Specializing Generic Java Data Structures

Dan Graur
Systems Group
Dept. of Computer Science
ETH Zürich
Zürich, Switzerland
dan.graur@inf.ethz.ch

Rodrigo Bruno
INESC-ID / Técnico, ULisboa
Lisbon, Portugal
rodrigo.bruno@tecnico.ulisboa.pt

Gustavo Alonso
Systems Group
Dept. of Computer Science
ETH Zürich
Zürich, Switzerland
alonso@inf.ethz.ch

Abstract

The Collections framework is an essential utility in virtually every Java application. It offers a set of fundamental data structures that exploit Java Generics and the `Object` type in order to enable a high degree of reusability. Upon instantiation, Collections are parametrized by the type they are meant to store. However, at compile-time, due to type erasure, this type gets replaced by `Object`, forcing the data structures to manipulate references of type `Object` (the root of the Java type system). In the bytecode, the compiler transparently adds type checking instructions to ensure type safety, and generates bridge methods to enable the polymorphic behavior of parametrized classes. This approach can introduce non-trivial runtime overheads when applications extensively manipulate Collections.

We propose the Java Collections Specializer (JCS), a tool we have developed to deliver truly specialized Collections. JCS can generate `ArrayLists`, `ConcurrentHashMaps` and `HashMaps` with true type specialization that incur no performance penalties due to bridge methods or type checking instructions. JCS offers the possibility to easily extend its use to other Collection data structures. Since the specialized data structures extend and inherit from the generic counterpart's superclasses and interfaces, the specialized versions can be used in most places where generic versions are employed. The programmer uses JCS to generate specializations ahead of time. These are generated under the `java.util` package, and need only be added to the class path and integrated into the application logic. We show that the specialized data structures can improve the runtime performance of data intensive workloads by up to 14% for read use-cases and 42% for write use-cases.

MPLR '21, September 29–30, 2021, Münster, Germany

ACM ISBN 978-1-4503-8675-3/21/09…$15.00
https://doi.org/10.1145/3475738.3480718

CCS Concepts: • **Software and its engineering** → **Object oriented languages**; **Object oriented frameworks**; **Source code generation**; **Polymorphism**.

Keywords: Java, data structures, code generation, runtime, specialization, generics, optimization

ACM Reference Format:
Dan Graur, Rodrigo Bruno, and Gustavo Alonso. 2021. Specializing Generic Java Data Structures. In *Proceedings of the 18th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR '21), September 29–30, 2021, Münster, Germany.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3475738.3480718

1 Introduction

Data structures stand at the core of all software applications. Structures such as hash maps, lists, double-ended queues, or sets are a necessity for building complex software solutions. Consequently, these fundamental data structures are frequently provided as part of the standard development kits across a large number of programming languages. For example, in Java these data structures are available as part of the Collections framework, in C++ they are part of the Standard Template Library, and in Rust they are offered as part of the Standard Collections Library. Such data structure libraries exploit language constructs to ensure seamless reusability across any type of application. These constructions carry different names and implementations across languages, but their ultimate goal is to allow the programmer to write behavior once and repeatedly reuse it across various data types. In Java and Rust, this mechanism is called Generics, and in C++ it is called Templates.

The Collections framework in Java represents a fundamental part of the programming language's ecosystem. The framework consists of a set of essential parametrized [8] data structures, which are represented, on the one hand, by contract-defining interfaces, such as: `Map`, `List` or `Set`, and, on the other, by the classes that carry out their implementations. Well known implementations include `HashMap`, `ArrayList`, `HashSet` and `ConcurrentHashMap`. The Collections framework extensively relies on the use of Generics and the Object type. This gives these data structures the ability to be easily reused with any data type. Due to the intrinsic ways in which Java implements Generics via the use of type erasure [1, 6], as well the frequent use of the

Object type, at runtime such data structures end up storing and manipulating references of the Object type rather than the target type to which the programmer specializes. Consequently, the compiler introduces a set of type checks, type casts, as well as additional proxy methods around these data structures, which imply a non-trivial overhead when the application processes large collections. This can come as a surprise to the programmer, which expects the data structures to be truly specialized and to incur no additional overheads.

We henceforth refer to the true specialized data structures as *specialized data structures*, and the original Java implementations of such data structures as *generic data structures*.

