VIETNAM NATIONAL UNIVERSITY - HO CHI MINH CITY
UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

BACHELOR OF ENGINEERING THESIS

SCALA BENCHMARKING SUITE - SCALA PERFORMANCE REGRESSION PINPOINTING

Advisor: M.Sc. Aleksandar Prokopec
Dr. Phung H. Nguyen
Prof. Martin Odersky

Examiner: Dr. Tho T. Quan

---o0o---

Student: Ngoc Duy Pham (50700393)

January 2012

Scala Benchmarking Suite -

Scala Performance Regression Pinpointing

Bachelor of Engineering Thesis

Scala Benchmarking Suite -

Scala Performance Regression Pinpointing

By

Ngoc Duy Pham

Department of Computer Science

Faculty of Computer Science and Engineering

Ho Chi Minh city University of Technology

January 2012

Guarantee

I guarantee that beside the results referenced from other research works which have been cited in this thesis, all works presented in this report are done by me and there are not any contents of this thesis which were submitted to get certificate from this universities or others.

Acknowledgement

My deepest thanks to Alex, M.Sc. Aleksandar Prokopec. He has made available his support in a large number of ways. During his supervision on the project, he taught me from Scala to functional programming and even coding convention. His friendliness and encouragement helped build up my anxiousness to complete the project successfully.

My honor gratitude to Dr. Phung Nguyen. He gave me the inspiration in programming language, helped me build firm background knowledge for this project, taught me about scientific writing and much more.

My special thanks to Dr. Tho Quan and Prof. Martin Odersky. They gave me the valuable opportunity to come and work on this thesis at LAMP – EPFL.

My sincerest and sweetest thanks to Mr. Hoang Hai Ly. He is rich. He gave us his food, taught us how to cook. He was our Switzerland tour guide when we had free time. He taught us programming when he had free time. We used to watch movies and cook together at weekends. He is like an elder brother and I wish he were.

And last but not least, many thanks to Mr. Quoc Viet Hung Nguyen for finding us accommodation, leading our first steps in Switzerland and letting us eat at his place when we were not able to cook.

Abstract

Scala is a programming language which integrates features of object-oriented and functional programming with concise syntax. Currently, Scala grows dramatically and thereby needs a benchmarking tool to guarantee its performance and reliability.

Scala Benchmarking Suite (SBS) is a tool developed to satisfy the request above. It allows users to write micro-benchmarks detecting the performance regression with statistical rigor in a way just as simple as the way they write unit tests. In addition, users can have SBS profile typical metrics during benchmark runs, such as method invocations, number of boxings, memory consumption, etc.

And finally, SBS comes with the implementation of a bottleneck finding algorithm, which combines bytecode instrumentation and statistically rigorous performance regression detection. The algorithm has the ability to dynamically and programmatically point out the piece of code that causes a performance drop without the needs for manual effort from users.

Contents

List of Figures

List of Listings

Lists of Algorithms

Chapter 1

Introduction

1.1. Motivation

The current growth of the Scala programming language is enormous, demonstrated by the number of leading companies that are successfully using Scala for critical business applications. It is common knowledge that more companies like Twitter, LinkedIn, Foursquare, the Guardian, Morgan Stanley, Credit Suisse, UBS, HSBC and Trafigura are now using Scala[1]. As a consequence, there is the high demand for performance and reliability to be guaranteed. At the time, there is no support for effectively using micro benchmarks to assess the affection to performance of small changes to Scala standard library and compiler. It's worth having a built-in tool integrated to the language project which does all the performance tests at nightly builds to detect all kinds of performance regression.

During the time spent to implement Scala Benchmarking Suite, we realized that all the manual work to find out the performance bottleneck in a program is a pain, even after its existence has been detected. The problem impulses us to develop a methodology to help developers spare most of the manual effort. The algorithm, which we call *performance regression pinpointing*, tries to dynamically point out the piece of the program that causes the performance regression by using the combination of bytecode instrumentation and *statistically rigorous performance regression detection* methodology (see section 2.3).

[1] http://www.scala-lang.org/node/10923

1.2. Contribution

This thesis makes the following contributions:

- We provide publicly available software, called Scala Benchmarking Suite, which will soon be contributed as a `package` in the `trunk` of the project Scala programming language. The tool is used to do benchmarking on the Scala programming language. It uses a statistically rigorous methodology to evaluate and detect regression on the performance of a Scala program. It can also profile some typical metrics and dynamically find out the bottleneck inside a code snippet. It enables users to write benchmarks as the way unit tests are written now - a single Scala source file or a directory contains many of them - and runs them individually or by groups.
- We introduce a methodology, called *performance regression pinpointing*, to dynamically programmatically point out the performance bottleneck lies inside a Scala program. It is able to do so by combining the performance evaluation using instrumentation technique and the performance comparison using *statistically rigorous performance detection* methodology.
- Comes along with *performance regression pinpointing* is the technique to do instrumentation on the bytecodes generated by compiling Scala programs. The technique is shown that finds performance-relevant parts of the bytecodes from all the automatically generated classes and do instrumenting with an instrumentation library for Java.

1.3. Contents overview

The thesis is organized as follows:

- In Chapter 1, we give the motivation, contribution and the organization of this thesis
- Chapter 2 introduces the Scala programming language along with some issues about dynamic compilation languages that may cause indeterminism in performance measurements. It also briefly describes how to add statistical rigor to performance evaluation and performance regressions detection.

