Advanced Multiprocessor Architecture (CS471) Project Report

Measuring the Performance Characteristics of Miniboxing-transformed Scala Programs

Vlad Ureche
EPFL
vlad.ureche@epfl.ch

1. Introduction

Generic programming [10] lets developers reuse common behavior in different contexts, such as, for example, using the same linked list behavior to store names for an address book application or file descriptors for an operating system. In both examples the linked list will offer the same insertion, deletion and search time guarantees, whether it carries names, file descriptors, on any other type. Rewriting the linked list for each use is both error prone and redundant: in most modern programming languages, a linked list can be encapsulated, usually as a class, irregardless of the type it carries. The type is then passed as a parameter, thus the name type parameter.

While generic programming is wide spread, the underlying support varies significantly from one language to the next, and it heavily influences performance. C++, C# and Java all support generic programming [2, 9, 13], but they do so in very different ways: a C++ compiler, for example, will create a different class for each type it carries. C#, on the other hand, will create a single class but will rely on the dot net virtual machine to create different classes at runtime [9], specialized for the types they carry, incurring a small overhead on each execution.

The Java ecosystem has its own take on generic programming: the compiler and virtual machine use a single class for generic code. While this looks ideal, as it does not incur any machine code duplication, Java makes an important assumption on the types that can be used with generic code: they have to be heap-allocated instances that are passed as references. This assumption affects performance significantly when the generic class is used with so-called value types, such as characters, integers or floating points, which are meant to be passed as stack values instead of references. To allow programmers to use value types with generic code, Java allocates heap instances that store the values. This encoding, known as boxing, incurs significant overheads, especially when working with arrays: instead of the values being stored one after the other, minimizing the memory footprint and improving locality, they will be stored as references, each pointing to a random location on the heap containing the boxed value. This wastes memory, as the boxed value also contains the standard class header, but also affects locality, as there is no guarantee consecutive values will occupy consecutive heap memory locations.

A solution that avoids this overhead is using a separate specialization transformation [11] to create copies of the generic class for each of their uses. Machine code that targets the Java Virtual Machine, also called bytecode, loses part of its type information during compilation [3], making runtime specialization in the virtual machine impractical. It follows that the only alternative to perform specialization is at compilation time, before creating the bytecode.

Scala, the language we study in this project, targets the Java Virtual Machine and is able to perform compile-time specialization [5]. This is currently implemented as a compiler phase that takes a class and creates multiple versions, called specialized classes, in which the types are propagated. For reasons we will explain in the next section, the static specialization transformation in Scala creates sizable amounts of bytecode by duplicating classes and methods several times, becoming hard to use in practice.

To avoid generating large amounts of bytecode, we proposed an alternative to the specialization transformation which obtains comparable performance with less bytecode. It does so by unifying all value types into one, which can accommodate all others without loss in precision. This allows our transformation, dubbed miniboxing, to keep the bytecode size under control while obtaining similar performance to specialization.

The results we obtained with the current prototype were puzzling. We prepared a prototype of the miniboxing transformation as a Scala compiler plug-in. As expected, the original experiments showed performance numbers that were between generic code and specialized code, since there were additional transformations that had to be done to extend the value types. But with all optimizations enabled, the miniboxing code was much faster than the specialized code [14] for linked lists, which was a pleasant but unexpected surprise. This project attempts to find reliable ways to benchmark and answer why, contrary to our expectations, the miniboxed code performed better.

In the next section we will present related work. Later, we will look at the specialization and miniboxing transformations in more detail. In section 4 we will present a series of hypotheses and corresponding crucial experiments in an attempt to find the cause of the puzzling results. In section 5 we will look at the challenges that we faced during the benchmarking and in Section 7 we will present the results we obtained for the benchmarks. We will conclude in section 8.

2. Related Work

The methods of supporting generic types have been developed in different directions in each language. Since the introduction of generic programming in ML[4], there has been constant interest in the benefits of type system guarantees. Even now, there is no one-size-fits-all solution for generic programming low level support: C++[13], for example, performs template expansion at compile time, so it requires the prototypes of all used classes to be available at compile time. Furthermore, there is no mechanism in place to provide compatibility between different instantiations of templates – treating an instantiation of a template as a different instantiation will lead to memory corruption and reading invalid memory locations. The Java programming language relies on erasure [2] to have a single memory layout for any instantiation of a generic class, but this comes at the expense of losing optimality for the value types in the language. Finally, the dot net virtual machine encodes the complete types in the bytecode [9] so is able to perform runtime generation of the memory layouts on demand. The tradeoff here comes from the fact that encoding full types in the bytecode creates a tight coupling between the language's type system and the virtual machine, forcing the virtual machine to keep up with the language specification.