In this paper we present an approach towards generating truly specialized data structures from the collections framework:

- We present JCS, a tool capable of generating specialized data structures from the Collections framework, and explain its mechanisms for generating specializations. The specialized data structures offer identical behavior, but do not rely on using Generics or the Object type, thus avoiding performance penalties.
- We show that specialized data structures generated by JCS produce up to 14% runtime improvement on read intensive use-cases, and 42% on write intensive use-cases.

2 Background

The ubiquitous nature of the Java Collections framework is due in no small part to its large degree of reusability. To enable this reusability, the classes and interfaces in the framework make extensive use of Java Generics. This allows such data structures to be easily employed in virtually any context, and with any super-type. For example, instantiating an `ArrayList` that stores instances of a the type rooted in the `Point` type, one can use `ArrayList<Point> myArrayList = new ArrayList<Point>()`.

Due to an early design decision, Java implements Generics without any type specialization. This contrasts, for example, with languages such as C++ and C#. Instead, Java implements it through type erasure. Type erasure allows multiple specializations of the same data structure to be reduced to a single one at compile time, reducing the memory footprint of the application code. To ensure consistent types and polymorphism, the compiler introduces type checks, casts and bridge methods. This design brings limited advantages in modern runtimes, as memory is no longer a limiting factor. Moreover, all the additional compiler code is expensive and increases the cost of working with parametrized data structures

2.1 Type Erasure and Type Checks

With the help of Generics, a programmer can define behavior without binding it to a specific type. Listing 1 shows the logic of the `ArrayList`'s `get` method. Note the use of the `E` placeholder, in lieu of the data element's type, which is only later revealed at run-time when the method is called. Such examples can be found all throughout the logic of the Collections framework.

```java
public E get(int index) {
    rangeCheck(index);
    return elementData(index);
}
```

Listing 1. `ArrayList`'s `get` method logic.

```java
public Object get(int index) {
    rangeCheck(index);
    return elementData(index);
}
```

Listing 2. `ArrayList`'s `get` method logic after type erasure.

While Generics are an essential tool in the implementation of the Collections framework, they only provide part of the solution. As Java is statically typed, one cannot instantiate generic instances, nor can one readily call methods on generic objects. In the context of the Collections data structures, it is thus not possible to store data elements internally under the form of generic arrays, as these would require instantiation. Consequently, such data structures often rely on the `Object` type for internal storage and data representation. For instance, `ArrayList` stores its data in an array of the form:

`Object[] elementData`,

which can be instantiated.

At compile time, Java programs undergo an additional stage, called *type erasure*, wherein generics get replaced with the `Object` type. For instance, Listing 2 shows the logic of Listing 1 after type erasure. Note that the `E` placeholder has now been replaced by the `Object` type. To compensate for type erasure and enforce type safety, the Java compiler transparently introduces type checking instructions in the compiled bytecode (as demonstrated below).

Through type erasure and the extensive use of the Object type in the bytecode, data structures in the Collections framework ultimately end up storing and manipulating `Object` references. They also heavily rely on frequent type checking instructions in the bytecode. To the average programmer, this behavior might end up being surprising as such data structures are expected to be truly specialized for the type they are declared for. Moreover, due to the numerous type checking instructions, non-trivial runtime overheads can occur when iterating over large collections.

A possible approach towards truly specializing Collections, refers to deploying a Java agent with the aim of intercepting and instrumenting an application's bytecode during the Class Loading stage. At this stage in the execution, the code

```
ArrayList<Point> pointArray = new ArrayList<Point>();
...
for (int i = 0; i < pointArray.size(); ++i) {
    Point p = pointArray.get(i);
}
```

Listing 3. Iterating through an ArrayList of Point instances

```
...
74: invokevirtual #29 // ArrayList.get:(I)LObject;
77: checkcast     #13 // class Point
80: astore        5
...
```

Listing 4. Simplified bytecode showing the unpacked instructions around the get method call from Listing 3.

has already undergone type erasure and the bytecode already features checkcast instructions that ensure type safety. Listing 3 exemplifies a for loop iterating though an ArrayList of Point instances. Listing 4 shows the compiled bytecode around

Point p = pointArray.get(i).

