# On the Algorithm for Specializing Java Programs with Generic Types

Daniel Selifonov, Nathan Dahlberg, Elena Machkasova
Computer Science Discipline
University of Minnesota Morris
Morris MN, 56267
selif004,dahlb061,elenam@umn.edu

## Abstract

Java 5.0 added classes with a type parameter, also known as generic types, to better support generic programming. Generic types in Java allow programmers to write code that works for different types, with the type safety checks performed at compilation time.
Generic classes in Java function by type erasure. Type erasure works by creating a single instance of the generic class, removing all type-specific information from the generic class, and inserting typecasts to guarantee type-safe calls to instances of the generic class.
The selection of the type erasure strategy when implementing the Java generics functionality meant that very few changes were necessary to the Java virtual machine. However, type erasure precludes dynamic optimizations that would have been possible if type information was preserved until run-time. Since most of the optimizations in the Java programming language are performed at run-time, Java programs using generic classes are slower than those that use type specialized classes.
In this paper we propose and discuss an optimization of Java programs that we call *specialization of of Java generic types*. The specialization selectively produces separate copies of generic classes for each type used in the program. This reduces the number of time consuming typecasts and dynamic method lookups. The optimization produces up to 15% decrease in program's run time. We discuss conditions under which the specialization can be performed without changing programs' behavior.
We present an algorithm that allows one to get substantial program's speedup with relatively few changes to the program. Using a quicksort sorting procedure as a benchmark, we compare the result of such specialization, which we call *minimal*, with the orginal non-optimized program and with the version of the program where all generic classes are specialized.

# 1 Introduction

Generic, or parameterized, types were added to the Java programming language in version 5 (also known as 1.5). Generic types, or generics for short, allow a programmer to write classes and functions that work with objects of different types without specifying the actual type, instead denoting the type by a type variable. Some proposals on adding generics to Java include [1, 3].

Generic style of programming was possible in Java before generic types. However, the addition of generics made it more convenient for programmers to write generic code. Even more importantly, it shifted the error-detection in such code from run time to compilation time, allowing earlier and easier error detection and providing type safety guarantees (i.e the absence of certain run-time errors) for programs that successfully compiled.

Generic types in Java are implemented using an approach called type erasure. In this approach a compiler checks the type correctness of all uses of generic codes. After the type usage is found to be correct, the compiler discards type-specific information and generates one copy of the code based on the most general type compatible with the generic class or function.

The type erasure approach does not duplicate code unnecessarily because only one copy of the code is created. However, this approach creates slower program because type-specific information that could have been used for type-specific run-time program optimization is discarded at compilation time.

In this paper we present an optimization that we call *specialization of Java generic types*. We developed the idea of the optimization in 2006. Our earlier work on the specialization is described in [2, 4].

We use a QuickSort implementation that uses Java collections as our benchmark. We perform two kinds of optimization for this example: the minimal optimization that specializes only the classes where we are likely to obtain a noticeable performance increase because of specialization, and a complete optimization that specializes all possible classes. We describe the algorithm for the minimal specialization.

We compare the running times of the two versions of the program (specialized with the minimal specialization and with the complete one) to each other and to the original program. We observe that the complete optimization results in up to 25% program speedup and the minimal optimization gives up to 28% speedup of the relevant program fragment. Thus the minimal optimization produces substantial efficiency increase with much less code duplication.

# 2 Overview of Java Type System

There are two kinds of types in Java, primitive types (such as integers and booleans) and object types (such as strings and arrays). Classes are a kind of *object type*. Every object in Java belongs to a *class*. Classes describe types of objects by specifying types and names of their internal data (also known as *fields*) and their functionality defined by methods.

## 2.1 Java Class Hierarchy

Like most object-oriented languages, Java supports *inheritance* between object types. This means that a class in Java can be declared to inherit from another class in which case it can use all methods of that class without redeclaring them. For example, if class A contains a method `m` and class B inherits from class A (or, following the Java syntax, B *extends* A), then `m` can be called on an object of class B because of inheritance.

All objects implemented in Java are part of the Java class hierarchy. If a class is not declared to inherit from any class, it implicitly inherits from a class called Object which is at the top of the class hierarchy.