Miniboxing builds on the specialization[5] transformation in Scala, which performs an opportunistic monomorphization of the generic code. The main difference between monomorphization and specialization lies in the fact that specialized classes maintain a common interface despite differences in the memory layout, allowing interoperability between the generic class and the monomorphic instances. This in turn eliminates the constraint that the reified type must be carried by all code, e.g. in Pizza[11] or Kafka[7]. In miniboxing, the type is known statically, the specialized class is instantiated, if not, the generic one is used. From an interface perspective, there is no difference between the specialized classes and the generic one, they can be used interchangeably.

Recent research on reducing bytecode size for embedded systems has produced a method of implementing generics that is similar to miniboxing. The insight in lightweight generics[12] is that more value types can be stored in a single, larger, value type. This approach is tested, among other techniques, for use in the area of embedded processors, where the compilers can work under the closed world assumption: code on an embedded device is not likely to be extended from outside. There are two fundamental differences between lightweight generics and miniboxing: one, miniboxing is meant to work under an open world assumption, with separate compilation and drop-in extensions to the compiled bytecode. And two, miniboxing, along with a reduction in the bytecode size, also allows a further runtime specialization that can significantly boost performance, which is not possible in the lightweight generics approach.

Micro benchmarking code running inside a virtual machine is a complex problem, mainly due to the indeterministic behaviors of the garbage collector and the just-in-time (JIT) compiler. The JIT compiler in a virtual machine is tasked with compiling interpreted bytecode to native machine assembly, which runs on the bare metal. It is tricky to balance the amount of code compiled and interpreted: too much interpreted code and the program might not obtain maximum throughput. Too much compiled code and the user might notice a lag that is caused by the compilation being on the critical path to seeing the results. These constraints, along with the unpredictable nature of memory allocation, garbage collection and application of optimizations on the compiled code create a very unstable environment for microbenchmarks. Fortunately, researchers looked at the problem and were able to isolate common pitfalls of benchmarking on the Java Virtual Machine[6], which we will mention in the Proposal and Evaluation sections.

3. Generics and Performance

Generic programming is crucial for code reuse, as can be seen in the following linked list example, written in Scala:

```
// list can be a node or the end of the list
abstract class LinkedList[+T]

// linked list node
case class Cons[T](head: T, tail: LinkedList[T])
    extends LinkedList[T]

// end of list marker
case object Nil extends LinkedList[Nothing]
```

In this example, the `LinkedList[T]` class stores elements of type `T`. The list itself can be either a node, represented by `Cons` or the end of the list, represented by `Nil`. The `abstract` keyword makes it impossible to instantiate the `LinkedList` class directly. Instead, a list is either a `Cons[T]` node or an empty list, `Nil`. The `case` keyword instructs Scala to treat `Cons` and `Nil` as algebraic data types. Using `object` instead of class makes `Nil` a singleton, allowing a single instance to exist. Finally, `Nothing` is the Scala bottom type, which is a subtype of all other types. Along with the covariant type parameter `T` in `LinkedList[+T]`, this makes it possible to have `Nil` end lists of any type.

Code targeting the Java Virtual Machine, loses some of its type information, so runtime class specialization is not possible without heavy annotations to recover the full types.

At the Java Virtual Machine bytecode level, the code loses its type parameters, as in the following example [1] :

```
class Cons(head: Object, tail: LinkedList)
    extends LinkedList
```

Since erasure eliminates the possibility of runtime specialization in the virtual machine, Scala employs a compilation phase which duplicates and transforms classes before generating bytecode.

3.1 Specialization Transformation

Specialization [5] is Scala's method of avoiding the overhead of boxing. It is a transformation phase in the compilation pipeline that runs before erasure and copies the code of a generic class once for each value type. In doing so, it also replaces type parameters with their corresponding value types. The erasure transformation takes place later in the pipeline, but it only affects the generic class, which kept the type parameters. The specialized classes, which had their type parameters replaced by value types, will now carry the values on the stack. The following code shows Cons specialized for Int, before erasure [1] :

```
case class Cons$Int(head$Int: Int, tail:
    LinkedList[Int]) extends Cons[Int](null,
    tail) {

  // override for head getter:
  def head: Object = Integer.valueOf(head$Int)
  // overridden constructor
  // bridge methods
}
```