Note that

74: invokevirtual #29

is responsible for calling the get method of pointArray. Due to type erasure, the return type of this method is Object. To ensure the returned object is indeed an instance of Point, instruction

77: checkcast #13

is added by the compiler. If the check passes, the instance is finally stored in

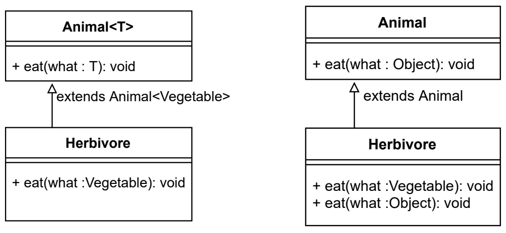
Point p

via

80: astore 5.

checkcast instructions also get added inside the logic of the Collections' data structures. For instance, such an instruction is placed inside the logic of ArrayList's add method or HashMap's put method, in order to ensure that the object added to the list is safe to add to the elementData array.

Consequently, a tool at this level needs to parse the bytecode, identify the instructions where data structures from the collections framework are instantiated, generate data structures with identical behavior and true type specialization, and finally instrument the bytecode, such that the specialized data structure gets instantiated at runtime. While this approach would ensure no checkcast instructions are added to the bytecode of the specialized data structure, this still leaves the checkcast instructions added by the compiler in the user code. The tool would also have to identify the checkcast instructions associated to the newly specialized data structure and remove them. Granted that all Collections data structures are specialized within the context of


Animal<T>
+ eat(what : T): void
extends Animal<Vegetable>
Herbivore
+ eat(what :Vegetable): void
Animal
+ eat(what : Object): void
extends Animal
Herbivore
+ eat(what :Vegetable): void
+ eat(what :Object): void

(a) Pre-compilation **(b)** Post-compilation

Figure 1. Class hierarchy which shows the generation of the bridge method eat(what: Object) in Herbivore.

an application, removing the checkcasts is safe, since no segment of code could be executed both by specialized and generic data structures. Otherwise, the decision to remove checkcasts becomes non-trivial, as one must keep track of what types of data structures share what sections of code, and only remove those sections that are not shared by specialized and non-specialized data structures. As analyzed codebases can be considerably sizable and complex, with arbitrary interactions between objects and methods, this task quickly becomes very complex.

2.2 Bridge Methods

A set of additional challenges stem from Java's approach of ensuring polymorphism in the context of type erasure via the use of *bridge methods*. Figure 1 describes this phenomenon graphically. The class Animal makes use of Generics (here identified by the type placeholder T). Herbivore implements Animal and parametrizes the generic to the class Vegetable, as shown in Figure 1a. After compilation, due to type erasure, T gets replaced by Object. Hence, the what parameter of the eat method is of type Object in class Animal. In class Herbivore no type erasure takes place, hence the same parameter in eat is of type Vegetable. This is problematic for enabling polymorphism, since calling the eat method on an object declared as

Animal<Vegetable> animal = new Herbivore()

would call the logic in Animal rather than Herbivore. To make sure polymorphism works correctly, the compiler automatically generates bridge methods in the sub-class, that have identical signatures to the generic methods in the parent class. Bridge methods are also added when a class implements a generic interface. These methods intercept calls, and redirect them to the parametrized method in the sub-class by casting the parameters to the correct type. Figure 1b shows the class diagram after compilation, when eat's bridge method is added. Listing 5 shows the logic of the aforementioned bridge method.

Bridge methods are necessary in order to ensure polymorphism works correctly in the context of type erasure. They however introduce additional overheads due to the

```
void eat(Object what) {
    eat((Vegetable) what);
}
```

Listing 5. Code of the `eat` bridge method.

extra level of indirection, and the intrinsic `checkcast` instructions added by the compiler in their bytecode to ensure type safety. In the context of specialized data structures they represent a significant challenge. If the tool generates specializations which extend the original generic classes or interfaces, then the specialized class would still feature bridge methods, which are employed whenever the type of the data structure is seen as that of the parent.

Generating specialized data structures in this manner can be desirable, in spite of the bridge methods, as it provides the specialized data structures with greater usability throughout the code. For instance, a specialized data structure of the form

`public class IntegerMap implements Map<Integer>`

can be used as an instance of `Map` wherever necessary. This type information inside the `Map` instance already unlocks several type optimizations such as Object Inlining. In a recent work [2], this very technique was used to inline objects inside Java `HashMaps`, resulting in significant reductions in the total memory footprint to keep the data structure in memory, reduction in GC latency, and improvements in data structure access throughput.

Alternatively, a tool can instrument the code, such that polymorphic behavior on specialized data structures is minimized by narrowing down the parameter types of methods, or duplicating them with narrower types. When polymorphic behavior is completely removed, specialized data structures no longer make use of their bridge methods, and the specialized data structure incurs no additional overheads due to the use of generics and type erasure.