- Chapter 3 introduces the tool Scala Benchmarking Suite – a benchmarking tool on the Scala programming language. It lists out the current abilities and features as well as the design description and implementation of the suite.
- In Chapter 4, we introduce our methodology and algorithms to dynamically find a performance bottleneck inside a code snippet. We give overviews about a few prevalent methodologies and their drawbacks in the comparison to ours. We also describe the techniques to implement the methodology which includes Scala bytecode instrumentation with Java instrumentation library and keeping previous bytecode version collected from the previous builds.
- In the last chapter, we conclude our work and discuss future directions.

Chapter 2

Background

This section gives a brief introduction about Scala programming language and highlights its advanced features. It also describes the basic ideas about dynamic compilation languages, which do most of the recompilations and optimizations at runtime, which lead to uncertainties and indeterminism in programs' performance. Finally, this section summarizes the methodology which uses statistics theory as a rigorous data analysis approach for dealing with the non-determinism and the experiment designs to evaluate performances.

2.1. Scala programming language

Scala [1] is the programming language designed by prof. Martin Odersky – the co-designer of Java Generics and the main author of the current generation `javac` compiler. Scala determines itself to be a statically typed, scalable language and a fusion of object-oriented language and functional one.

This section provides a glimpse of the Scala programming language altogether with its advanced features. We assume that readers have the basic knowledge about programming languages (mostly concern object-oriented and functional ones) and Java.

Classes and objects

Scala is an *object-oriented language* in the sense that every value (and even method) is an object. Scala's class definition syntax is borrowed from Java, except that Scala classes can have parameters and the main constructors can be defined directly in the class body.

A Scala program is a set of Scala classes that represent abstract things which can perform operations, changing states, and communicating with other ones in the system. For this *abstraction* to be more powerful in reality, Scala comes along with *polymorphism*, which

is a concept describes the fact that even though classes are derived or inherited from the same parent class, each derived class will have its own behavior performs different functions.

`object` is a language primitive which represents a singleton class – class that has only once instance. Objects hold all the definitions considered `static` in Java. Therefore, the entry of a Scala program, a main method is defined in an `object`. An `object` which has the same name and is declared in the same source file with a `class` is called the companion object of the class. It has the accessibility to all the members of the class including the `private` members also. For example, in Listing 2.1 method `extract` of object `Companion` can access to the private member `field` from class `Companion`:

```
class Companion {

  private val field = 1

}

object Companion {

  def extract(obj: Companion) = println(obj field)

}

object ProgramEntryPoint {

  def main(args: Array[String]) {
    // not compile
    // val obj = new ProgramEntryPoint

    val c = new Companion
    Companion extract c
  }

}

// program output: 1
```

Listing 2.1 – Example: Scala object

Traits

Traits are a fundamental unit of code reuse in Scala. Scala's trait in some aspect may resemble Java's interfaces, many of which are able to be mixed-in one class definition. The biggest difference distinguishes a trait from an interface is that traits can have methods implemented.

Traits are Scala's approach to enable multiple-inheritance. A concrete class can be composed by mix-in composition from many traits. The *diamond problem*[2] does not exist in Scala because traits have no constructor and the concrete implementation of a method is selected based on the mix-in order. In Listing 2.2, the creation of `Centaur` illustrates a case of multiple inheritance in Scala. `Centaur` is created by mix-in the two traits `Human` and `Horse` which have the same super class `Creature`. Both `Human` and `Horse` have their own concrete method `run`, but based on the mix-in order, `Centaur` inherits its `run` from `Horse`.

```
trait Creature {

  def run: Unit

}

trait Human extends Creature {

  override def run = println("walk with 2 feet")

}

trait Horse extends Creature {

  override def run = println("gallop with 4 feet")

}

class Centaur extends Human with Horse

object MixinComposition {

  def main(args: Array[String]): Unit = {
    new Centaur run
  }

}

// program output: gallop with 4 feet
```

Listing 2.2 – Example: Scala trait

Another big use of traits is to widen thin interfaces to rich ones. That solves one of a big trade-off in object-oriented design about the selection between providing a large number of methods for callers' convenience and the heavy coding work left for implementers.

[2] http://en.wikipedia.org/wiki/Diamond_problem

Built-in control structures

Scala has a modest number of built-in control structures all of which are `if`, `while` `for`, `try`, `match` and function call. The reason why is that Scala includes function literals to give the developers the ability to create their own using *higher-order functions*[3], *currying*[4] and *call-by-name* parameters[5]. In Listing 2.3, there is an example of creating new control structure called `afterHello`. The control structure executes the body block after printing a "Hello!" message:

```
// afterHello is a higher-order function
// and operation is a call-by-name variable
def afterHello(operation: => Any): Any = {
  print("Hello!")
  operation
}

// this is how to use the new control structure
afterHello {
  println(" How do you do!")
}

// program output: Hello! How do you do!
```

Listing 2.3 – Example: new control structure

Almost all of Scala's control structures result in values. This is the approach of functional languages in which the programs are the processes to compute values. This facility results in simpler code and prevents bugs where the value of a variable is modified unexpectedly.