The specialization scheme in Scala does not guarantee all instances of a specialized class are specialized instances. In the current transformation, class instantiation is specialized if and only if the type parameters are known at compile time. For example, `new Cons[Int](...)` will automatically be replaced by `new Cons$Int(...)`. This guarantees that the specialized instances will be used each time the type parameters are known statically. But the language allows class instantiation inside generic code, so not all values of type `Cons[Int]` are instances of `Cons$Int`. The listing below shows this loophole:

```
// generic code, instantiates class Cons, as
    there
// is no static information about the type T
def List[T](t: T) = Cons(t, Nil)

var intList: Cons[Int] = _

intList = Cons(1, Nil) // instance of Cons$Int
intList = List(1) // instance of Cons
```

[1] the code has been rewritten for readability, it is not an exact representation of either bytecode or Scala transformed abstract syntax trees

Scala-generated bytecode pays a heavy price for specialization. Since `Cons` and `Cons$Int` can take each other's place, they need to be compatible with respect to the methods, getters and setters they expose. This means, in our example, that both classes need to expose two getters for `head`: one returning `Object` and another one returning `Int`. But the `Cons` class must be compatible with all specialized classes, so it must expose the `head` getter for all the value types. And so do all specialized classes. Now, instead of a single getter, specialized classes must contain N getters, where N is the number of supported value types, 9 in the case of Scala. So not only is the class itself is duplicated N times, so are the methods inside it, creating a quadratic behavior in terms of bytecode size, only to maintain compatibility.

What if the class contains multiple type parameters like, for example, the common mapping from keys to values, `Map[K,V]`? This class has two type parameters, so the number of copies needs to be N^2, to cover the Cartesian product of all possible value types. What's more, the `def apply(key: K): V` method needs to be duplicated N^2 times in each class, creating a quartic (N^4) explosion in terms of bytecode size. To have a feeling of the numbers, starting from a map with a single method, `apply`, specialization would create 81 specialized classes each containing 81 overloads of the method. This is the reason specialization is used little in practice.

In the next section we will present a different approach to specialization that avoids the bytecode explosion.

3.2 Miniboxing Transformation

The miniboxing transformation uses a key insight to keep bytecode size under control. All value types can be stored in the language's largest value type: disregarding the operation semantics, it's a problem of having enough bits to store the information. In the case of the Java Virtual Machine, the long integer data type, `long`, can store all other value types without loss in precision. With this insight specialization is greatly simplified: we either have references to heap-allocated objects or value types that can be stored inside a `Long`. This reduces the number of specialized classes to $N = 2$, and allows a specialization for Map with only 4 classes. The tricky part of the transformation is coding and decoding the original type, whenever it is necessary. For this, we store an additional type byte, which we use to encode the type of the original value. We need to decode the type in very few and precise situations, one of which is shown in the next listing:

```
case class Cons[@miniboxed T](head: T, tail:
    LinkedList[T])
```

Compiling the example above will generate the following specialized class:

```
trait Cons[T] {
  def head: T
  def tail: LinkedList[T]
  def head$mbox: Long
  def tail$mbox: LinkedList[T]
}

case class Cons$mbox[T](val head$mbox: Long, val
    tail$mbox: LinkedList[T], T$type: byte)
    extends Cons[T]{
  def head =
      MiniboxingRuntime.miniboxToBox(T$type,
      head$mbox)
  def tail = tail$mbox
}

case class Cons$gen[T](val head: T, val tail:
    LinkedList[T]) extends Cons[T] {
  def head$mbox: Long =
      MiniboxingRuntime.boxToMinibox(head)
  def tail$mbox: LinkedList[T] = tail
}
```

The role of the `MiniboxingRuntime` object is to provide the conversions necessary from and to miniboxed values. These are required each time a miniboxed class's code interacts with generic code or fully-specialized code. Each call to `MiniboxingRuntime` needs to dispatch on the type byte `T$type` and perform type-specific operations.

There are three reasons for miniboxing code to be slower than fully-specialized code:

- miniboxing-transformed classes carry more data around: instead of the value type itself, the class carries a `Long` and a `Byte`, which take up more space
- the conversions from value types to long and back involve small but repetitive overheads
- the calls to `MiniboxingRuntime` perform an extra dispatch before doing the same work as their specialization counterparts

While the first two problems are inherent to miniboxing, the third can be addressed: `MiniboxingRuntime` calls and the type byte `T$type` value can be inlined to eliminate the costly dispatches.