Carrying out this type of instrumentation at the bytecode level is difficult, as the agent needs to undo changes introduced by the compiler (e.g. `checkcast` instructions), and make extensive changes to the bytecode in order to remove the polymorphic behavior of specialized data structures. A more straight-forward approach is, thus, to modify the source code directly, prior to compilation. Our tool acts on the code in this way, generating and plugging in specialized data structures in the code prior to compilation.

3 JCS Design

JCS generates the specialized data structures *prior* to application compilation. To benefit from the generated data structures, the programmer simply uses them inside the application code. JCS currently provides support for specializing the `ArrayList`, `HashMap` and the `ConcurrentHashMap` data structures, but additional classes can be easily added by exploiting the current codebase of JCS. The generated specializations have a high degree of usability, as they extend and implement the superclass and interfaces of the original generic type. It should be noted, however, that instances of the specialized type should generally not be cast to more generic types (e.g. a specialized version of `ArrayList` to `List`), as this will force the compiler to add type checking instructions, and to make use of bridge methods.

To partially address this problem, it would be possible to generate a specialized version of an interface, such as `List`, that extends the base interface, and is used by the specialized data structure in lieu of the base interface itself. For instance, one could generate the specialized

`interface ListInteger extends List<Integer>`,

which is then used in the specialization

`class ArrayListInteger implements ListInteger`.

This would allow one to upcast specialized subclasses such as the class `ArrayListInteger` to `ListInteger` and sidestep bridge methods. The programmer could then use the new `ListInteger` type in code in order to increase its reusability. As `ListInteger` extends `List` it would be possible for its subclasses to be upcasted to `List` as well. In the latter scenario, however, bridge methods would be employed. JCS does not support interface specialization out-of-the-box, but adding support for it is straight forward.

To use the specialized types generated by JCS, the programmer needs to follow two steps. In the first step, the programmer runs JCS in order to generate a set of specialized data structures and compile them to bytecode, such that they can later be used in the application. For the the second step, the programmer adds the generated binaries to the class path of the application, and readily uses the structures generated in the first step by importing them into the source code and employing the new types as any other regular class.

3.1 Generating the Specialized Data Structures

The specialized data structures must be generated prior to running the application. To do so, the programmer runs JCS and specifies the types of specialized data structures that need to be generated. For instance Listing 6 exemplifies the command line required to run the generation part of our tool. Note that

`[<specializations with full paths>]+`

represents a list of space separated data structures from the collections framework with fully qualified paths and explicit data types. An example of such an element is

`HashMap<String, Point>`.

This latter element would request the generation of a `HashMap` specialized to `String` and the user defined class `my.testpackage.Point`. The generated data structure will be called `HashMapStringPoint`, and is compiled under the `<generated binaries directory>`.

JCS generates these data structures by parsing the source code of the generic version of the data structure from the Collections framework and generating its Abstract Syntax Tree

```
java TypeSpecialization.jar <path to jdk sources> \
    <generated sources directory> \
    <generated binaries directory> \
    ["<specializations with full paths>"]+
```

Listing 6. Bash command for generating specialized data structures using JCS.

(AST). Via the visitor pattern, nodes of interest in the AST are modified. Generic placeholders are replaced by the concrete types for which the data structure is being specialized. For example, to generate the `HashMapStringPoint` data structure, JCS replaces the `K` type placeholder with `String` and the `V` placeholder with `Point`. JCS also makes sure to add any necessary imports, e.g. when using a user defined class such as `my.testpackage.Point`. Inner classes also require specialization. For instance, the `Node`, `KeySet`, `EntrySet` and `Values`, all nested classes in `HashMap`, require specialization. In many cases, classes from the Collections framework rely in some form on auxiliary classes, outside of their own compilation unit, which also employ Generics. For instance, `HashMap` relies on `LinkedHashMap` and `LinkedHashMap.Entry`. Consequently, when `HashMap` gets specialized, these classes need to be specialized as well. JCS handles these additional tasks automatically.

Extending JCS such that it supports additional data structures should be a relatively straight forward task. JCS's main class, called `TypeSpecialization`, needs to be augmented in two ways. The first change implies the definition of a new method which creates a specialization request. The specialization request contains essential information for the specialization process, such as the name of the generic placeholders to be replaced, the names of the original and the target compilation units, and any additional classes outside of the original compilation unit which also need to be specialized due to dependency concerns (e.g. the `LinkedHashMap` and the `LinkedHashMap.Entry` in the case of the `HashMap`). The second change refers to augmenting `TypeSpecialization`'s `main` method such that the previously defined specialization method can be used correctly. The changes required in `main` refer to recognizing an incoming specialization request for the new data structure, calling the new method, and passing its return value further down along the specialization logic.