Each class in Java can only inherit directly from a single other class, but can inherit from many more indirectly. For example, if class A inherits from class B and class B inherits from class C, then class A directly inherits from class B and indirectly, through class B, inherits from class C. This means that all classes inherit from the Object class, since it is at the top of the class hierarchy. If a class A inherits from a class B (directly or indirectly), then we say that A is a *subclass* of B and B is a *superclass* of A.

A subclass, such as class A in the previous example, may contain its own implementation of a method defined in a superclass, such as class B. In this case we say that a method in A *overwrites* or *overrides* the corresponding method in B. This happens when the two methods (one in A and one in B) have the same name and argument types. Any call of the method on an object of a class A will then call the method defined in A, while any call to the corresponding method on an object of the class B will still call the method defined in B.

If a class inherits a method but does not implement it, then any calls to that method will invoke the method in the closest (via the inheritance chain) superclass that does implement it. For instance if class B contains a method `m` and class A is a subclass of B that does not define a new method `m`, any calls to `m` on class A will call the method `m` in class B. If B inherits from C that also defines `m`, the method of B will still be called for an object of the class A since B is closer to A in the inheritance chain. The process of determining the right method to call is performed at run-time and is called *dynamic method lookup*.

## 2.2 Interfaces

The Java type hierarchy includes interfaces in addition to classes. An interface is similar to a class except it only declares but does not implement any methods. It lists method descriptors (names, return values, and the types of parameters) without any implementations, such as in the following example:

```
interface List {
    ...
    void clear();
    ...
    int size();
    ...
}
```

All classes that implement an interface must implement all of the methods in the interface. This means that any class that implements `List` will have to implement `clear()` and `size()`. Implementing these classes also require that the implemented methods have the same parameters and return types.
Classes are allowed to implement more than one interface, in which case they must implement all methods in all the interfaces. A class that implements an interface is referred to as a *subtype* (or sometimes as a *subclass*) of that interface.

# 3 Java Compilation Model

Traditionally many programming languages are compiled into a set of platform-specific CPU instructions. The majority of Java compilers follow a different model: they convert programs into platform-independent sets of instructions called *bytecode* which are then run by a Java interpreter referred to as the *Java virtual machine* or JVM. This part of the Java compilation process is called *static compilation*. It allows Java to compile independently from the platform it is running on which allows for runtime safety and platform independence.
Originally JVMs interpreted the bytecode, but since this was slow, JIT (*just-in-time*) compilers were added to JVM to compile the bytecode to native code as it is run. Since JIT compilers add a lot of overhead to programs because they have to compile and optimize code at runtime, Sun Microsystems developed HotSpot™JVM which decreases this overhead by profiling the program being run to selectively compile only frequently called methods into native code. This means that time is not spent performing expensive compilation and optimization of code that will not affect the runtime of the program. This compilation of bytecode to native code at run time is called *dynamic compilation*.
There are two ways of running the HotSpot JVM: client and server mode. The client mode compiles a program using a simple and fast JIT compiler. It starts the program as quickly as possible without heavy optimizations. In the server mode the dynamic compilation takes longer and focuses more on optimizing the runtime of the program, performing more effective but time-consuming optimizations. As the result, long programs often (though not always) run faster in the server mode because of the optimizations performed by the JVM.

# 4 Overview of Generic Types

Many modern programming languages, including Java, are *strongly typed:* they require that types of all variables are explicitly declared by the programmer. The type compatibility is checked by the compiler. Many of such languages have a feature called *generic types*. They allow a programmer to write code for a data structure or function that works with different data types. The type used in these data structures or functions is specified as a *type parameter*. This type parameter is represented by a *type variable*. For example, a linked list would be declared like this as a generic class:

```
LinkedList<K>
```

Here `K` is the type variable that represents the type parameter.
When using a generic data structure or function, the programmer specifies a concrete type for the type parameter. In this case, replacing `K` with a concrete type:

```
LinkedList<Integer>
```

This declaration would create a `LinkedList` of `Integers` and is called *type instantiation*. `LinkedList` could then be instantiated in the same program using a different concrete type, such as a `String`. The `LinkedList` of strings would then only contain strings, and the `LinkedList` of integers would only contain integers.
The use of generics makes code more reusable and eliminates unnecessary code repetition, such as writing separate implementations of a list for integers and strings. Unlike other ways of writing generic code (i.e. code that works for different types), generic types also provide type safety guarantees, as described in section 4.4.
Generic types in Java where added in Java release 1.5 (Java 5). They were implemented using an approach called *type erasure* (see section 4.3 for details). One of the reasons for choosing this implementation is so that older Java code would still be usable with Java 5.