Functions

The term *function* in Scala has a larger meaning than one in Java which literally means *method*. A function in Scala may be a method, a function defined inside a method body (local function) or an argument passed to a function (first-class function). Example of those above in Listing 2.4:

```
class ScalaFunction {

  // foo is a method
  def foo(): Unit = {
    // this is a local function of foo
    def infoo() = println("defined inside foo's body")
```

[3] http://en.wikipedia.org/wiki/Higher-order_function
[4] http://en.wikipedia.org/wiki/Currying
[5] http://en.wikipedia.org/wiki/Call_by_name

```
    infoo()
    println("foo")
  }

  // f is a first-class function
  def bar(f: Int => Int): Int = f(0)

}
```

Listing 2.4 – Example: Scala function

All the types of functions mentioned above illustrate a fundamental characteristic of functional languages: every function is a value. Like other language in the field of *functional programming*, Scala supports first-class and higher-class functions, *partially applied function*[6] and *currying* altogether with *closures*[7] and light-weight syntax to define anonymous *function literals*[8].

Case classes and pattern matching

Pattern matching[9] is a fundamental tool in functional programming. Being also an object-oriented language, Scala can do pattern matching on class as a convenient replacement for type tests and type casts. A simple example is shown in Listing 2.5. The case classes imitate data structures used to parse the statements of a programming language.

```
trait Statement
case class Assignment(name: String) extends Statement
case class IfElse(cond: Boolean) extends Statement
case class While(block: Statement) extends Statement

def recognize(statement: Statement) = statement match {
  case Assignment(n) => println("Assign to " + n)
  case IfElse(cond)  => println("if " + cond)
  case While(block)  => println("while" + block)
  case _             => throw new Exception("Not recognize")
}
```

Listing 2.5 – Example: Scala case class and pattern matching

Scala's built-in case classes and support for pattern matching models *algebraic type*[10] used in many functional programming languages. Case classes are regular classes which

[6] http://en.wikipedia.org/wiki/Partial_application
[7] http://en.wikipedia.org/wiki/Closure_(computer_science)
[8] http://en.wikipedia.org/wiki/Anonymous_function
[9] http://en.wikipedia.org/wiki/Pattern_matching
[10] http://en.wikipedia.org/wiki/Algebraic_data_type

expose their constructor parameters for outside view. `case class` keyword automatically adds a factory method in the *companion object*[11] of the class and also an extractor method to support pattern matching.

Collections

`scala.collections` package is the set of all the pre-defined data structures in the Scala standard library. They consist of sequences, sets, maps and other data structures express the vast variety collections of data elements. They are easy to use, concise, safe, and fast and are powerful building blocks rather than a bunch of ill-organized utilities. Scala collections are distinguished into *mutable* and *immutable*[12] data types which are respectively familiar with imperative and functional programming styles.

Support for XML[13]

Scala has built-in support for XML. In Scala, there exist the XML literals and mechanisms for constructing them. XML can be processed with pattern matching or can be taken part by the methods that have already been defined as members of XML elements. Finally, Scala has the library routines that support all the conversions back and forth between XML and byte stream or String literals to make it easy for loading – saving and data serialization – deserialization.

```
// a XML literal
val xml = <example>text</example>

// pattern matching
xml match {
  case <example>{content}</example> => println(content)
  case _                            => throw new Exception
}

// storing to file
scala.xml.XML.saveFull("file.xml", xml, "UTF-8", true, null)

// loading from file
val loaded = scala.xml.XML.loadFile("file.xml")
```

Listing 2.6 – Scala example: XML

Listing 2.6 illustrates Scala's ability of handling XML. A XML value name `xml` can be declared just like an `Int` or `Double`. Its elements can be easily extracted by pattern

[11] http://daily-scala.blogspot.com/2009/09/companion-object.html
[12] http://en.wikipedia.org/wiki/Immutable_object
[13] http://en.wikipedia.org/wiki/XML

matching. Also, the processes of storing and loading `xml` have been already implemented in Scala standard library.

Actors and concurrency

When it becomes necessary to design a program to express things happened independently in parallel, Scala provides the mechanism for that concurrency, which is *actors*. Actors and message passing model the interactions between two or more entities in the system. Unlike Java's support for concurrency (threading and synchronization which uses the shared fragments of memory) Scala's actors library avoids bugs on asynchronous situations and deadlocks with the share nothing approach.

In Listing 2.7, `Starter` and `Replier` represent two of the most simple Scala actors. The three objects `Forth`, `Back` and `Stop` are the messages passed between `Starter` and `Replier`, they are defined as `case objects` for it to be able to apply pattern matching on them. A `Starter` sends four messages `Forth` to a `Replier` and expects to receive four `Back` messages, one after each. In the final step, it sends `Replier` a `Stop` message to stop interacting.

```
import scala.actors.Actor
import scala.actors.Actor._

case object Forth
case object Back
case object Stop

class Starter(replier: Actor) extends Actor {

  def act() {
    replier ! Forth
    (0 util 3) foreach {
      _ => receive {
        case Back => {
          println("Back")
          replier ! Forth
        }
      }
    }
    replier ! Stop
  }

}

class Replier extends Actor {

  def act() {
```

```
    while (true) {
      receive {
        case Forth => {
          println("Back")
          replier ! Forth
        }
        case Stop => {
          println("Stop")
          exit()
        }
      }
    }
  }

}
```

Listing 2.7 – Scala example: Actor

Combining with Java

The interaction between Scala and Java is seamless, due to the fact that Scala is most compiled to Java bytecodes and run on JVM. In a more technical aspect, a Scala program is just like any other normal Java programs. The difference is Scala runtime environment is a JVM environment added by a hierarchy of the classes from Scala standard library.

