. This is encouraging because the server JVM mode is usually used for high-performance programs. We also observed that the interface specialization (LS) performs just as well as the complete one (C). This observation opens an opportunity for a specialization that affects only a subset of a program.
Despite an extensive data collection, however, we still do not have a proven explanation of the differences between the six optimizations and between the three methods that we have considered. While we have formed plausible hypotheses, the lack of transparency

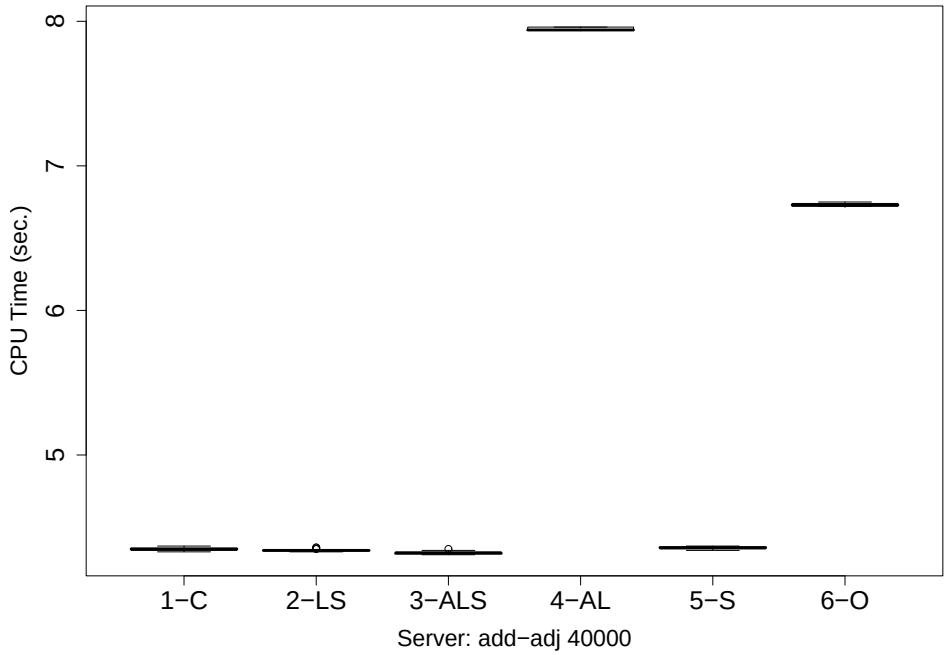

Figure 5: A test run using the add method with 40000 loops on the server JVM

of the JVM optimizations makes it difficult to prove or disprove them. Our immediate future goals are two-fold. Firstly, we would like to improve the testing methodology to allow us to narrow down program behavior patterns, perhaps by creating methods in our copy of the `ArrayList` that would emphasize specific features of the three methods in question (`get`, `set`, and `add`). Secondly, we would like to utilize tools, such as JVM command-line options and profilers, to observe the behavior of the JVM with respect to dynamic optimizations. For instance, if we can detect from an output of a profiler that a certain method was inlined in one optimization but not in the other, we can generalize these patterns to other program. Our ultimate goal is to implement an algorithm for specialization of generic types which would apply it when it is beneficial.

References

[1] BRACHA, G. Generics in the java programming language. *Sun Microsystems, java.sun.com* (2004).

[2] EJ-TECHNOLOGIES. *Jclasslib Bytecode Viewer, version 3.0*. www.ej-technologies.com.

[3] ISHIZAKI, K., KAWAHITO, M., YASUE, T., KOMATSU, H., AND NAKATANI, T. A study of devirtualization techniques for a java just-in-time compiler. In *OOPSLA*

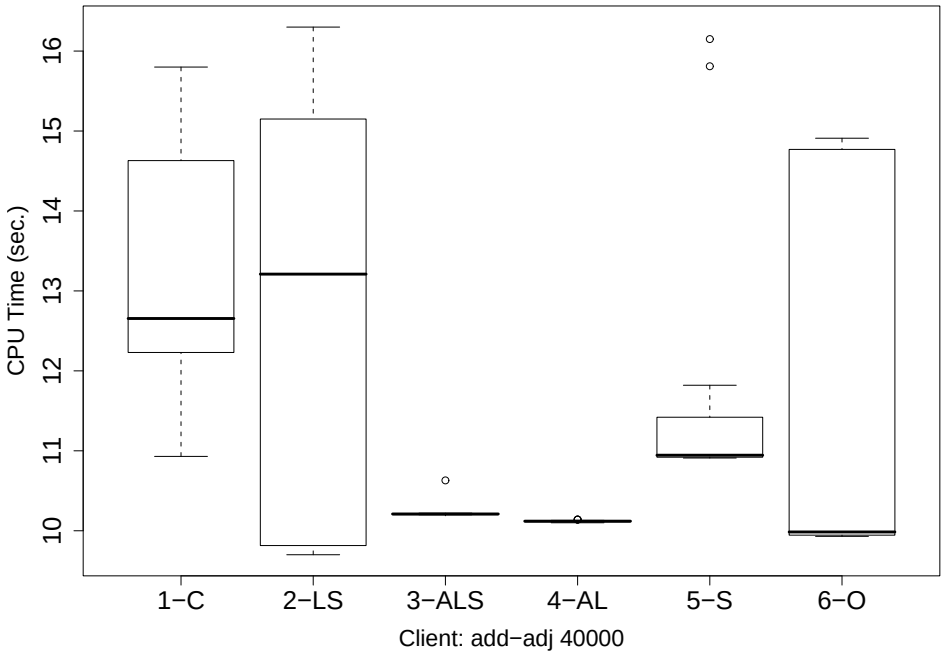

Figure 6: A test run using the add method with 40000 loops on the client JVM

'00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (New York, NY, USA, 2000), ACM Press, pp. 294–310.

[4] LINDHOLM, T., AND YELLIN, F. *The Java(TM) Virtual Machine Specification (2nd Edition)*. Prentice Hall PTR, April 1999.

[5] MAYFIELD, E., ROTH, J. K., SELIFONOV, D., DAHLBERG, N., AND MACHKASOVA, E. Optimizing java programs using generic types. In *OOPSLA '07: Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion* (New York, NY, USA, 2007), ACM, pp. 829–830.