## 4.1 Use of Generic Classes in Java

In a parameterized class, the type parameter is put in angle brackets in the class declaration. Within the scope of the class the parameter given in the angle brackets may be used just as any type would be used. For instance, it can be used as a method parameter or as a return type.
The methods in the class refer to the type parameter as they would to any concrete type. As an example, consider the following class declaration:

```
public class LinkedList<K> {
    ...
    public K get() {...}
    ...
    public void add (K newItem) {...)
    ...
}
```

Here `K` is the type parameter passed to this class upon instantiation. `K` is then used as a return type for the method `get` and as a parameter for `add`. This means that `get` returns an object of a type `K` and `add` takes such an object as a parameter.
An *upper bound* (or simply a *bound*) of a generic type can be used when writing parameterized code to restrict the range of types allowed to substitute for the type parameter. A class or an interface can be set as an upper bound so that only subtypes of that bound can be passed as a type parameter. The keyword `extends` is used to set a bound. For instance:

```
public class LinkedList<K extends Number> {...}
```

Any class that implements the interface `Number` will be a possible type parameter for `LinkedList`.
When an instance of such a class is created, a concrete value that is a subtype of the upper bound of must be specified as the type parameter. For example, the following is a valid instantiation of the `LinkedList` class:

```
LinkedList<Integer> testList = new LinkedList<Integer>();
```

The compiler checks to make sure that the concrete type, `Integer` in this case, is a subtype of the upper bound of the type parameter, in this case `Number`. The code will fail to compile if this is not the case.
The compiler guarantees that a class passed as a type parameter is a subtype of the class or an interface specified as a bound. Thus any method defined for the bound class or an interface is defined for any class passed as parameter in any instance of the generic type. Therefore bounds are used when a generic class requires that a specific method is called on objects of the class represented by the type parameter.
For example, if a method `intValue()` is called in a generic class on the objects of the parameter type `K` then the programmer specifies the upper bound `Number` since `Number` has the `intValue()` method, hence all of its subclasses have that method as well.
A parameterized class with no upper bound specified is in fact a class with an upper bound `Object`.

## 4.2 Type Recursion: `Comparable<T>`

Java allows a programmer to use a type parameter recursively within declarations of generic types. A common example of this is the use of the generic interface `Comparable<T>`. The interface requires any class that implements it to provide a method

```
int compareTo(T obj)
```

If a class A implements the interface, the `compareTo` method is used to compare two objects of the class A to each other according to some order. If `a` and `b` are two objects of the class A then `a.compareTo(b)` returns a negative number if `a` is less than `b` according to that order, a positive number if `b` is less then `b`, and zero if the two objects are equal. Classes `String` and `Integer` both implement the interface. Integers are compared by their value and strings are compared lexicographically.
It makes sense to compare objects of a given class only to objects of the same class or its subclasses. For instance, strings cannot be compared to integers. However, before generic types in Java the only way to ensure that objects were properly compared was to include a typecasting operation that would give a run-time error if the object passed to the method was of the wrong type. Adding a type parameter to `Comparable` interface allows this check to be performed at compilation time which leads to earlier and reliable error-detection.
The Java 5 version of the interface leads to a recursive style of declaring generic classes for comparable objects. The following is a declaration of a generic priority queue class:

```
PriorityQueue<T extends Comparable<T>>
```

Objects are retrieved from the queue according to the order defined by `compareTo` method. Note that the type parameter `T` is used recursively in the type declaration `T extends Comparable<T>`. The first occurrence of T names the parameter, and the second occurrence (the one in `Comparable<T>`) refers to the same T. Thus, the type declaration means that objects of type T are comparable to other objects of type T.

## 4.3 Implementation of Generic Types

There are two basic ways to implement generics in modern programming languages. The first one is a *template* approach used by C++. The second approach is called *type erasure*, and is used by Java.

The template approach creates a separate copy of the data structure or function for each instantiation of the data structure or function that uses a different data type. This means that there will be several copies, one for each type used in the program, of the data structure or function in system memory while the program runs.