In Scala, one can easily call Java APIs without worrying about the incompatibility. There is actually a little notice about difference in syntax that makes Java in Scala looks just like the "original" Scala. For example, a call to Java's `Thread.sleep()` method is as simple as shown in Listing 2.8.

```
object A {

  def run = java.lang.Thread.sleep(5000)

}
```

Listing 2.8 – Scala example: Using a Java's class

In Java, use of Scala classes may encounter some difficulty. The reason is the way Scala compiler compiles Scala source codes into bytecodes. Additionally, there are things in Scala that Java does not have. Those require some deep knowledge to be overcome (a few of them will be described in section 4.4).

2.2. Benchmarking on Java virtual machine

See articles [4, 5] for further details.

2.2.1. Dynamic compilation

The compilation process for a dynamically compiled language like Java or Scala is different from that of statically compiled languages like C. Compilers for statically compiled languages convert source code directly to machine code that can be immediately executed on the target platform. But the cost to pay is that different hardware platforms require different compilers. Meanwhile, dynamically compiled languages require a specific runtime environment (typically a virtual machine) for each different hardware platform but they gain a vast benefit: all the effort to recompile the whole user project when new platforms come into place is gone – compile once run everywhere. Compilers for dynamically compiled languages convert the source code into portable runtime code, which consists of virtual machine instructions for the runtime environment. Unlike those compilers for statically compiled languages, compilers for statically compiled languages do very little optimization - the optimizations are performed instead in the runtime when the program is executed. Runtime environments using dynamic compilation typically have programs run slowly for the first few time intervals, and then after that, most of the compilation and recompilation is done and it runs more quickly.

Just-in-time compilation

The first generation of JVMs was entirely an interpreter. That JVM interpreted the bytecodes rather than compiling them to machine code and executing the machine code directly. But interpretation simply is slow. Nowadays, JVMs used just-in-time (JIT) compilers to speed up execution. It converts all bytecodes into machine code before execution, but does so in a lazy fashion: The JIT only compiles a code path when it knows that code path is about to be executed. This approach allows the program to start up more quickly, as a lengthy compilation phase is not needed before any execution can begin. JIT removed the overhead of interpretation but to avoid a significant startup penalty for Java applications. The JIT compiler has to be fast to prevent influencing the actual performance of the user program too much, which means that it could not spend as much time doing optimization.

HotSpot dynamic compilation

The Java HotSpot Virtual Machine is a JVM for desktops and servers. It is a core component of the Java SE platform. It implements the Java Virtual Machine Specification and

includes dynamic compilers that adaptively compile Java bytecodes into optimized machine instructions and efficiently manages the Java heap using garbage collectors. Based upon the platform configuration, it will select a suitable compiler, Java heap configuration, and garbage collector.

The HotSpot execution process combines interpretation, profiling, and dynamic compilation. Rather than convert all bytecodes into machine code before they are executed, HotSpot first runs as an interpreter and only compiles the "hot" code - the code executed most frequently. As it executes, it gathers profiling data, used to decide which code sections are being executed frequently enough to merit compilation. No time is wasted compiling code that will execute infrequently, and the compiler can spend more time on optimization of hot code. Furthermore, by deferring compilation, the compiler has access to profiling data, which can be used to improve optimization decisions, such as whether to inline a particular method call.

HotSpot has two compilers: the client compiler and the server compiler:

- The client compiler has been optimized to reduce application startup time and memory footprint, employing fewer complex optimizations than the server compiler, and accordingly requiring less time for compilation.
- The server compiler has been optimized to maximize peak operating speed, and is intended for long running server applications. It can perform many of the standard optimizations found in static compilers, such as code hoisting, common sub-expression elimination, loop unrolling, range check elimination, dead-code elimination, and data-flow analysis, as well as a variety of optimizations that are not practical in statically compiled languages, such as aggressive inlining of virtual method invocations.

Continuous recompilation

After a code path is interpreted a certain number of times, it is compiled into machine code. But the JVM continues profiling, and may recompile the code again later with a higher level of optimization if it decides that code path is particularly hot or future profiling data suggests opportunities for additional optimization. The JVM may recompile the same bytecodes many times in a single application execution.

Dynamic deoptimization

Many standard optimizations can only be performed within a "basic block" and so inlining method calls is often important to achieve good optimization. By inlining method calls, not only is the method call overhead eliminated, but it gives the optimizer a larger basic block to optimize, with substantial opportunity for dead-code optimizations.

On-stack replacement

The initial version of HotSpot performed compilation one method at a time. A method was deemed to be hot if it cumulatively executed more than a certain number of loops, which it determined by associating a counter with each method and incrementing that counter every time a backward branch was taken. However, after the method was compiled, it did not switch to the compiled version until the method exited and was re-entered -- the compiled version would only be used for subsequent invocations. The result, in some cases, was that the compiled version was never used, such as the case of a compute-intensive program, where all the computation is done in a single invocation of a method. In such a situation, the heavyweight method may have gotten compiled, but the compiled code would never be used.