Calls to the miniboxing runtime library can be eliminated. The first step is inlining the miniboxing runtime library calls into the caller code. Once inlined, the result is a switch on the type byte. But the type byte is a per-instance constant value that is set once at creation time and is never modified afterwards. A classloader can duplicate the class bytecode at runtime and mark the value of the type byte as a constant. This way, without any change to the class methods, the JVM is able to propagate the constant value of the type byte in the code and remove all unreachable branches, thus specializing the class. While conceptually a simple step, adding a classloader and creating the specialized instances is cumbersome in practice and requires extra work in terms of execution time. Nevertheless, our prototype showed it can be done and it offers a good tradeoff between instance speedup and on-time overhead for the first specialized instance of the class.

We benchmarked the miniboxing transformation and obtained results on par with current specialization [14]. The expectation was that miniboxing-transformed code would have performance between generic code and specialized code. This was indeed the case for the static compile-time transformation. What was puzzling was that using the runtime specialization we obtained better results for the miniboxing transformation compared to the specialization-transformed code.

4. Hypotheses and Experiments

The project will analyze the benchmarks and propose an explanation for the unexpected speedup of miniboxed code over specialized code. The comparisons will be done on the two benchmarks, array and list operations, with specialized code running side-by-side with miniboxed code.

The Java VM provides a layer of non-determinism by adding a series of moving parts, all of which may influence the running time: code can be interpreted or compiled, garbage collection may kick in at any time, optimizations are done for code that is run frequently and there is no direct way to control these events. This project aims to reduce the noise introduced by the virtual machine as much as possible and to use low level measurements to determine the root cause of the performance discrepancy observed.

The first two phases of the project will focus on removing noise from the benchmarks, such that we can analyze the different execution phases individually and with minimal intrusion from inherent noise, while the third phase will focus on measuring the low level hardware events using the hardware performance counters.

4.1 Hypothesis: Just in Time Compilation Affects Results

The JVM micro benchmarking is known to be troublesome [8], so the first step in the project is to analyze the benchmarking setup. At this step we need to make sure the programs are compiled to machine code instead of being interpreted and that one-time overheads are not affecting the measurements. We can set the Java Virtual Machine to log method compilation, so we can distinguish between the four or more phases of the benchmark execution:

- Startup latency, measured as a one-time cost to start the virtual machine, load support classes, and launch the benchmark
- Interpretation phase, observed between the startup and the time the benchmark inner method is compiled
- Compiled execution phase, after the inner benchmark method is compiled
- One or more optimized execution phases, as the virtual machine may optimize the method better and run the new version

Measuring the execution speed in each phase will either prove or disprove the hypothesis that the Just In Time compiler is triggered differently for the two benchmarks and consequently affects total execution time. We will also benchmark the execution time of the compiler, as the compilation time may be different between benchmarks.

The virtual machine typically fires the garbage collector if and only if it is low on heap space and more memory is requested. This may affect the benchmark running time, especially in the case of the linked list benchmark, which allocates memory at each iteration. Setting the heap size to a large value, using a parallel garbage collector and logging the garbage collection phases will allow us to remove the noise generated by the garbage collector.

4.2 Hypothesis: Different Optimizations

In the Java Virtual Machine array access is checked for correctness. In many typical cases, the virtual machine is able to prove the correctness of an array access, making it significantly faster, but this is not an optimization that can be relied upon for all code shapes. We have documented accounts of the array runtime checks not being removed and slowing down provably safe programs [8].

To verify the hypothesis that the two programs are optimized differently a code dump of the optimized x86 machine code will be analyzed.

4.3 Hypothesis: Level 1 Cache Thrashing

Another theory is that the larger methods generated by specialization do not fit in the level one instruction cache, and this causes a higher miss rate in the level one instruction cache. Using the hardware counters[1] will provide an insight into the cache behavior of the application.

5. Experiment Preparation

The miniboxing transformation is currently implemented as a Scala compiler plugin. This section will describe the plugin and will explain the transformations it performs on Scala programs.

5.1 The Plugin

The miniboxing plugin adds a code transformation phase in the Scala compiler. Any transformation phase in the the Scala compiler will receive the abstract syntax tree that resulted from parsing the source code and is expected to output the transformed tree. Normally, transformation phases run after the tree has been annotated with type information. In this case, the output tree is also expected to further be annotated with coherent type information. Furthermore, the symbol table in Scala can be manipulated by adding, removing or changing symbols' types. In our case, the miniboxing phase will create new entries in the symbol table for the specialized classes and will create trees for them.