There are cases when adding support for a new data structure requires the visitor logic to be augmented. For instance, in the case of the `ConcurrentHashMap`, the default specialization visitor class `SpecializationVisitor` needs to be extended, in order to add additional visitor behavior that makes the specialization work correctly, such as adding an additional import or removing certain code elements.

3.2 Compiling and Launching Applications

Once the AST has been modified, JCS requests its compilation. Both the generated sources and the compiled class file are stored on disk, at the paths indicated by the user. The specialized `ArrayLists` and `HashMaps` are added to the `java.util` package. The specialized `ConcurrentHashMaps` are added to the `java.util.concurrent` package. The generated classes are ready to be imported and used directly into a project, as long as the application includes the binaries of the new data structures in the class path. Listing 7 shows an example of how the `HashMapStringPoint` generated by JCS can be used in a user application. The advantage of this approach is that no bridge methods are employed, since the specialized data structure object requires no polymorphic behavior.

```
import java.util.HashMapStringPoint;
...
HashMapStringPoint specializedMap =
    new HashMapStringPoint();
```

Listing 7. Using a specialized data structure directly in code.

Since the newly generated specialized data structures are defined in already existing and well-known packages (`java.util`, for example), additional flags may need to be passed into the `javac` and `java` commands. In particular, depending on the JVM implementation, extra command line flags may be necessary to allow new classes (i.e., the specialized data structures) to be added to already existing packages and modules.

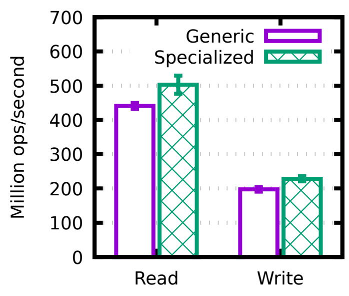
JCS does not add any additional security elements to the JVM runtime. This implies JCS enjoys the default level of security of any `jar` whose code is running inside the JVM. It would be possible to add additional security measures, such as checking a security digest of the specialized class against a trusted reference hash. This would ensure with high probability that the executed code is indeed trustworthy. We leave this aspect for future work.

4 Evaluation

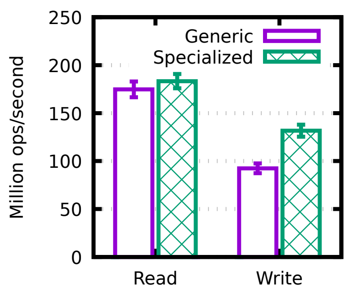
This section presents the performance benefits of specializing popular Java data structures. In particular, we are interested in measuring the throughput improvements that result from the simplified data structure access code when using specialized data structures.

4.1 Experimental Setup

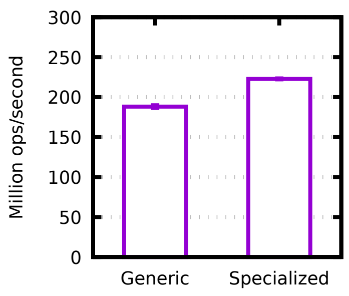
We do not report results on other performance metrics such as memory footprint, Garbage Collection overhead, or application tail latency as these metrics remain unchanged with the proposed transformation. Given that specializing data structures creates additional classes to compile and store in the code cache, we also measured such overheads. However, no measurable JIT compilation nor code cache overheads could be observed and therefore such metrics are not reported.



(a) `ArrayList`



(b) `HashMap`



(c) Producer-Consumer

Figure 2. Throughput comparison of Generic versus Specialized data structures.

Experiments run in isolation for at least 1000 iterations (more iterations are used if results take longer to stabilize). The last 500 iterations are utilized to create average values and the standard deviation. To reduce interference in our measurements, we run all experiments with enough memory such that no GC cycle ever runs during our measurements (we found that 256MB of heap size is enough to prevent GC cycles). Experiments take place in a single cluster node running Debian 10 (Linux kernel 4.19.0-10) equipped with an Intel(R) Xeon(R) CPU E3-1225 v6 @ 3.30GHz, and 32GB of DDR4 DRAM. CPU frequency scaling and hyper-threading are disabled. A GraalVM CE 21.1.0 (based on Java 11) Java Virtual Machine is used in all experiments. The following JVM flags are passed to the JVM to add the specialized data structure classes to Java's `java.base` module:

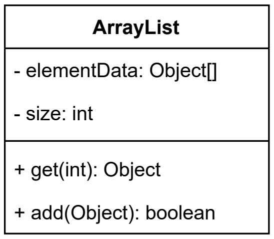
`-patch-module java.base=<spec. classes>` and
`-add-reads java.base=ALL-UNNAMED`.

