Choosing Efficient Inheritance Patterns for Java Generics

Fernando Trinciante, Isaac Sjoblom, and Elena Machkasova
Computer Science Discipline
University of Minnesota Morris
Morris MN, 56267
trinc002@umn.edu, sjobl014@umn.edu, elenam@umn.edu

Abstract

Java generic types allow a programmer to create parameterized data structures and methods. For instance, a generic Stack type may be used for integers in one instance and for strings in another. Java compiler guarantees in this case that integers and strings are not mixed in the same stack. We study runtime efficiency of a certain inheritance pattern related to Java generic types: *narrowing of a type bound*. This pattern takes place when a generic type allows a more restricted type of elements than its supertype. We examine a slowdown caused by this pattern for some method calls and study the reasons for it. Knowing cases when the slowdown takes place and the reasons for it would allow software developers to make informed choices when using generic types in their programs.

1 Introduction

Generic types are a feature of Java that allows writing program code that is parameterized over types. For instance, generic types allow creation of a Stack class that works for strings or integers, but will not allow mixing strings and integers in the same stack. In this case String or Integer class are referred to as type parameters for the generic Stack class. When a generic type in Java is defined, the programmer must specify the most general type of its type parameter, called the type bound. For instance, the type bound of a generic `EmployeeList` type would be `Employee`. This means that the type parameter of `EmployeeList` is allowed to be any subtype of `Employee`, e.g. `HourlyEmployee`. If no explicit bound is specified for a generic type, its bound is `Object`.
Generic types may by subtypes of other generic types. Commonly the type bound of a subtype is the same as the type bound of its supertype. However, Java allows for the subtype to have a more restrictive bound than its supertype. We call this inheritance pattern a *bound narrowing*. We examine runtime efficiency effect of bound narrowing. We are comparing 3 cases of relations between a generic type and its supertype:

1. Both the type and its supertype have an `Object` as a type bound.

2. Both the type and its supertype have a more specific bound, such as `Integer`.

3. The type has a specific bound such as `Integer` and its supertype has a more general one, such as `Object`.

In our tests we use a class `ArrayList` and an interface `List` in the Java Collections Framework. We create modified versions of the `ArrayList` hierarchy, narrowing bounds for some of the types. Our tests make multiple calls to `ArrayList` methods (such as `get` and `add`) via a `List` interface. We measure the total running time of the program. We run our tests in different settings of the JVM, including a mode that does not perform any dynamic program optimizations (pure interpreted), a mode where a specific optimizations *inlining* is turned off. We also run them in the regular mode where all optimizations are enabled.
We observe that in an interpreted mode there is a substantial slowdown in case 3 compared to cases 1 and 2 above. We discuss reasons for this slowdown. We relate this slowdown to inlining and show that for some methods dynamic optimizations in the regular mode are able to remove this slowdown, but it remains for some methods. Our results will allow software developers to make informed choices about bound narrowing when implementing hierarchies of Java generic types.

2 Background

2.1 Java Generic Types

2.1.1 Overview of Java Generic Types

The Java programming language includes classes and interfaces, both are collectively referred to as types. Java relies on the fact that its variables have specific types. Java's type hierarchy has a class Object at the top and all other types are subtypes of type Object by default. Classes can be subtypes of interfaces or classes and inherit all the methods of the parent type, while having more specialized methods than the parent.
Note that in this paper we are considering only Object types; in Java there is a difference between Object types and corresponding primitive types. For example, there is an Object type `Integer`, and a corresponding primitive type `int`. Object type `Integer` has methods and expanded usability while the primitive type `int` is the raw data form of an integer. In our examples, we use `Integer` and not `int`.
Java and several high level programming languages that use types share a feature called generic types, or generics for short. Java generics allow for the inclusion of parameters on data containers such as list. Parameters force the container to hold elements of a specific type. An example of this would be an `ArrayList` with a type `String` (i.e. `ArrayList<String>`). This ensures that all the elements in the `ArrayList` are strings. String-specific operations can be done and methods can be called on elements of the `ArrayList<String>` without having to verify that every element is a `String`.

Ultimately this is used to iterate over lists and call methods on its individual elements, as shown below:

```
ArrayList<String> alString = nonEmptyArrayList();
        String concat = "";
        for(String s: alString){
        concat += s.toLowerCase();
}
```

Bounds in Java generics are types used to limit generic parameters while still allowing more than one specific type of parameter. The limit of generic parameters is to subtypes of the bound. An example of a bound at the class declaration of a class called `MyComparableList` is:

```
public class MyComparableList<T extends Comparable>{
//methods and constructor
}
```

Here, the bound is type `Comparable`, which is an interface with the method compareTo(). This ensures that all elements in `MyComparableList` implement `Comparable`interface (note that the keyword for declaring a bound is `extends` regardless of whether it is followed by a class or an interface) [1].