Creating new entries in the symbol table drives the tree transformation. While creating symbol tables for the new entries, the miniboxing phase records information about how each new entry should be constructed: whether it will be the Generic or Miniboxed version of the class, where will it take the code from, what new members need to be created to maintain compatibility and so on. Later, this information is used to manipulate the tree, creating nodes for the new classes and methods. Looking at the example in the previous section, when the transformation encounters class`Cons`:

```
case class Cons[@miniboxed T](head: T, tail:
    LinkedList[T])
 // miniboxing - I need to generate:
 // trait Cons[T] { <members> }
 // case class Cons$mbox[T] extends Cons[T] {
 //  <members + how to generate>
 // }
 // case class Cons$gen[T] extends Cons[T] {
 //  <members + how to generate>
 // }
```

After the classes and methods have been generated, the miniboxing phase needs to address the final and most tricky part: restoring the type annotations in coherent way. The requirements here are threefold:

- whenever possible, miniboxed versions of methods should be used - for example, if we have a value of type `Cons[Int]` we should use `head$mbox` to access the head of the list instead of `head`, in order to avoid boxing
- along with the method rewiring (e.g. calling `head$mbox` instead of `head`) another step needs to be done: argument and return types have to be adapted: external parameters passed to the method need to be wrapped in `boxToMinibox` and the result needs to be extracted with `miniboxToBox` or another similar primitive. These need to be done in a clear order to maintain type annotations coherent
- finally, arrays must be treated specially, as the Java Virtual Machine bytecode offers distinct array instructions for each value type. The Scala language is invariant in that arrays use the unboxed representations for the value types. Thus miniboxed code cannot expect arrays of `Long`, but needs to dispatch on the type of the array and the type tag to apply the correct operations. These operations must also be treated specially in the plugin.

While giving a full description of the transformation would be useful, it is beyond the scope of this project report. We will conclude our technical discussion on the plugin here.

5.2 The Challenges

The rules of the transformation were not clear from the beginning. When starting work on the project, the plugin prototype was able to generate code that would run for a series of small examples. For more complex examples, the type annotations became corrupt and later phases in the compiler would crash with failed invariants regarding the tree's types.

Going deeper into the problem, we gave a formalization and devised an order for the tree transformations such that type annotations would be kept coherent. Unfortunately, implementing the new set of transformations requires rewriting a significant portion of the plugin.

To be able to deliver the results for the project, we decided to approach the problem from a different perspective: knowing the transformation and how it would work, we will apply the transformation by hand on the benchmarked examples and measure the running time for those. The rest of the project is based on manually-transformed code. This proved beneficial for two reasons:

- we were able to immediately observe problems that the new rules had and adapt them even before writing Scala plugin transformations
- tweaking the transformed code yields faster results than tweaking a transformation and running it on code - by simple tweaks to the code we were able to refine rules and yield 3 time faster code.

Having the examples in the original miniboxing report transformed under the new rules, we set out to reproduce the performance obtained in a more rigorous fashion. The next chapter will present the evaluation methodology and will describe the several iterations we went through to reproduce the original evaluation.

6. Methodology

This section will describe how we took the measurements that guided our decisions on the transformation.

For the benchmark, we tested the same programs as in the original report, namely a mutable resizable-array backed data structure and an immutable linked list. Both data structures are parameterized on the data type. Instead of testing all value types, we decided to only test Integer, but dedicate more time to each test so we decrease the statistical errors. The following methods were tested:

- `ResizableArray.insert()` with 10^6 elements
- `ResizableArray.reverse()` on a data structure with 10^6 elements
- `ResizableArray.find()` 10^5 elements in a list of 10^6 elements
- `LinkedList.insert()` with 10^6 elements
- `LinkedList.hashCode()` on a list with 10^6 elements
- `LinkedList.find()` 10^5 elements in a list of 10^6 elements

The experiments above were done on 4 versions of the same class:

- **Generic** - the generic version of the code, without any transformations. Uses boxed values
- **Specialized** - the class transformed by the existing specialization phase
- **Miniboxed** - the class is transformed by the miniboxing rules
- **Miniboxed with classloader** - the class is transformed by the miniboxing rules and the runtime miniboxing classloader (simulated)

The main concern in this experiment was to remove as much of the noise as possible from the running time measured. As such, we looked for a tool that would automate the data collection and statistic processing for us. We found two candidates: Caliper, a Java benchmarking suite developed by Google and ScalaMeter, a newly-developed regression testing tool capable of finding asymptotic behavior changes in the Scala collections library. Since ScalaMeter is written in Scala and is integrated in our build tool, we decided to use it for micro benchmarking.