Type erasure, on the other hand, makes only one copy of the code. This copy is not specific to any type instance. Type erasure occurs during static compilation. When the program compiles, the compiler checks the type compatibility and after that it removes all instance-specific type information and replaces it with the type bound. If there is no upper bound specified, the upper bound is `Object`. Since specific type information is removed, type casts are inserted by the compiler for objects returned from generic methods and for some other instance-specific uses of the type. For example:

```
public class LinkedList<K> {
    ...
    public K get() {...}
    ...
    public void add (K newItem) {...}
    ...
}
```

The static compilation would generate bytecode for this program as if the the program was written like this:

```
public class LinkedList {
    ...
    public Object get() {...}
    ...
    public void add (Object newItem) {...}
    ...
}
```

All calls to `LinkedList` would then be typecast. For example, suppose a programmer writes the following code:

```
public void static main (String args[]) {
    ...
    LinkedList<Integer> newList = new LinkedList<Integer>();
    ...
    Integer n = newList.get();
    ...
}
```

This would be compiled to the bytecode equivalent to:

```
public void static main (String args[]) {
    ...
    LinkedList newList = new LinkedList();
    ...
    Integer n = (Integer) newList.get();
    ...
}
```

The object returned from `get` is cast to the instance type, in this case an `Integer`, by the automatically inserted cast.
Templates allow for shorter runtimes because type-specific information allows the compiler to generate more efficient code and to perform type-specific program optimizations. They are somewhat less memory efficient and less convenient for programmers because of creating multiple copies of the code. On the other hand, type erasure is more memory efficient but tends to produce longer runtimes. This is because erasing type information at compile time does not allow further type specific program optimizations.

### 4.4 Compile Time Type Checking for Generics

In versions of Java before generic types a programmer also could write a "generic" collection (say, a list). However, such a collection had to be written to contain elements of type `Object` (or another most general type of the "generic" collection). Since any type is a subtype of `Object`, an instance of such a list could be used for any type.
However, one could not be sure that all objects in a generic collection were of the same type. This meant that whenever an element was removed from a collection, it had to be explicitly cast into its real type. For instance, if a list was used for strings, the programmer had to insert code to typecast objects removed from the list to the type `String`. There was no possibility to check the correctness of typecasting at compilation time, thus the only possible indication of an incorrect type use was a run-time `ClassCastException`.
With the introduction of generics, collections have a declared type that can be checked at compile time. Since this happens at compile time instead of runtime, the type casts are guaranteed to be safe. This feature is an important element of type safety of Java generics.

## 5 Specialization of Generic Types

We describe the proposed optimization and discuss reasons for its efficiency.

## 5.1 Description of the Optimization

When the Java compiler performs type erasure on parameterized classes, it replaces instances of the type parameter with the bound. Typically the bound is less specific than the actual instance type because of the generic nature of the parameterized classes. Since the bound is less specific than the type of a particular instance, the Java Virtual Machine must perform computationally expensive method lookup when calling methods on objects represented by the type parameter.

## 5.2 Specialization of Generic Type Bounds

To mitigate the performance inefficiency of generic types, it is possible to *specialize* the generic classes. The specialization approach used for this optimization produces duplicates of the generic class, and changes the parameterized variables to a specific concrete class.

For example, the given `LinkedList` example can be specialized from the generic version:

```
public class LinkedList<K> {
    ...
    public K get() {...}
    ...
    public void add (K newItem) {...}
    ...
}
```

To an Integer specialized version:

```
public class LinkedListInteger<K extends Integer> {
    ...
    public K get() {...}
    ...
    public void add (K newItem) {...}
    ...
}
```

Which is type erased to the Integer bound during static compilation to produce:

```
public class LinkedListInteger {
    ...
    public Integer get() {...}
    ...
    public void add (Integer newItem) {...}
    ...
}
```

### 5.3 Effects of Type Specialization

There are two sources of increased program efficiency due to specialization of type bounds: elimination of typecasts and elimination of runtime overhead of dynamic method lookup.
*Dynamic lookup* is a necessary component of method invocation in an object-oriented language. It is possible that multiple versions of a method could exist in superclasses and in subclasses. During the dynamic method lookup the correct version of the method is found for the specific object instance on which the method is invoked. With the specific type information lost as part of typical type erasure, the dynamic lookup cost is comparatively high.
However, it is possible to reduce the dynamic lookup cost by specializing the bound of a parameterized class to be the same as the types used in the instances of the class. As the result, the type erased version of the parameterized class would not discard specific type information, there would be no need to include additional type casts, and method lookup becomes much less computationally expensive. The cut-down of the method lookup time in cases when the method address can be determined exactly is called *devirtualization*. By duplicating the class and specializing the bound we provide enough information to the dynamic compiler to devirtualize method calls.
Since a parameterized class can be instantiated with multiple type parameters, it is necessary to produce multiple copies of the parameterized class, specialized with each of the individual instance type bounds used in the program.