2.1.2 Narrowing a Type Bound

Interestingly, if a Java generic type inherits from a generic interface or a class, the supertype may have a less restrictive type bound than the type itself. We refer to this inheritance pattern as *narrowing the type bound*. 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 (narrow) the generic type parameter to `Number`:

```
class ArrayListNumber<T extends Number> implements List<T>
```

In this case only `Number` and its subtypes are allowed as actual type parameters for `ArrayListNumber`. For instance, `ArrayListNumber<Integer>` is valid because `Integer` is a subtype of `Number`, but `ArrayListNumber<String>` is invalid since `String` is not a subtype of `Number`.
This inheritance pattern allows more flexible code (for instance, we may have only one `List` implementation, but multiple more specific classes that inherit from it). However, there are two prime concerns with this choice of generic bounds for inheritance.
One concern is the possibility of code that typechecks at compilation type, but has a runtime type exception. For instance, consider the following code fragment:

```
List<String> theList = new ArrayListNumber();
theList.add("hello");
```

Since `String` is a valid type parameter for `List`, the first line is valid. The second one is also valid because it is adding a string to an list of strings. However, since `theList` refers to `ArrayListNumber` and the bound of the `ArrayListNumber` is `Number`, the call to `add` with a string parameter throws a runtime `ClassCastException`.
Note that in order to get the code to compile we had to use an unsafe style of programming: declaring a generic type without the actual type parameter. If we use a type parameter in the declaration of `ArrayListNumber`, the code does not compile because the actual type parameters do not match:

```
List<String> theList = new ArrayListNumber<Integer>();
```

While the direct declaration does not compile, in a complex program it is possible that a subtype with a narrowed bound (such as `ArrayListNumber`) is passed via a method or a sequence of methods so that the static compiler cannot trace the type parameter mismatch. Narrowing type bounds opens a possibility of a runtime error in a statically type checked code which is a type safety violation.
The other possible issue related to narrowing type bounds is efficiency. Since it is possible for this pattern to throw a `ClassCastException`, there must be a type check performed at runtime to detect a type mismatch. Our paper investigates whether this type check leads to a runtime inefficiency.

2.2 Java Compilation Model

Unlike statically compiled languages, in which a program is optimized and compiled into machine code at once, Java has a two phase compilation model. First, the program is compiled by a *static compiler* such as Javac into bytecode for portability. Second, the bytecodes are executed on any system that has a Java execution environment, referred to as the *Java Virtual Machine (JVM)*. Most modern JVMs are equipped with a *Just-in-time compiler* (JIT) that performs optimizations as the program is being run and may convert a portion or all of a program's bytecode to native code "on the fly". This process is called *dynamic compilation*.

2.2.1 Overview of HotSpot JVM client and server mode

HotSpot is the name of Sun Microsystems' JVM, which is one of the most commonly used JVMs in the market. This 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. *Server* mode is slower loading but analyzes the program more thoroughly and thus optimizes the code more effectively. The performance increases in the server mode can be noticeable in longer running programs.
On the Java 2 Platform, the client dynamic compiler has the following key features [8]:

- It uses *static single assignment* (SSA) form as an internal program representation. SSA is a representation in which each variable is assigned only once. It improves

compiler performance because values of variables do not change so the code is easier to analyze [3].

- The emphasis is placed on extracting and preserving as much information as possible from the bytecode. For example, variable usage information and dependencies of method calls are extracted. This approach leads to reduced compilation time. The client VM does only minimal inlining and no deoptimization (see section 2.3). The lack of deoptimization leads to focusing primarily on local optimizations [3].

- Default compilation threshold for client is $\sim$1500. This means that the JIT selects the methods to be compiled based on the frequency of their execution. When a method's execution counter reaches the compilation threshold, the method is compiled to native code.

For the server dynamic compiler, we should mention [8]:

- It is tuned for the performance patterns of a typical server-side application.

- It uses an *advanced static single assignment (SSA)-based* internal representation, different from the one used by the client JVM.

- The optimizer performs all the classic optimizations, including dead code elimination, loop invariant hoisting, common subexpression elimination, constant propagation, and deep inlining (see Section 2.3).

- Default compilation threshold: $\sim$10000.

2.2.2 Overview of Java Bytecode

When a Java program is compiled by a static compiler such as *Javac*, a portable and platform independent code is generated and saved in *.class* files. A bytecode file is a collection of step by step instructions representing the original program.
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. The first letter of an instruction indicates the type of operand it is operating upon, for example: `i` - *integer* and `a` - *a reference to an object*; there are also indicators for other types. See [5] for more details. The bytecode instructions relevant for this paper can be grouped into six groups:

- Load and store (e.g. `aload, istore`)

- Object creation (e.g. `new`)

- Control transfer (e.g. `ifeq, goto`)

- Getting a field of an object (e.g. `getfield`)

- Typecasting and exceptions (e.g. `checkcast, athrow`)
- Method invocation and return (e.g. `invokespecial, invokevirtual, areturn`)

For example, `istore_3` stores an `int` value into variable 3, `aload_1` loads an object reference onto the stack from local variable 1.
Furthermore we should explain the following four bytecode instructions:

- `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.
- `invokeinterface` is used for method calls on an interface variable. For instance, if `theList` is a variable of type `List`, the call `theList.get(i)` translates into `invokeinterface`. If the method invoked is not in the current object, superclasses of the current object are searched for the method.
- `invokespecial` is used when invoking a private method in a class or in other cases when the class is known at the static compilation time, such as when accessing a method via the keyword `super`.
- `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.

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.