More recent versions of HotSpot use a technique called *on-stack replacement* (OSR) to allow a switch from interpretation to compiled code (or swapping one version of compiled code for another) in the middle of a loop.

2.2.2. Memory management

Java HotSpot Virtual Machine performs automatic memory management therefore helps Java developers avoid the complexity and inconvenience of memory allocation as well as deallocation. To achieve this target, the memory available for user's programs in runtime is well-organized and automatic managed by a program called *garbage collector*. With garbage collection run along at runtime, developers no longer worry about errors such as *memory leaks* and *dangling references*.

The memory (heap space) is organized by HotSpot JVM into generations, that is, separate pools holding objects of different ages. The purpose is for a garbage collection algorithm named *generation garbage collection* to be used. The algorithm exploits observations regarding software applications written in object-oriented languages, known as *weak generation hypothesis*, which says that:

- Almost all objects do not live long, which means they will soon no longer be referenced
- There are few references from older to younger objects exist

The most widely-used generation organization configuration consists of:

- *Young generation* – usually small and likely stores many of short-lived objects. When a new object is created, it is placed in this memory fragment.
- Old or *tenured generation* – stores objects that meet some promotion criteria, such as having survived a certain number of garbage collections.
- *PermGen* or permanent generation – holds data needed by the virtual machine to describe objects that do not have equivalence at the Java language level (such as objects describing classes and methods as well as the classes and methods themselves). This generation is never garbage collected.

Garbage collection occurs in each generation (except the PermGen) when the generation fills up. For the young generation, the garbage collection strategy is the minor collection. The strategy is fast since collections on young generation occur frequently and for the pause of user's program running to be short. The garbage collection used when *tenured generation* is full is called the *major collection* or *full collection*. It is not only attempt to collect the objects allocated in *tenured generation*. When it runs, all generations are collected.

Garbage collection is quite a complex task taking time and resources of its own, therefore, may heavily influence the overall performance of the user's programs.

2.2.3. Microbenchmark

The traditional way to determine if an approach is faster than another one is to write a small benchmark program, often called a microbenchmark. Writing - and interpreting - benchmarks is far more difficult and complicated for dynamically compiled languages than for statically compiled ones. In many cases, microbenchmarks written in dynamic compilation language don't give the expected results.

HotSpot JIT will continuously recompile Java bytecodes into machine code as the program runs, and recompilation can be triggered at unexpected times by the accumulation of a certain amount of profiling data, the loading of new classes, or the execution of code paths that have not yet been traversed in already-loaded classes. Timing measurements in the face

of continuous recompilation can be very noisy and misleading, and it is often necessary to run the source code for very a long time before obtaining useful performance data.

JVM warming up

Measuring the performance of on approach generally means measuring its optimized compiled implementation performance, not interpreted one. That requires "warming up" the JVM - executing the target operation enough times that the compiler will have had time to run and replace the interpreted code with compiled code before starting to measure the desired execution performance.

With today's dynamic compilers, it is a lot more difficult. The compiler runs at an unpredictable time, the JVM switches from interpreted to compiled code at will, and the same code path may be compiled and recompiled more than once during a run.

Garbage collection

Garbage Collection is another element that can badly distort timing results - a small change in the number of iterations could mean the difference between no GC and one GC, skewing the "time per iteration" measurement. If the benchmarks run with -verbose:gc JVM option, timing data can be adjusted accordingly to the quantity of time spent in garbage collection. Even better, ensuring that many garbage collections are triggered, more accurately amortizing the allocation and garbage collection cost.

Dead-code elimination

One of the challenges of writing good benchmarks is that optimizing compilers are adept at spotting dead code - code that has no effect on the outcome of the program execution. But benchmark programs often don't produce any output, which means some, or all, of the source code can be optimized away, at which point the result measurement is less execution than what it should be. That dead-code optimization that makes such short work of the benchmark (possibly optimizing it all away) is not going to do quite as well with code that actually does something.

Because runtime compilation uses profiling data to guide its optimization, the JIT may well optimize the test code differently than it would do to real code. As with all benchmarks, there is a significant risk that the compiler will be able to optimize away the whole thing, because it will realize that the benchmark code neither actually do anything nor produce any result that is used for anything. Writing effective benchmarks requires fooling the compiler into not pruning away code as dead, even though it really is.

For example, supposed that we want to measure the performance of the function `scala.Math.sqrt()`, we intend to do so by repeating the operation for 500.000 times. Unfortunately, with our first implementation in `object Wrong` in Listing 2.9, the JIT will recognize that all the calls to `sqrt()` are good for nothing and optimize it away. Therefore, the running time result is too small to be mentioned. We will fix that in `object Right` by fooling the JIT that the values computed using `sqrt()` are used to update the public `var str` which may be read by another object in the future. That makes the 500.000 calls to sqrt() remain at runtime so that we can measure their performance.

```
// this does not work
object Wrong {

  def main(args: Array[String]): Unit = {
    // these 500.000 calls to Math.sqrt()
    // will be optimized away
    (0 until 500 * 1000) foreach (Math sqrt _)
  }

}

// this will work
object Right {

  var str = ""

  def main(args: Array[String]): Unit = {
    (0 until 500 * 1000) foreach (i =>
      str = "" + (Math sqrt i))
  }

}
```

Listing 2.9 – Exmample: Dead code elimination

In addition, the problem is not strictly that the optimizer is optimizing away the benchmark, but that it is able to apply a different degree of optimization to one alternative than to another, and that the types of optimizations that it can apply to each alternative would not likely be applicable in real-world code.