ScalaMeter has a number of features to improve JVM micro benchmarks: first of all, it warms up the virtual machine by running tests until the time differences between successive tests become negligible. This way, running the test is guaranteed not to trigger the Just-In-Time compiler or the optimizer. This cuts two noise sources out of three: we're left with garbage collection. Fortunately, ScalaMeter also has a solution for the garbage collector: before the test, it triggers a full garbage collection. If the memory is sufficient, the test should not trigger the garbage collector again. But if it does, ScalaMeter monitors garbage collections and will exclude the measurement from the data set. With these three features, the noise is greatly reduced.

ScalaMeter is integrated with the build tool we are currently using for the miniboxing project. This prompted us to use it from the build tool's command line interface. What we noticed after a couple of runs was that the order in which tests ran was affecting timing. The quick hypothesis was that the classes loaded either affect the polymorphic caches or prevent optimistic inlining. The next step was to run the tests in a separate instance of the VM, automatically spawned by ScalaMeter. And indeed, this has given more stable results, with each result being an average of 50 tests running on 5 instances of the JVM.

7. Evaluation

In this section we will look at the data collected by the micro benchmarks tool. Compared to the original plan of searching for explanations why miniboxing is faster than specialization, we are now going the other way around: the initial results showed a very slow miniboxed class compared to specialization. We successively made hypotheses about the transformation and tweaked it (and implicitly the transformation rules) until we reproduced the original results. This does not give a proper answer to the question of why minibox-transformed classes are faster than specialized ones: theoretically, the tweaks we made to obtain the same results as before could be different and there would be room for more speedup if all tweaks were applied. But in reality, the way the measurements guided us along the way indicates there is a single path that can be taken to improve the numbers.

7.1 Initial Benchmarks

We started by analyzing the 4 versions of the two classes running side by side: generic, specialized, miniboxed and miniboxed with classloader. The results struck us as being biased towards `Array.find()` taking very long, as can be seen in Table 1.

Looking at the code, we find that the only difference between the generic and specialized code is the call to `mboxed_eqeq`:

```
def contains(elem: T): Boolean = {
  var pos = 0
  while (pos < elemCount){
    // == comparison between two Objects
    if (getElement(pos) == elem)
      return true
    pos += 1
  }
  return false
}
```

```
def contains_J(elem: Long): Boolean = {
  var pos = 0
  while (pos < elemCount_J){
    // == comparison between two Objects
    if (mboxed_eqeq(getElement_J(pos), T$Type,
         elem, T$Type))
      return true
    pos += 1
  }
  return false
}
```

The slowdown must be caused by `mboxed_eqeq`, since this is the only different statement. Of course, `getElement`, which is also exercised during `arr.reverse()` can't be held accountable for the slowdown considered. But looking at the code for method `mboxed_eqeq` we find:

```
/*
 * Equality between miniboxed values.
 * Optimized for the case when they
 * have the same type.
 */
@inline final def mboxed_eqeq(x: Minibox, xtag:
     Tag, y: Minibox, ytag: Tag): Boolean = {
  if (xtag == ytag) {
    x == y
  } else {
    minibox2box(x, xtag) == minibox2box(y, ytag)
  }
}
```

The `mboxed_eqeq` method is supposed to be inlined and the fact that we have `xtag` and `ytag` equal should reduce to the then branch in the if statement. Let us first see if it is really inlined by the Scala compiler. As can be seen from Table 2, there's not much difference between inlining or not. So the method is too large to be inlined: instead of a fast comparison, we're calling a method which is evaluating a condition and only then doing what it needs to do: compare two long values. The solution for this is pretty obvious: add another rule that if both miniboxed values are of the same tag, it only needs to compare the values:

```
/*
 * Equality between miniboxed values
 * provided that they have the same type
 */
@inline final def mboxed_eqeq(x: Minibox, y:
     Minibox): Boolean = {
  x == y
}
```

So the lesson is: inlining will only work for very small methods. Don't count on it for bigger methods, be it Scala inlining or JVM inlining.

7.2 The Next Benchmark

The third round of benchmarking in Table 3 looks at the `List.contains` time. Looking at the code, we found a possible optimization:

```
def tail: MBList[Tsp] = tail_J

// ... later:
def contains_J(e: Long): Boolean = {

  @annotation.tailrec def containsTail(list:
       MBList[Tsp]): Boolean =
    if (mboxed_eqeq(head, e))
      true
           // could be tail_J
    else if (list.tail == null)
      false
    else
          // could be tail_J
      containsTail(list.tail)

  containsTail(this)
}
```

It looks like it is going through an extra method call, which should take very little. Changing the rules again to correctly rewire methods resulted in a significant speedup, seen in Table 4. The lesson was: even for small methods, unless they're final, they're expensive to call. Rewire to the most specific method possible.