2.3 Dynamic Program Optimizations

Within HotSpot's execution process a few things happen as the program is running, such as bytecode execution, profiling, and dynamic optimization [4]. While execution of the program is taking place, the JVM gathers profiling data which is used to determine the "hot" sections of code, i.e. those that are executed frequently. Once such a section is identified, the *JIT* can spend more time optimizing these "hot" sections. Profiling data and run-time program analysis allow the dynamic compiler to determine what optimizations to perform [2]. This dynamic process allows Java to generate code that is specifically optimized for the underlying hardware and for a specific usage of a program. We will be referring to this process as *dynamic optimization* [4]. Below we discuss dynamic optimizations that are particularly relevant to our work:

- *Devirtualization*: replacing a dynamic method lookup along a class hierarchy by a direct jump to the method when the target of the call can be uniquely determined [5].

- *Method inlining*: replacing a method call by a copy of the methods code when the call target can be uniquely determined (often follows method devirtualization) [5].
- *Deep inlining* and *inlining of potentially virtual calls*: method inlining combined with global analysis and *dynamic deoptimization* are used by both the server compilers to enable deep inlining (i.e inlining of methods which call other methods that are also being inlined) [5].
- *Dead code elimination*: removal of unneeded code that does not affect the actual program. For example if a method returns a result of a mathematical operation and this result is never used, then this instance of the method call is removed [5].

The JIT also performs optimizations more specific to Java technology, such as null-reference check, range-check elimination (eliminating unneeded checks for an array index positioned within the array), and optimization of exception throwing paths [9].

2.4 Implementation of Java Generic Types

Java generic types are implemented using *type erasure*. Type erasure is the implementation of generic types where only one compiled copy of a generic type definition exists in a program. All instances of the generic type refer to that same definition. For instance, if a program uses `ArrayList<String>` and `ArrayList<Integer>`, they both refer to the same definition of `ArrayList` – the one with the bound `Object`. At compilation time type correctness of generic types is verified and type casts are inserted for methods that return objects, such as `get`, as shown below.
`ArrayList` takes a generic parameter that extends `Object`. Java adds casts in bytecode to specific types that are returned from methods. It is shown in bytecode that there is in fact a type check when accessing a container with generic types. For example:

```
import java.util.ArrayList;
public class BytecodeExample {
    public static void main(String[] args) {
        ArrayList<String> alString = new ArrayList<String>();
        //add some elements to alString
        String exampleString = alString.get(3);
        System.out.println(exampleString);
    }
}
```

Below is a segment of the bytecode of the above example. On line 41 it shows that there is a cast check to string when accessing `alString.get(3)` to verify that it can be stored in the string container done on line 44 with `astore`. This corresponds to the above assignment statement `String exampleString = alString.get(3)`.

```
38 invokevirtual #30 <java/util/ArrayList.get>
41 checkcast #34 <java/lang/String>
44 astore_2
```

Other languages, such as C++, do not use type erasure. Consequently, there is a new instance of every type generated for every declaration where a new parameter type is used.

3 Specialization of Generic Types

3.1 Goals and Overview.

Our overall research is centered around an optimization of Java programs that we call *specialization of generic types*. The specialization aims at counteracting negative performance effects of type erasure (see Section 2.4). The key idea is to create a copy of a generic type with the type bound replaced by a more specific type, such as an actual type parameter. For instance, if a program uses `ArrayList<Integer>`, a separate specialized 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 practically the same as in `ArrayList` with a few differences that are not important here.
If a program additionally uses `ArrayList` with a different actual type parameter, e.g. `ArrayList<String>`, then a separate type `ArrayListString` with the type bound `String` is created. The specialization may also be applied to interfaces and parameterized static methods. The optimization changes the Java source code of the program.
The 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. In our tests we have observed program speed-ups of up to 20% due to specialization (see [7]).
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` type 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.

3.1.1 Partial and Complete Specializations

A straightforward way to perform specialization is to specialize the entire generic type hierarchy containing a given type. For instance, if we want a specialize `ArrayList`, we would also specialize all generic classes and interfaces that `ArrayList` inherits from, such as its superclass `AbstractList` and the interface `List` that it implements. We call such a specialization *complete*.
While a complete specialization provides the most specific type information for every type, it requires changes to large segments of a program, sometimes in parts of the program that are not related to the type in question. This approach duplicates a lot of program code since specialized versions of many classes need to be created. Handling these additional classes may potentially have a negative performance effect of its own.

An alternative approach is to specialize only a subset of classes and interfaces based on the actual usage. This approach leads to narrowing a type bound (see section 2.1.2) for at least some generic types, i.e. it creates cases when the bound of a class is more specific than the bound of a class that it extends or an interface that it implements. For instance, when specializing `ArrayList` to `Integer` bound we may leave the original `Object` bound on a `List` that `ArrayList` implements. We refer to specializations that specialize only a part of a class's hierarchy as *partial*.