The Heisenberg principle

The performance of operation X is being measured, so there should be nothing to run besides X. But often, the result is a do-nothing benchmark, which the compiler can optimize away partially or completely, making the test run faster than expected. If extraneous code Y is

put into the benchmark, the performance of X + Y is to being measured, introducing noise into the measurement of X, and worse, the presence of Y changes how the JIT will optimize X. Writing a good microbenchmark means finding that elusive balance between not enough filler and dataflow dependency to prevent the compiler from optimizing away the entire program, and so much filler that truthfully performance gets lost in the noise.

2.3. Statistically rigorous performance regression detection

Performance of a program runs on JVM platform is not trivial to benchmark because it is affected by various factors (some of the most commons are described in section 2.2). JVM uses timer-based sampling to drive the JIT compilation and optimizations. That methodology may lead to non-determinism and execution time variance: different executions of the same program may result in different samples being taken and, by consequence, different methods being compiled and optimized to different levels of optimization. There exist many other sources of non-determinism such as thread scheduling in timeshared and multiprocessor systems, garbage collections, and various system effects like system interrupts etc.

Another issue on performance benchmarking is that, researchers and/or software developers use a wide variety of Java performance evaluation methodologies. These methodologies differ from each other in a number of ways. Some report average performance over a number of runs of the same experiment; others report the best performance observed; yet others report the worst. Some iterate the benchmark multiple times within a single JVM invocation; others consider multiple JVM invocations and iterate a single benchmark execution; yet others consider multiple JVM invocations and iterate the benchmark multiple times. All these prevalent methodologies can be misleading, and can even lead to incorrect conclusions. The reason is that the data analysis is not statistically rigorous.

This section briefly describes how to use statistics theory as a rigorous data analysis approach for dealing with the non-determinism in managed runtime systems as well as the experiment designs to evaluate the benchmarks' performances advocated in [2]:

- Adding statistical rigor to performance evaluation studies of managed Java runtime systems. The motivation for statistically rigorous data analysis is that statistics, and in particular confidence intervals, enable one to determine whether differences observed

in measurements are due to random fluctuations in the measurements or due to actual differences in the alternatives compared against each other.

- Performance evaluation methodologies for start-up and steady-state performance, and the following methods to detect statistically significant difference in the achieved performances of different alternatives which can be used to detect performance regressions of Scala programs.

2.3.1. Confidence interval for the means

In each experiment, a number of samples is taken from an underlying population. A confidence interval for the mean derived from these samples then quantifies the range of values that have a given probability of including the actual population mean. The confidence interval $[c_1, c_2]$ is defined such that the probability of μ being between c_1 and c_2 (i.e $c_1 \leq \mu \leq c_2$) equals the confidence level of $1 - \alpha$; α is called the significance level. Let:

- μ is the *population mean*[14], that is the expected accuracy value we want to measure
- σ^2 is the *population variance*[15] that is a measure of how far a set of sample values is spread out from the mean. σ is called the *standard deviation*[16], the "average" of the all the differences of every value from the mean
- n is the number of samples taken

The sample mean is the average value of all the collected samples, computed as

$$\bar{x} = \frac{\sum_{i=1}^{n} x_i}{n}$$

The *sample standard deviation*[17] s is the most common estimator for σ, calculated as the squared root of the sum-of-squares of all the subtractions of each sample by the sample mean, divided by the number of samples subtracted by 1:

$$s = \sqrt{\frac{\sum_{i=1}^{n} (x_i - \bar{x})^2}{n - 1}}$$

The following mathematical formulas are used to calculate the confidence interval for the respective case of:

[14] http://en.wikipedia.org/wiki/Mean#Population_and_sample_means
[15] http://en.wikipedia.org/wiki/Variance#Population_variance_and_sample_variance
[16] http://en.wikipedia.org/wiki/Standard_deviation
[17] http://en.wikipedia.org/wiki/Standard_deviation#With_sample_standard_deviation

When the number of measurements is large (n ≥ 30)

$$c_1 = \bar{x} - z_{1-\alpha/2}\frac{s}{\sqrt{n}}$$

$$c_2 = \bar{x} + z_{1-\alpha/2}\frac{s}{\sqrt{n}}$$

The value $z_{1-\alpha/2}$ is defined such that a random variable Z that is Gaussian distributed with mean $\mu = 0$ and variance $\sigma^2 = 1$ (normal distribution), obeys the following property: the probability of a variable Z is less than or equals $z_{1-\alpha/2}$ equals to $1 - \alpha / 2$. It is usually pre-computed.