7.3 Final Benchmarks

Table 4 shows the final benchmark, where the miniboxed w/classloader optimization is faster than specialization. A valid question here is how come `List.contains()` takes so long irregardless of the transformation. The answer is that it's doing pointer chasing, so there's no locality to be exploited for caching data. While the results are still not significantly faster for the miniboxing compared to generic code in the case of list, further optimizations are not possible without sneaking in per-use-case heuristics in the transformation rules.

8. Conclusion

We have seen a proposed transformation to address the memory layout of generic programs, called miniboxing. Building on the experience of a compiler plugin prototype we were able to infer a set of formal rules for the transformation in the general case. Furthermore, we have refined the rules in several iterations to obtian the resulting performance, which in most cases is better than generic code and specialized code. While benchmarking and tweaking, we learned a series of lessons that might be useful in a more general context:

- polymorphic inline cache and the ammount of loaded classes can significanlty affect benchmarks
- inlining won't help if the method is large (JVM/Scala)
- if a method is not inlined, constants won't be propagated inside it, so there will be no optimization for it
- for a very small method, adding a small overhead can turn into a significant slowdown.

References

[1] *Intel 64 and IA-32 Architectures Software Developer's Manual Combined Volumes 3A, 3B, and 3C: System Programming Guide, Parts 1 and 2.*

[2] JSR 14: Add generic types to the javatm programming language. `http://jcp.org/en/jsr/detail?id=14`, May 1999.

[3] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: adding genericity to the java programming language. *SIGPLAN Not.*, 33(10):183–200, Oct. 1998. ISSN 0362-1340. doi: 10.1145/286942.286957. URL `http://doi.acm.org/10.1145/286942.286957`.

[4] M. Coppo. An extended polymorphic type system for applicative languages. In *MFCS*, pages 194–204, 1980.

[5] I. Dragos and M. Odersky. 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*, ICOOOLPS '09, pages 42–47, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-541-3. doi: 10.1145/1565824.1565830. URL `http://doi.acm.org/10.1145/1565824.1565830`.

[6] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous java performance evaluation. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 57–76, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-786-5. doi: 10.1145/1297027.1297033. URL `http://doi.acm.org/10.1145/1297027.1297033`.

[7] M. Grabmller. *Dynamic Compilation for Functional Programs*. PhD thesis, Fakultt IV - Elektrotechnik und Informatik der Technischen Universitt Berlin, 2009.

[8] M. Jonnalagedda. Benchmarking the delite framework. *EPFL Semester Project Report*, 2012.

[9] A. Kennedy and D. Syme. Design and implementation of generics for the .net common language runtime. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, PLDI '01, pages 1–12, New York, NY, USA, 2001. ACM. ISBN 1-58113-414-2. doi: 10.1145/378795.378797. URL `http://doi.acm.org/10.1145/378795.378797`.