3.2 `ArrayList` Hierarchy in the Java Collections Framework

In our research we use a subset of the *Java Collections Framework, or JCF*. JCF is an architecture that implements standard data structures for storing data, such as stacks, queues, maps, etc. As of Java 1.5, most classes in JCF are generic. All of these data structures are referred to as *collections*. All collections have some common operations, such as adding an element via a method `add`. This general functionality is specified in the interface `Collection<T>` that all collections implement.
More specific functionality may be given by sub-interfaces of `Collection<T>`, such as `List<T>` interface which requires additional methods that specify an index at which an element is located. For instance, method `get(int index)` returns an element at a given index.
To study the effects of the specialization, we used the JCF implementation of `ArrayList` class. We made our own copy of the source code of the entire `ArrayList` hierarchy and also created several copies of this code specialized in different ways. We used our own copy of `Integer` class with a slightly different name to guarantee that there are no inadvertent conversions between this class and a primitive type `int`. Just like a predefined `Integer` class, our version contains a single `int` value and implements the `Comparable` interface. As shown in Figure 1, `ArrayList` implements `List` interface and additionally extends `AbstractList` class. `AbstractList` in turn extends `AbstractCollection`. `Collection` interface is a supertype for both `List` and `AbstractCollection`. The hierarchy also includes `Iterable` interface.

3.3 Structure of the Program.

In our research we study runtime behavior of different methods in `ArrayList`, such as `get` and `add`, and the effect of type bound narrowing on their behavior. We started with a non-specialized version of the code and then created several specialized versions of the program (see Section 3.4) that all have the same structure as described here and differ only in the versions of the classes used. Examples in this section show original (non-specialized) code. Our testing program is a Java application that starts by running the method `main` which then calls a specific testing method based on a flag passed at the startup.

3.3.1 Main Class and `ListReader`

The main class reads necessary flags for a test run that indicate which specific test will be running, as well as the number of loops and other setup information. It creates `ArrayList`

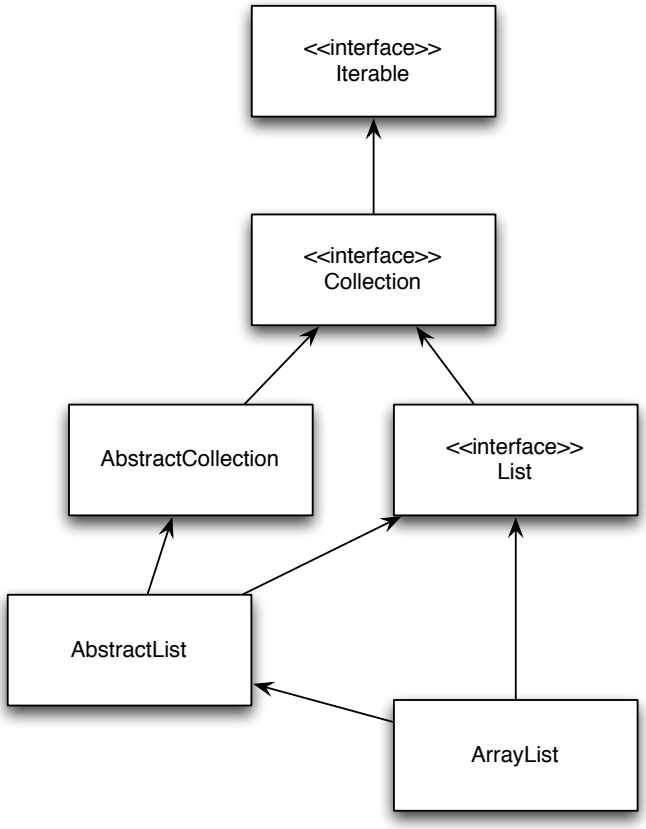

Figure 1: `ArrayList` hierarchy in JCF.

(or its specialized version). The array list is referenced via its interface `List`:

```
List<Integer> theList = new ArrayList<Integer>();
```

Elements are then added to the list. Since we would like to avoid effects of Java automatic Garbage Collection (GC), we use very few elements in the array (usually two) and iterate over them multiple times.
The main class also creates an instance of a class that we call `ListReader`. This is our own class, not a part of JCF. The `ListReader` has all our testing methods (see Section 3.3.2) and is itself generic. This class was originally designed to perform quicksort and therefore has a type bound `Comparable` to allow for calls to `compareTo`:

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

In some versions of the code the bound of `ListReader` is different, see Section 3.4. An instance of `ListReader` is created in the main class:

```
ListReader<Integer> reader = new ListReader<Integer>();
```

3.3.2 Testing Methods in `ListReader`

Testing methods in our approach are methods that repeat calls to a specific method in `ArrayList` (or in its specialized version) in a loop. Loops have very large values of

counters (for instance, a test commonly performs 400,000,000 calls to a given method, such as `get`). In our tests we compare the times of running the same test method in different versions of the specializations. The large number of calls allows us to accurately measure time differences between different versions of the code. It also guarantees that the startup time of the JVM is insignificant compared to the total program running time and that the compilation threshold is reached early in the test for both the client and the server mode (see Section 2.2).
All of our testing methods are located in `ListReader` class (or its specialized versions). Consider a testing method for `get` in `ArrayList`. `T` is the type bound of `ListReader` class (see Section 3.3.1).

```
public Object testGet(List<T> theList, 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 < 10000; j++) {
            objects[j % size]= theList.get(j % 2);
        }
    }
    return objects[r.nextInt(size)];
}
```