In addition, the following JVM flag is used to allow specialized data structure classes to access user data types:

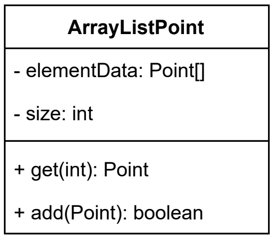
`-Xbootclasspath/a:<user data type classes>`.

These JVM flags are required for two reasons: i) specialized data structures are defined inside the `java.base` module and therefore these generated classes need to be appended to the `java.base` module (specialized data structures could not be defined outside the `java.base` module due to dependencies to package-private definitions in the original data structures); ii) the specialized data structure classes may need to import user data types (`Point`, for example), which therefore also need to be loaded by same Java classloader used for `java.base` (bootstrap classloader). No other flags are utilized.

The remainder of this section is further divided into two sub-sections. First, we explore the throughput improvements resulting from type specialization when using two popular Java data structures (`ArrayList` and `HashMap`). Second, we provide a more complete use-case using a producer-consumer pattern to showcase the performance impact of the specialization.

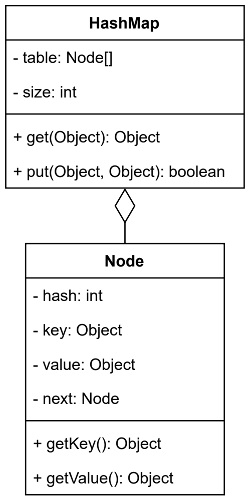


(a) Generic

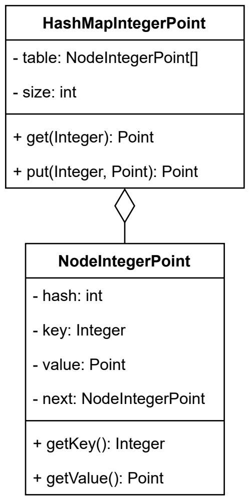


(b) Specialized

Figure 3. Generic and specialized Java `ArrayList`.



(a) Generic



(b) Specialized

Figure 4. Generic and specialized Java `HashMap`.

4.2 Specialized Java Data Structures

Data structures play an important role not only in application development, but also in its performance. In this section we analyze the performance impact of generic data structures compared to specialized data structures. To that end, we select two of the most popular Java data structures: `ArrayList` and `HashMap`.

4.2.1 `ArrayList`.

is a popular generic data structure that internally stores an array of objects. Figure 3 (left-hand side) presents a simplified class diagram of `ArrayList`. The diagram also includes two key methods in the implementation of the data structure, the `get` method that allows accessing stored objects and the `add` method that allows inserting objects into the data structure. A second version of the same data structure is also present in Figure 3 (right-hand side). This data structure is specialized to the `Point` class, a simple data type representing an N-dimensional point in space. From the diagram, it is possible to identify that the internal array (`elementData`) and both methods utilize the specialized type (`Point`) instead of the original `Object` type.

We compare the performance of both versions by performing a total of 1M read (`get` method) and write (`add` method) operations per iteration on each data structure. Results are presented in Figure 2a and show that the specialized data structure, for the same read and write workload, improves read throughput by 14% and write throughput by 16%. These performance improvements result from the reduced number of instructions to access the data structure, which no longer has to perform type checks.

4.2.2 `HashMap`.

is, perhaps, the most popular data structure in Java. It provides a simple interface to a key-value store which maps keys to values. Figure 4 (left-hand side) presents a simplified class diagram of the data structure. A `HashMap` contains (among other fields and methods), a `Node` array (`table`). A `Node` represents a single key-value pair. Both the `HashMap` and the `Node` classes deal with objects of `Object` type.

In this experiment we specialize a `HashMap` to `Integer` (the key type) and to `Point` (the value type). Figure 4 (right-hand side) presents the class diagram of the specialized classes. As depicted, all types are now specialized including the new `NodeIntegerPoint` which replaces the original `Node`.