## 6 Conditions for Behavior Preservation

Program optimization must be performed only in cases when they do not alter behavior of a program. There are some cases when bound specialization cannot be performed, at least not automatically, as it would have a potential for changing the program's behavior. In these cases it may be left up to the programmer to decide whether they would consider the specialization to be safe.
Since this method of specialization produces specialized duplicates of classes with different names, the specialization cannot be performed on any class that relies on the class name. Thus, a class used in Java reflection or a class that may be dynamically reloaded cannot be specialized.
Specialization cannot be performed if a class has typecasting of an object to the type variable: since the type variable gets replaced by the type bound during the type erasure, tightening the bound may change a successful typecasting to a run-time error.
Serialization refers to writing out object's internal data to a file for later retrieval. A class whose objects can be serialized must be implement `Serializable` interface. Serialization of objects may present a problem similar to that of dynamic loading or reflection: since a serialized object contains the class signature, changing the class name via the optimization will make the object incompatible with the non-optimized version of the class. However, if the object is read back by a program that is optimized the same way, there should be no issues with serialization. The optimization must choose a new `serialVersionUID` for an optimized class to avoid confusion with the non-optimized class. One should also note

that serialization creates an overhead of its own. If performance of a program is an issue, alternatives to serialization (such as directly writing data to a file) should be considered.

# 7 Algorithm

## 7.1 Quicksort

To demonstrate the effectiveness of our specialization algorithm, we selected a benchmark example that was heavy on dynamic lookup: a generic quicksort implementation. Many algorithms require sorting, thus many programs use sorting in the context of the algorithms that they implement. Quicksort is one of them most commonly used sorting algorithms, therefore our example is an element of many real-life programs.
Quicksort is among the fastest general purpose comparison sorts. The average case time complexity for sorting a list of $n$ elements is $\Theta(n * \lg(n))$ comparisons. The Java programming language `Comparable` interface is used to determine natural ordering of objects of mutually comparable types, as described in Section 4.2.
With the Comparable interface in mind, it makes sense to produce a Quicksort class that can operate on any type whose elements can be compared to each other. To that end, the following class was produced:

```
public class Quicksort<T extends Comparable<T>> {
    void sort(List<T> in) {
        quickSort(in, 0, in.size() - 1);
    }
    void quickSort(List<T> in, int frst, int last) {
        if (frst < last) {
            int mid = partition(in, frst, last);
            quickSort(in, frst, mid - 1);
            quickSort(in, mid + 1, last);
        }
    }
    int partition(List<T> in, int frst, int last) {
        T piv = in.get(last);
        int sbp = frst - 1;
        for (int i = frst; i < last; i++) {
            if (in.get(i).compareTo(piv) <= 0) {
                sbp++;
                T tmp = in.get(sbp);
                in.set(sbp, in.get(i));
                in.set(i, tmp);
            }
        }
        T tmp = in.get(sbp + 1);
        in.set(sbp + 1, in.get(last));
        in.set(last, tmp);
```

```
        return sbp+1;
    }
}
```

This `Quicksort` class accepts a list of any self-comparable objects and uses a method invocation to compareTo in order to determine natural ordering. Due to type erasure, all of the compareTo invocations are on the type Comparable, thus requiring a computationally expensive dynamic lookup to find the correct method code.
For testing purposes, the `Quicksort` class is utilized by a runtime measurement class called `MeasureRuntimes`. `MeasureRuntimes` sorts twelve `Lists` (six of Integers, and six of Strings) using the `Quicksort` class, and reports the fragment sort time. The differences in the fragment sort time are reported in results (see Section 7.4) and were used to guage the effectiveness of the optimization through specialization.

The `MeasureRuntimes` class utilizes the `List` and `ArrayList` types for storing the comparable elements. `List` and `ArrayList` are two component types of the Java *Collections* library. Collections contains many types of data structure implementations, and is commonly used in many Java programs. As part of this specialization research, we traced the classes related to Java Collections connected to the use of `List` and `ArrayList` in the Quicksort example.