This method is called in `main` as

```
reader.testGet(theList, numLoops);
```

where `reader` and `theList` are as shown in Section 3.3.1.
It is important to observe that the `ArrayList` is passed to the test method not directly but as its interface `List`. This is a common practice in software design since this approach allows a programmer to replace a concrete class by another one, as long as it implements the same interface. No "client" code (i.e. code that uses the class) needs to be changed when this replacement takes place. The call `theList.get(...)` is a call on a variable of type `List`, and therefore compiles into the bytecode instruction `invokeinterface`. This is the most common pattern used in our tests. However, we also have test cases that pass `ArrayList` via its superclass `AbstractList` which results in `invokevirtual` bytecode instruction.
There are several issues that we need to consider when setting up testing methods. One issue is typecasting: if we assign an object returned from the `get` method to a variable of type `T` (that is more specific than `Object`), we will observe runtime slowdown due to the execution of `checkcast` bytecode instruction for each method call. Since the type `T` is different in different versions of `ListReader`, this check will affect these versions differently. Since our goal is to measure just the overhead of the call to `get`, we assign the result to `Object` array to eliminate typecasting.
Another important concern is avoiding dead code elimination (see Section 2.3). Thus we store the results in an array and then after the loop, it returns a random element from such

array. Because it is unpredictable which element will be chosen, the loop cannot be eliminated. The object array is small to avoid effects of GC. Note that later elements overwrite earlier ones in the array so earlier elements are actually useless, but the optimizing compiler is not sophisticated enough to optimize away earlier calls.

3.4 Eight Versions of Bound Narrowing.

Recall that the key element in all our tests is `ArrayList<Integer>` that is usually accessed via its supertype `List<Integer>` (see Sections 3.3.1, 3.3.2). We developed eight different versions of the specialization which specialize different subsets of the `ArrayList` hierarchy. All versions of the code follow the structure given in 3.3
Recall that a specialized version of a class means that its type bound is replaced with a more specific one (see Section 3.1). For instance `ArrayListInteger` is a specialized version of `ArrayList` which means that `ArrayListInteger` has a type bound `Integer` whereas `ArrayList` has a bound `Object`. Also note that `ListReader` has three possible bounds: `Comparable` in the original code, `Integer` in the specialized version, and also `Object` as a test with a more relaxed type bound (see **OO** and **ALO** below).
The following are the cases that we considered. Each version is denoted by a letter code that is used in the graphs of our results.

- **O** is the original code with no classes specialized.
- **OO** is the same as the original, except that the bound in `ListReader` is changed (relaxed) from `Comparable` to `Object`.
- In **S** the `ListReader` class is specialized, i.e. its bound is `Integer`.
- In **AL** the only specialized class is `ArrayList`.
- In **ALO** `ArrayList` is specialized class and the bound of `ListReader` is relaxed from `Comparable` to `Object`.
- In **ALS** combines AL and S: `ArrayList` and `ListReader` are specialized.
- In **LS** the interface `List` and all classes below it in the hierarchy (`ArrayList`, `AbstractList`) are specialized. For this versions to compile `ListReader` must be specialized as well. Some inner classes in `ArrayList` hierarchy also must be specialized.
- **C** is a complete specialization: the entire `ArrayList` hierarchy is specialized. Just like in **LS**, `ListReader` must be specialized for the program to compile.

In our tests we observed that in many cases the 8 versions of the specialization tend to group in the following way, based on their runtime behavior:

- **O**, **OO**, and **S**. We call it the **O**-group.
- **AL, ALO**, and **ALS**. We call it the **AL**-group.

- **LS** and **C**. We call it the **C**-group.

Note that the **AL**-group represents the narrowing of the type bound between the non-specialized `List` interface and a specialized `ArrayList` class.

4 Tests and Results

4.1 Testing methodology

All code and test results are stored inside a subversion repository. This allows for the test to be easily checked out and changes to the code recorded. All tests are run on the same machine in /tmp to ensure that network problems do not interfere with the tests.

```
Our Test Machine.
AMD AthlonTM 64 Processor 3200+
512MB DDR RAM
Fedora Core 7
Kernel: 2.6.23.17-88.fc7 SMP i686
Java Version: Sun JDK 1.6.16 and Sun JDK 1.6.17
Time Binary: GNU time 1.7
glibc version: 2.6
```

Our test platform uses ruby scripts to set up the environment, run the tests, record runtimes, and auto-generate from the runtimes. Runtimes are measured with the /usr/bin/time binary. Our scripts use *time -f %U* command piped to a file to calculate the program runtime. In addition we also measure *wall clock* time by recording the test start time and ending time and subtracting one form the other for the total runtime. Both times are compared, and if discrepancies arise on the time comparison, we use the wall clock time since it measures the overall runing time of the test program from beginning of the execution to the end. Our approach of using a stop clock methodology to gather runtime information is due to the fact that it is not possible to utilize a software profiler to get reliable runtime data without disabling dynamic optimizations [6].
We run all our test in three different JVM configurations:

- *Normal Mode*: The JVM does not receive any special flag. This enables the JVM to perform all dynamic optimizations.

- *Pure interpreted*: In pure interpreted mode the JIT is not running, and therefore all optimizations are disabled. This implies that the JVM will execute bytecodes instructions one at a time. This is accomplished by utilizing the JVM flag *-Xint*. Due to the slow operation of the JVM in the interpreted mode, we run our test with *5000* loop iterations as opposed to 40000 in the other tests.

- *No-Inline*: In this cases we run our test with the JIT turned on but with inlining disabled in order to better understand effects of inlning. For this case we used the JVM flag *-XX:-Inline* [6].