When the number of measurements is small (n < 30)

$$c_1 = \bar{x} - t_{1-\alpha/2;\, n-1}\frac{s}{\sqrt{n}}$$

$$c_1 = \bar{x} + t_{1-\alpha/2;\, n-1}\frac{s}{\sqrt{n}}$$

The value $t_{1-\alpha/2;\, n-1}$ is defined such that a random variable T that follows Student's t distribution with n – 1 degrees of freedom, obeys the following property: the probability of a variable T is less than or equals $t_{1-\alpha/2;\, n-1}$ equals to $1 - \alpha / 2$. It is also usually pre-computed.

2.3.2. Startup performance measuring

The goal of measuring start-up performance is to measure how quickly a Java Virtual Machine can execute a relatively short-running Java program. There are two key differences between startup and steady-state performance. First, startup performance includes class loading whereas steady-state performance does not, and, second, startup performance is affected by JIT compilation, substantially more than steady-state performance.

For measuring startup performance, use a two-step methodology:

- First, measure the execution time of multiple JVM invocations, each VM invocation running only one single benchmark iteration. This results in p measurements x_{ij} with $1 \leq i \leq p$ and $j = 1$
- Then, compute the confidence interval for a given confidence level as described in Section 3.2. If there are more than 30 measurements, use the standard normal z -statistic; otherwise use the Student t -statistic

In practice, the first JVM invocation in a series of measurements may change system state that persists past this first JVM invocation. To reach independence, the first JVM invocation is discarded and only the subsequent measurements are retained.

2.3.3. Steady-state performance measuring

Steady-state performance concerns long-running applications for which start-up performance is less interested. Since most of the JIT compilation is performed during start-up, steady-state performance suffers less from variability due to JIT compilation. However, the other sources of non-determinism, such as thread scheduling and system effects, still remain under steady-state, and thus need to be considered.

There are two issues with quantifying steady-state performance. The first issue is to determine when steady-state performance is reached. The second issue with steady-state performance is that different JVM invocations running multiple benchmark iterations may result in different steady-state performances. Different methods may be optimized at different levels of optimization across different JVM invocations, changing steady-state performance.

To address these two issues, the following methodology is used for quantifying steady-state performance. Consider p JVM invocations (each for one alternative), each running at most q benchmark iterations:

- For each JVM invocation i of the p invocations, determine the iteration s_i ($s_i \leq q$) where steady-state performance is reached, i.e., once the *coefficient of variation* (CoV) of the k iterations ($s_i - k$ to s_i) falls below a preset threshold, say 1% or 2%.
- For each invocation, compute the mean x_i of the k benchmark iterations under Steady-state:

$$\bar{x}_i = \sum_{j = s_i - k}^{s_i} x_{ij}$$

- Compute the confidence interval for a given confidence level across the computed means from the different JVM invocations. The overall mean equals

$$\bar{x} = \sum_{i = 1}^{p} \bar{x}_i$$

Where k is the number of measurements we want to retain per invocation

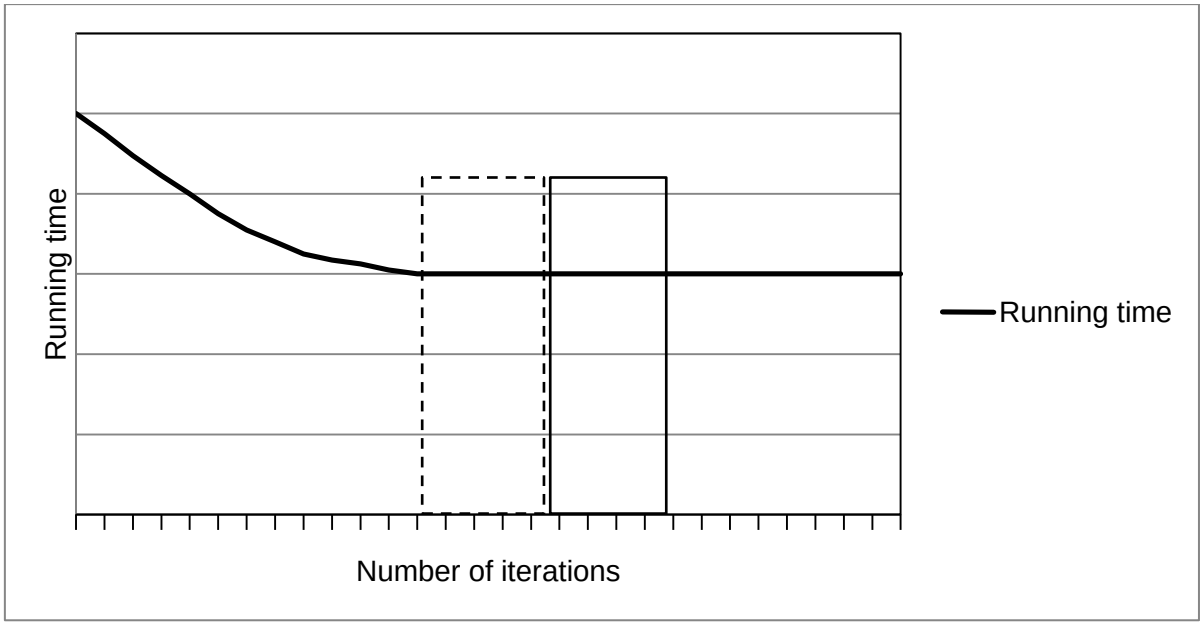

Figure 2.1 – Steady-state measurement process

In practice, the actual measurement process is illustrated in Figure 2.1. The running time of the benchmark is high in the beginning, and then the benchmark runs faster through all of the JIT optimizations. The measurements inside the dash line window are the measurements from iterations $s_i - k$ to s_i. They are used for determining that the benchmark running has reached its steady-state. That target is achieved by calculating its *coefficient of variations* (CoV) which should fall below the preset threshold. The measurements intended to be the result is the series of measurements in the next k iterations, i.e the measurements inside the continuous line window. The purpose of continuing collecting the next k iterations is to avoid the first few measurement results that maybe have not reached steady-state yet.