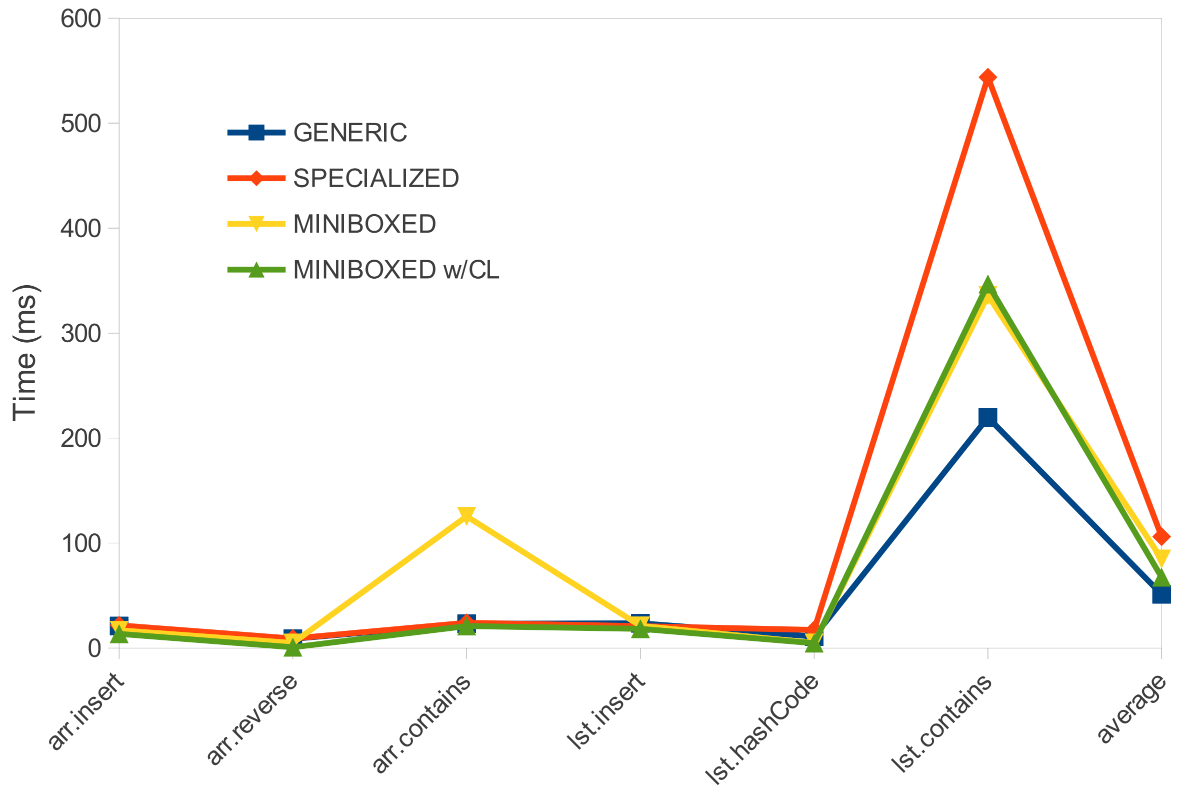
[10] D. Musser and A. Stepanov. Generic programming. In P. Gianni, editor, *Symbolic and Algebraic Computation*, volume 358 of *Lecture Notes in Computer Science*, pages 13–25. Springer Berlin Heidelberg, 1989. ISBN 978-3-540-51084-0. doi: 10.1007/3-540-51084-2_2. URL `http://dx.doi.org/10.1007/3-540-51084-2_2`.

[11] M. Odersky, E. Runne, and P. Wadler. Two ways to bake your pizza translating parameterised types into java. In M. Jazayeri, R. Loos, and D. Musser, editors, *Generic Programming*, volume 1766 of *Lecture Notes in Computer Science*, pages 114–132. Springer Berlin Heidelberg, 2000. ISBN 978-3-540-41090-4. doi: 10.1007/3-540-39953-4_10. URL `http://dx.doi.org/10.1007/3-540-39953-4_10`.

[12] O. Sallenave and R. Ducournau. Lightweight generics in embedded systems through static analysis. *SIGPLAN Not.*, 47(5):11–20, June 2012. ISSN 0362-1340. doi: 10.1145/2345141.2248421. URL `http://doi.acm.org/10.1145/2345141.2248421`.

[13] B. Stroustrup. *The C++ Programming Language, Third Edition*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 3rd edition, 1997. ISBN 0201889544.

[14] C. Talau. Improving the jvm bytecode generated by the scala compiler. *EPFL Semester Project Report*, 2012.



	arr.insert	arr.reverse	arr.contains	lst.insert	lst.hashCode	lst.contains
generic	19.869	9.924	26.891	30.087	13.949	475.397
specialized	20.610	10.161	25.006	27.309	14.784	775.588
miniboxed	19.738	12.102	309.978	27.039	8.043	759.850
miniboxed w/classloader	23.466	12.874	305.506	27.264	8.304	762.524

Table 1. Round 1 of benchmarking (times in milliseconds)

	arr.insert	arr.reverse	arr.contains	lst.insert	lst.hashCode	lst.contains
miniboxed - with inline	19.738	12.102	309.978	27.039	8.043	759.850
miniboxed - w/o inline	17.768	10.944	301.397	22.407	7.197	762.704

Table 2. Round 2 of benchmarking (times in milliseconds)

	arr.insert	arr.reverse	arr.contains	lst.insert	lst.hashCode	lst.contains
generic	19.869	9.924	26.891	30.087	13.949	475.397
specialized	21.228	11.248	30.619	29.681	24.031	811.544
miniboxed	18.041	4.874	130.313	22.962	9.086	845.762
miniboxed w/classloader	11.551	0.925	21.701	22.805	4.780	600.282

Table 3. Round 3 of benchmarking (times in milliseconds)

	arr.insert	arr.reverse	arr.contains	lst.insert	lst.hashCode	lst.contains
generic	20.991	8.792	23.136	23.587	10.980	219.686
specialized	21.622	9.066	24.024	20.859	17.233	543.732
miniboxed	16.893	4.804	125.469	21.160	4.900	335.852
miniboxed w/classloader	13.468	0.745	20.957	18.277	4.640	346.676

Table 4. Final round of benchmarking (times in milliseconds)