4.2 `ArrayList` Methods Being Tested

In this study we tested methods that are already defined in `ArrayList` (namely, `get`, `add`, and `set`) and also methods that we added to the `ArrayList` to test certain properties. In most cases methods that we test have the same bytecode in all eight versions of the program, but have runtime differences because of different bound narrowing patterns in different specializations. As an example, consider the method `get` in the Java collections library `ArrayList` class:

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

    return elementData[index];
}
```

Here `T` is the type parameter of the `ArrayList` class. Because of type erasure (Section 2.4) `T` gets compiled into the type of the `ArrayList` bound (`Object` in the non-specialized version, `Integer` is the specialzied ones). `elementData` is an array in `ArrayList` of type `T` that stored the actual objects. `RangeCheck` is a private method that checks whether the `index` is within the array. Recall that private methods are called via `invokespecial`.
The method `get` is compiled into the following bytecode instructions, extracted using *jclasslib bytecode viewer*. Our comments before each instruction explain its purpose. See Section 2.2.2 for details on bytecode.

```
//Load ArrayList:
0 aload_0
//Load the int index for the method RangeCheck:
1 iload_1
//Call to a private method RangeCheck:
2 invokespecial #36 <research/util/ArrayList.RangeCheck>
//Load a reference from an array for the operation in RangeCheck:
5 aload_0
//Get the elementData array:
6 getfield #12 <research/util/ArrayList.elementData>
//Load the index:
9 iload_1
//Load a reference to an element in the array:
10 aaload
//Return the reference to the calling method:
11 areturn
```

We tested `get` method in two different ways: accessing it via its interface `List` and via its superclass `AbstractList`.
The `get` method above returns an object of type `T`. We wrote a method similar to `get`, but returning a boolean, not an object:

```
public boolean getEqual(int index) {
if (index >= size)
   throw new IndexOutOfBoundsException("Index: " + index
          + ", Size: " + size);
return (elementData[index] == elementData[(index+1)%size]);
}
```

The method is set up in a way that prevents dead code elimination by comparing two different elements on every call. The code of `RangeCheck` is inlined directly into the method. In separate tests we observed that inlining `RangeCheck` by itself has the same behavior as `get` above so this inlining does affect our results.
The method handles array access in a way that is similar to `get`: it performs bytecode instructions `getfield` and `aaload`, just as in the bytecode for `get` above. The comparison == compares object references (memory addresses) so it works the same regradless of the type bound of `ArrayList`, i.e. it works the same in all versions of our program. The key difference between `getEqual` and `get` is `ireturn` (returing a primitive type, i.e. a boolean) in the former instead of `areturn` (retunring an object) in the latter. As we show in the following sections, this creates a substantial difference in bytecode behavior.
We also tested standard `ArrayList` methods `add` and `set`. Due to a lack of space we are not showing their code.

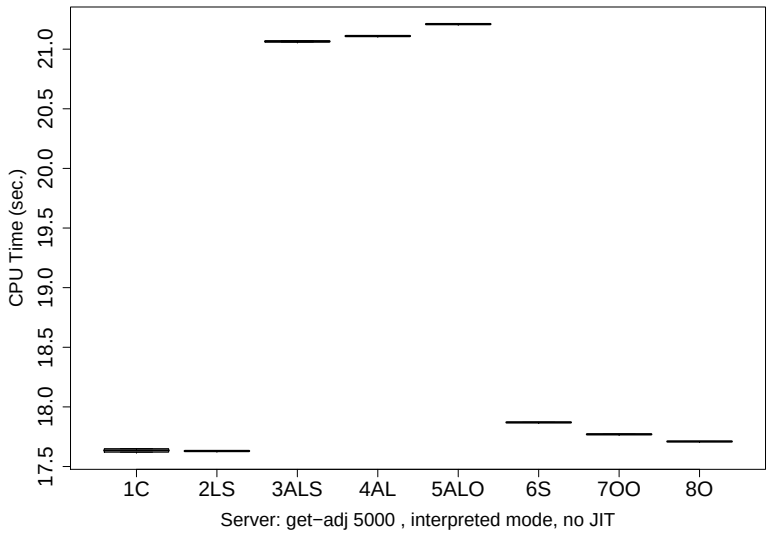

Figure 2: Testing `get`method, interpreted mode in server JVM.

4.3 Tests in Interpreted Mode

In this section we discuss results of testing our eight versions of bound narrowing in a pure interpreted JVM mode, i.e. with the JIT turned off. In this case the client and the server modes of the JVM behave identically so we only show results for server. As explained on

Section 3.4, we can see a distinct grouping based on the specialization of the program code. The graphs in Figures 2 and 3 show the results of testing `get` and `getEqual` methods (described in Section 4.2), respectively.

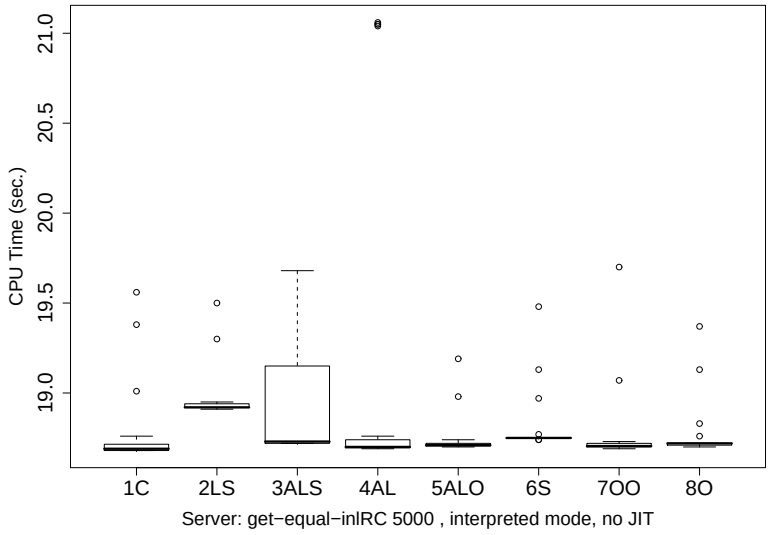

Figure 3: Testing `getEqual` method, interpreted mode in server JVM.

After all the eight class were compiled, the resulting bytecode for the methods were exactly the same. However at the execution time the way in which some of the methods were handled by the JVM were not the same. As shown in Figure 2, the **O**-group which does not have a bound narrowing performs as well as the **C**-group which is fully specialized. In either case when the method call is executed an `Integer` is returned and either the fully specialized or not specialized tests are performed in the same manner by the JVM. In these two groups the interface and the `ArrayList` have the same type bound.
In the case of the **AL**-group, which has a mixed type specialization, where the bounds of `ArrayList` and `List` are mismatched (`ArrayList` bound is more specific than `List` bound), we can see a drastic slow down. We argue that this slowdown is due to an implicit typecheck being done by the JVM when an object is passed through a narrowed type bound. In a test case when we access the `ArrayList` via its superclass `AbstractList`, rather than an interface `List`, we observe a similar slowdown for the **AL**-group, compared to the other two group. This is due to the same bound mismatch, even though the call is done in the bytecode as invokevirtual. Also the same behavior is observed for methods `add` and `set`, since they both take an object as a parameter.
However, the above grouping does not happen when we call the method `getEqual`, see Figure 3. Since this method returns a boolean which is a primitive type, the implicit type check done by the JVM is not longer needed. This lack of implicit check is evident as we observe no significant difference in runtime for all our tests.

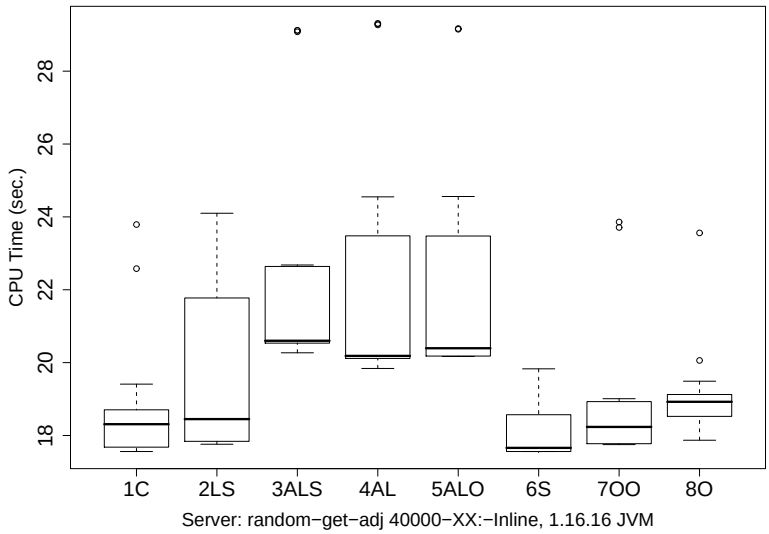

Figure 4: Testing `get` method, no inlining in server JVM.

4.4 Tests with Inlining Turned Off

In this section we look at how our test cases behave while running with the JIT, but with the inlining optimization disabled. The graph in Figure 4 shows the runtime behavior of the method `get`. Note that the absolute running times for this test are not comparable with the interpreted runs because we are performing 40000 iterations of the outer loop in our testing code instead of 5000 in the interpreted mode runs.