Eight types were traced in connection to use of Collections in the Quicksort example.
Interfaces:

- Iterable
- Collection (extends Iterable)
- List (extends Collection)
- Iterator
- ListIterator (extends Iterator)

Classes:

- AbstractCollection (implements Collection)
- AbstractList (extends AbstractCollection, implements List)
- ArrayList (extends AbstractList, implements List)

These types were specialized in the *complete* optimization to all non-recusive type instances.

## 7.2 Definitions

In this section we define notations necessary for specifying the algorithm.

- Static Data - A Java class can contain *static* data. Static data belongs to the class, and is shared by all instances of a class. For example, a class can keep track of the number of instances by utilizing a static counter that is incremented whenever an object of the type is constructed.
- Type-Erased Operations - A type erased operation is an operation that is made generic before static compliation. For example, a method that intentionally takes a type-erased parameter (such as `Object`) instead of a generic type parameter; a common example is the `equal` method, which takes an `Object`.
- Nested (Type) Instance - A nested type instance can occur when a generic parameterized class has a sufficiently wide bound that allows the parameterized class to be instantiated with itself as a type parameter. For example, a `List` of `Lists` of `Integers` can be typed as `List<List<Integer>>`.

## 7.3 Algorithm Description

The algorithm is a whole program optimization that starts from the program's entry point main function. *Whole program optimizations* are optimizations that need to analyse the entire program before optimizing.

### 7.3.1 Algorithm for Minimal Optimization

A straightforward implementation of bound specialization would specialize all generic classes that are used with a particular concrete type. For instance, in the case of the quicksort program all generic classes instantiated with the class `Integer`, such as `Quicksort, ArrayList,` , and all of the classes in the collections subset described in Section 7.1 would be specialized. However, since the most substantial optimization benefit comes from devirtualization, it makes sense to optimize starting from the class that performs method calls on the objects whose type is represented by the type parameter.
It turns out that in many cases it is sufficient to optimize only such classes in order to obtain substantial benefits in the server mode of HotSpot JVM (see results in section 7.4). While it may be needed in some cases to optimize programs that run in the client mode, generally longer programs are expected to run in the server mode to take full advantage of dynamic optimization.
An optimization that specializes only classes that directly use objects that belong to the classes represented by the type parameter and leaves the rest of the generic hierarchy unspecialized is called *minimal optimization*. Below we present the algorithm for minimal specialization. The algorithm also checks the conditions for the minimal specialization to preserve the program's behavior.

1. Make a list of all non-nested instances of parameterized classes, based on the calls to constructors.

2. For each class on the list, make a list of methods called on the type parameter.

3. For each pair on the list, check if the method is overridden from the method in the type bound. Remove items that do not have any overridden methods.

4. Check for classes on the list for static data, or "type-erased" operations. Remove those.

5. For each class C on the list with a parameter T:

   - Find all parameterized classes whose parameters are within the scope of T.
   - If all classes generated by the previous step have non-nested bounds, specialize C.

### 7.4 Results

| *Type* | *Time* | *Percentage* |
|---|---|---|
| Complete | 23.96 sec | 5.48% faster |
| Minimal | 24.04 sec | 5.17% faster |
| Original | 25.35 sec | - |

Table 1: Client Times

| *Type* | *Time* | *Percentage* |
|---|---|---|
| Complete | 11.14 sec | 25.18% faster |
| Minimal | 10.73 sec | 27.94% faster |
| Original | 14.89 sec | - |

Table 2: Server Times

Results were obtained by running the three different versions of the program five times and recording the fragment time for the sorting operation. In both optimized versions, the Quicksort class is the location in which the devirtualization provided the most profound increase in program execution speed. We measured the sorting fragment time in this specific instance; in our experience, the total program execution time tends to be very close to the fragment time.

## 8 Conclusions and Future Work

We observe that for the QuickSort test example specialization of generic types leads to a substantial efficiency increase: up to 5.5% in the client mode of the JVM and up to 28% in the server mode for the relevant fragment of the program. The minimal specialization produces results comparable, and for the server JVM, even better than the complete specialization.