Similarly to the `ArrayList` experiment, we also exercise both data structures (generic and specialized) through 1M read (`get` method) and write (`put` method) operations per iteration. Figure 2b shows that the specialized data structure improves read throughput by 5% and write throughput by 42%. As for the previous experiment, all improvements come from the reduced number of instructions necessary to fetch and insert items into the data structure.

4.3 Specialized Producer-Consumer

While in the previous section, all experiments were focused on measuring the performance of a particular data structure operations, in this section, we test a widely used programming pattern, *producer-consumer*. This is a useful pattern used to synchronize the communication between a number of producers and consumers through a shared data structure (typically a queue). Among other use-cases, this pattern can be used to implement load balancers.

In our particular instance of producer-consumer, we use a single producer inserting objects into a shared `ArrayList` (similar to the one presented in Figure 3), and three consumers which remove objects from the shared data structure and place it into a private one. It should be noted that for each producer-consumer operation, two data structures are necessary: i) one operation to remove an item from the producer data structure, and ii) one operation to insert the item into the consumer data structure. For this experiment, a total of 1M operations per iteration were issued.

Results in Figure 2c show that, following the trend of the previous experiments, specializing data structures does improve data structure access performance. In this experiment, data structure specialization improved throughput by 18.4% compared to using the original generic data structures.

5 Related Work

Previous work has focused on generating specialized types in Scala [5]. The tool allows the user to annotate generic parameters in classes or methods. During compilation, classes which permit specialization (i.e. those that are annotated), are expanded if they are further used in a specialized manner, e.g another class extends the generic one, and specializes at least one of its specializable generics. Expansion refers to generating several versions of the original class, where methods with specializable generics are duplicated: once with the Object type, and once with the concrete type. The newly generated classes are then introduced between the original subclass and superclass. JCS differs from this approach, as it focuses on specializing canonical classes from the Collections framework. Moreover, by allowing the programmer to generate specialized data structures prior to compiling the user applications, the programmer can use the specializations as types directly in code, avoiding bridge methods completely.

Alternative approaches to implement Generics in Java without modifying the JVM have focused on producing alternative compilers, such as NextGen [3]. For each instantiation of a parametrized type, NextGen generates wrapper classes and interfaces which carry type information. The generated classes extend an artificial abstract base class constructed from the logic of the type erased parametrized base class. Our method differs from this, since it does not generate wrapper methods to carry additional type information, rather it generates specialized classes that directly use the target class, and trivially extend and inherit the classes and interfaces of the original base class. Moreover, our tool works prior to the compilation step.

Other work has focused on identifying how JIT compilers are suitable in carrying out runtime optimizations to collections based workloads in Scala [9]. The authors recognize the challenge of speculating on data structure invariants, and endorse work on static specialization of generic data structures. Improving the performance of the type checking system in the JVM presents another line of work which benefits the runtime of `checkcast` and `instanceof` bytecodes [4]. Ultimately, however, these type checks would still need to be executed, and would still incur an overhead. In contrast, our work tries to completely sidestep type checks.

The C# programming language implements generics in an efficient manner, by generating specialized code at runtime via just-in-time type specialization, a feature which is permitted by the programming language's dynamic runtime [7]. Consequently, C# can generate truly specialized code for generic types with low overhead, and have it linked at runtime. Moreover, this mechanism also provides support for specializing generics towards primitive types, without employing boxing and unboxing.

6 Future Work

JCS provides much opportunity for future work. We see potential in extending a priori specialization to parametrized classes beyond the Collections framework. In addition to this, work on instrumenting legacy code, such that only specialized data structures are used, is an interesting avenue for future research. Adding extra security features to the specializations is another potential topic for future work.

A current limitation of the current JCS implementation is the lack of support for nested data structures. For example, if a user needs a specialized version of
`HashMap<String, HashMap<String, Point>>` into
`HashMapStringHashMapStringPoint`,
two consecutive specializations would have to be invoked instead of just one. The first specialization would create a specialize
`HashMap<String, Point>` into
`HashMapStringPoint`,
and then a second specialization would be required to specialize
`HashMap<String, HashMapStringPoint>` into
`HashMapStringHashMapStringPoint`.
Supporting nested data structures does not require significant design changes and it would allow users to easily specialize any combination of data structures.