This data shows a similar pattern as the previous interpreted mode. The same patterns with different runtimes are also observed for `add` and `set` test and for accessing `get` via `AbstractList`. This pattern is present in both client and server modes of the JVM, although with different absolute times. We hypothesize that the typecheck in **AL**-group performed in the interpreted runs cannot be eliminated when the inlining is turned off.

4.5 Tests in Regular Mode

In this section we look at three type of test results.

- `get method`: Figure 5 shows a normalization pattern in all the test runs: all versions of the code run in about the same time. Note that the ranges of times are very small in each of these runs: 17.4-17.5 sec in server and 26.34-26.38 in client. Although type check in the **AL**-group tests must be performed prior to the actual call getting inlined, the inlining process renders the type check unnecessary once the compilation threshold is reached. This is observed in both server and client JVMs.

- `add method`: Figure 6 shows the `add` method tests. This new runtime grouping shows `ALS` performing closer to the **C**-group, and the **O**-group performing closer to

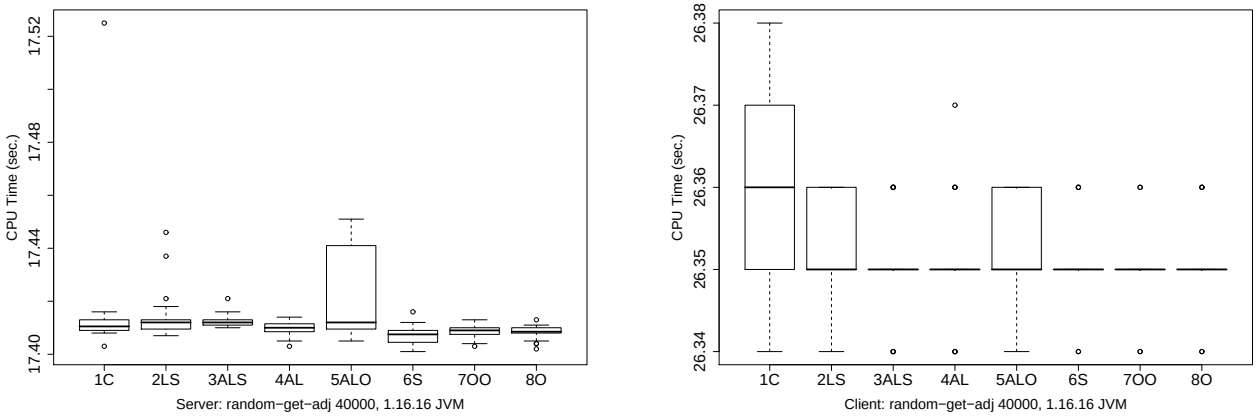

Figure 5: Testing `get` method, normal mode in client and server JVM.

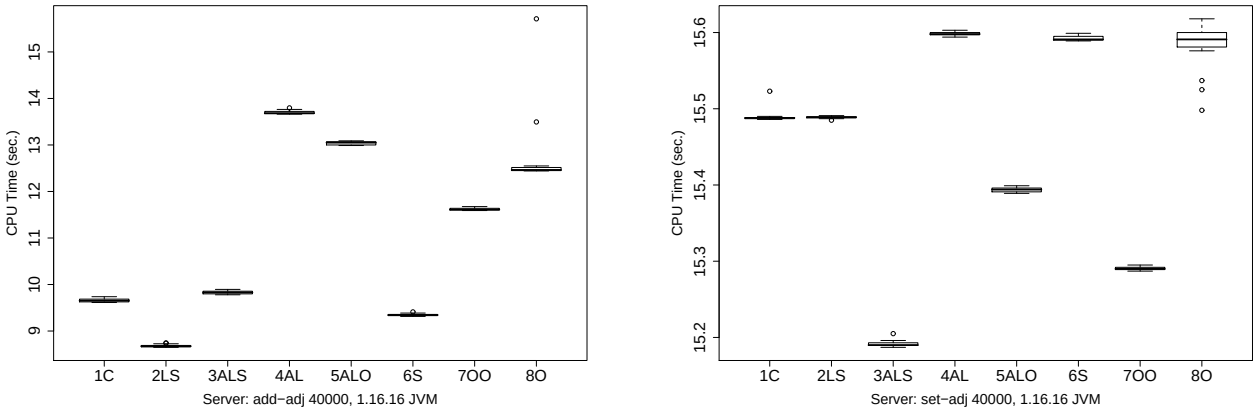

Figure 6: Testing `add` and `set` methods, normal modes in server JVM.

AL-group. Once inlining is turned on, the more specialized the hierarchy, the better the program tends to behave. However this does not explain the behavior of `S` and some observed runs in the client mode.

- `set method`: The same argument can be apply to the `set` method, which has the added complexity of not only setting an element at a given index, but also returning the old object at that index.

5 Conclusion and Future Work

We discovered that narrowing the type bound leads to noticeable runtime inefficiencies in interpreted mode (i.e. with the JIT turned off) in cases when an object is passed to or

from a generic container. This inefficiency does not take place for methods that take and return only primitive type. This led us to conclude that the slowdown is due to a typecheck performed by the JVM to make sure that only objects of the correct type are passed to and from the methods.
We also observed that in the absence of inlining the JVM does not eliminate the typecheck overhead entirely. When inlining is turned on, in some methods (such as `get`) the overhead is eliminated completely, but for other methods (such as `add`) it is not offset by dynamic optimizations. While there is a clear connection between this behavior and parameters passed to the method, more research is needed to pinpoint and explain the exact dependency. We also plan to use a JVM called Jikes for testing purposes and compare the results.
One should note that, although our test cases stem from the study of generic types specialization, the results are applicable to any software design pattern that uses type bound narrowing.

References

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

[2] GOSLING, J., JOY, B., AND STEELE, G. L. *The Java Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[3] KOTZMANN, T., WIMMER, C., MÖSSENBÖCK, H., RODRIGUEZ, T., RUSSELL, K., AND COX, D. Design of the java hotspot™client compiler for java 6. *ACM Trans. Archit. Code Optim. 5*, 1 (2008), 1–32.

[4] LINDHOLM, T., AND YELLIN, F. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

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

[6] MACHKASOVA, E., ARHELGER, K., AND TRINCIANTE, F. The observer effect of profiling on dynamic java optimizations. In *OOPSLA Companion* (2009), pp. 757–758.

[7] 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.

[8] SUN DEVELOPER NETWORK. The java hotspot performance engine architecture. *Sun Microsystem* (2007).

[9] SUN DEVELOPER NETWORK. The java hotspot™server vm. *Sun Microsystem* (2008).