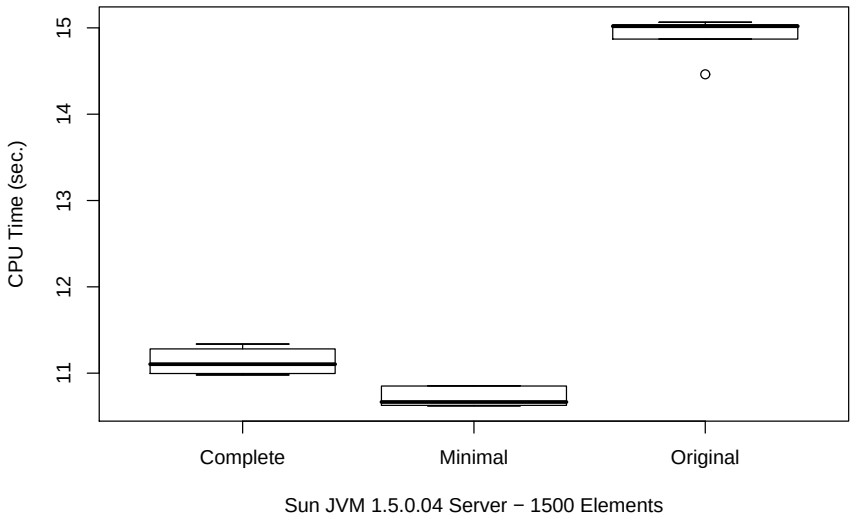


Figure 1: Server Fragment Time Results

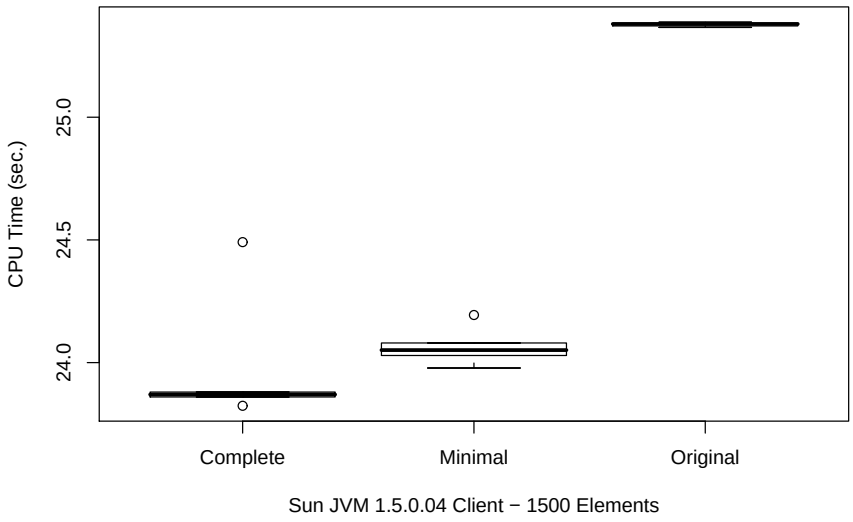


Figure 2: Client Fragment Time Results

Since the minimal specialization copied only one class (namely, QuickSort), the results show that it is sufficient to duplicate a small portion of a program to obtain full benefits of specialization. This makes specialization of type bounds of generic types a promising candidate for Java optimization.
Our future work is to further develop the algorithm for specialization and to study possibilities and effects of various partial specializations of generics. Our ultimate goal is to develop a tool for automatically specializing generic types.

# References


[1] ALLEN, E., AND CARTWRIGHT, R. The case for run-time types in generic java. In *PPPJ '02/IRE '02: Proceedings of the inaugural conference on the Principles and Practice of programming, 2002 and Proceedings of the second workshop on Intermediate representation engineering for virtual machines, 2002* (Maynooth, County Kildare, Ireland, Ireland, 2002), National University of Ireland, pp. 19–24.

[2] BEVIER, S., AND MACHKASOVA, E. Specialization of java generic types. *UMM Working Papers Series 2*, 1 (June 2006).

[3] BRACHA, G., ODERSKY, M., STOUTAMIRE, D., AND WADLER, P. Making the future safe for the past: adding genericity to the java programming language. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (New York, NY, USA, 1998), ACM Press, pp. 183–200.

[4] LEMBCKE, S., BEVIER, S., AND MACHKASOVA, E. Specialization of java generic types. *Midwest Instruction and Computing Symposium* (April 2006).