We also see research potential in producing more automated means of using specializations which avoid bridge methods and type checking instructions, with as little programmer intervention as possible. For instance, a potential future research idea is a code analysis tool which identifies all uses of generic data structures and specializes them on the fly. While very appealing, this approach brings a set of interesting challenges. For instance, one would have to trace the use of the specialized data structures throughout the code, and identify what methods they are used in when passed as parameters. To ensure that such programs continue to run as expected, such methods, where specialized data structures appear, would have to be replicated once for the generic version and once for the specialized version of the data structure. Methods which make use of several specializable parameters will have to potentially be expanded into all possible combinations of specialized and non-specialized parameters. Most of these expansions are required to ensure the specialized code runs as expected when interacting with foreign non-specialized code. The problem can be simplified by limiting the scope of the specialization exclusively to self-contained code, which does not interact with foreign logic.

Another possibility would be to use user annotations in order to indicate which data structures to specialize. Such data structures could then be specialized on the fly and instrumented into the bytecode during the compilation process. This approach removes the need of generating the data structures prior to writing the code.

7 Conclusion

We have presented JCS, a tool capable of generating specialized versions of data structures from the Collections framework. The use of the specialized data structures prevents the compiler from introducing type checking bytecode instructions, and avoids the use of bridge methods. JCS currently offers support for `ArrayLists`, `HashMaps` and `ConcurrentHashMaps`, with the possibility of easily extending its use to other data structures. JCS generates the specialized data structures prior to compiling the application logic. The generated specializations are added to the `java.util` package, and need only be added to the class path at compilation to be visible. The specializations can be interchangeably used with their generic counterparts, as they extend and inherit from the generic version's superclass and interfaces. In other words, specializations are siblings to their generic counterparts in the class hierarchy. We have shown that the specialized data structures offer up to 14% runtime improvement for read intensive use-cases and up to 42% write intensive use-cases. Our experiments also show that the popular producer-consumer pattern sees an 18.4% runtime performance improvement. JCS is open source, and is available at https://github.com/rodrigo-bruno/specialized-java-datastructures/tree/specialization.

Acknowledgments

The work of Dan Graur and Rodrigo Bruno has been funded in part by grants from SBB and Oracle Labs. The work was done while Rodrigo Bruno was at ETH Zürich.

References

[1] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. 1998. Making the Future Safe for the Past: Adding Genericity to the

Java Programming Language. *SIGPLAN Not.* 33, 10 (Oct. 1998), 183–200. https://doi.org/10.1145/286942.286957

[2] Rodrigo Bruno, Vojin Jovanovic, Christian Wimmer, and Gustavo Alonso. 2021. Compiler-Assisted Object Inlining with Value Fields. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI'21)*. Association for Computing Machinery. https://doi.org/10.1145/3453483.3454034

[3] Robert Cartwright and Guy L. Steele. 1998. Compatible Genericity with Run-Time Types for the Java Programming Language. *SIGPLAN Not.* 33, 10 (Oct. 1998), 201–215. https://doi.org/10.1145/286942.286958

[4] Cliff Click and John Rose. 2002. Fast Subtype Checking in the HotSpot JVM. In *Proceedings of the 2002 Joint ACM-ISCOPE Conference on Java Grande* (Seattle, Washington, USA) *(JGI '02)*. Association for Computing Machinery, New York, NY, USA, 96–107. https://doi.org/10.1145/583810.583821

[5] Iulian Dragos and Martin Odersky. 2009. Compiling Generics through User-Directed Type Specialization. In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems* (Genova, Italy) *(ICOOOLPS '09)*. Association for Computing Machinery, New York, NY, USA, 42–47. https://doi.org/10.1145/1565824.1565830

[6] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. 2000. *Java Language Specification, Second Edition: The Java Series* (2nd ed.). Addison-Wesley Longman Publishing Co., Inc., USA.

[7] Andrew Kennedy and Don Syme. 2001. Design and Implementation of Generics for the .NET Common Language Runtime. In *Programming Language Design and Implementation* (programming language design and implementation ed.). ACM Press.

[8] Andrew Myers, Joseph Bank, and Barbara Liskov. 1999. Parameterized Types for Java. *Conference Record of the Annual ACM Symposium on Principles of Programming Languages* (08 1999). https://doi.org/10.1145/263699.263714

[9] Aleksandar Prokopec, David Leopoldseder, Gilles Duboscq, and Thomas Würthinger. 2017. Making Collection Operations Optimal with Aggressive JIT Compilation. In *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala* (Vancouver, BC, Canada) *(SCALA 2017)*. Association for Computing Machinery, New York, NY, USA, 29–40. https://doi.org/10.1145/3136000.3136002