After achieving a number of series of running time, one of the two following statistics tests is applied to detect statistically significant differences among those alternatives.

2.3.4. Compare two alternatives

The simplest approach to comparing two alternatives is to determine whether the confidence intervals for the two sets of measurements overlap. If they do overlap, the difference seen in the mean values is possibly due to random effects. If the confidence intervals do not overlap, however, we conclude that there is a statistically significant difference with the probability of $1 - \alpha$ (this also means that there is a probability of α

suggests that the difference between the two alternatives is caused by random effects). The statistics necessary to be computed described as follows. Let:

- n_1, $\bar{x}_1$, s_1 are respectively the number of measurements, sample mean and sample standard deviation of the first alternative
- n_2, $\bar{x}_2$, s_2 are respectively the number of measurements, sample mean and sample standard deviation of the second alternative

The difference of the means is

$$\bar{x} = \bar{x}_1 - \bar{x}_2$$

The standard deviation of the difference of the means is

$$s_x = \sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}$$

The following mathematical formulas are used to calculate the confidence interval for the respective case of:

When $n_1 \geq 30$ and $n_2 \geq 30$

$$c_1 = x - z_{1-\alpha/2} s_x$$

$$c_2 = x + z_{1-\alpha/2} s_x$$

When $n_1 < 30$ or $n_2 < 30$

$$c_1 = x - t_{1-\alpha/2;\, n_{df}}\, s_x$$

$$c_2 = x + t_{1-\alpha/2;\, n_{df}}\, s_x$$

With n_{df} is called the degrees of freedom, computed as

$$n_{df} = \frac{(\frac{s_1^2}{n_1} + \frac{s_1^2}{n_2})^2}{\frac{(s_1^2/n_1)^2}{n_1 - 1} + \frac{(s_2^2/n_2)^2}{n_2 - 1}}$$

If $[c_1; c_2]$ includes zero, we can conclude, at the confidence level chosen, that there is no statistically significant difference between the two alternatives.

2.3.5. Compare many alternatives

In the case the number of alternatives is larger than 2, a more general and more robust technique is applied. The technique is called Analysis of Variance (ANOVA) test. It separates the total variation observed in (i) the variation observed within each alternative, which is assumed to be a result of random effects in the measurements, and (ii) the variation between the alternatives. If the variation between the alternatives is larger than the variation within each alternative, then it can be concluded that there is a statistically significant difference between the alternatives.

To explain the ANOVA test, let:

- k is the number of alternatives to be compared
- n is the number of measurements for each alternative
- y_{ij} is the j^{th} performance value measured for alternative i ($i \leq k, j \leq n$)

The mean of each alternative is computed as

$$\bar{y}_i = \frac{\sum_{j=1}^{n} y_{ij}}{n}$$

The overall mean is computed as

$$\bar{y} = \frac{\sum_{i=1}^{k} \sum_{j=1}^{n} y_{ij}}{n * k}$$

The variation due to the effects of the alternatives, sum-of-squares due to the alternatives (SSA) is computed as

$$SSA = n \sum_{i=1}^{k} (\bar{y}_i - \bar{y})^2$$

The variation due to random effects within an alternative is computed as the sum-of-squares of the errors (SSE) between the individual measurements and their respective alternative mean

$$SSE = \sum_{i=1}^{k} \sum_{j=1}^{n} (y_{ij} - \bar{y}_i)^2$$

When the above components are computed, a statistical test named F-test is performed to detect the statistically significant difference. F-test assess whether the expected values of a quantitative variable within several pre-defined groups differ from each other. The formula to compute the statistic F value is

$$F = \frac{SSA \cdot (n \cdot k - k)}{SSE \cdot (k - 1)}$$

with k is the number of alternatives. This F value follows the Fisher's F distribution with k – 1, n·k – k degrees of freedom. If this F value is larger than the pre-computed $F_{k-1;\, n \cdot k-k}$, we can conclude that there is actually statistically significant difference and vice versa.

Chapter 3

Scala Benchmarking Suite

Scala Benchmarking Suite – SBS (`scala.tools.sbs`) is a tool developed to do benchmarking on the Scala programming language. It is designed mainly being intended to detect various kinds of regressions on Scala standard library and Scala compiler, which are caused by changes to the source code, on the nightly build of each revision. Also, it can be used by Scala developers for their own purposes of improving Scala program quality by the mean of performance and optimization.

3.1. Features

This is the comprehensive list of features currently supported:

- Enable users to write their own benchmarks in the same way they write tests now - a file or a directory of files which corresponds to a Scala program. Users no longer need to worry about the implementation of benchmark iteration, warming up phase or statistical rigor, etc., which are all controlled by the built-in mechanisms of SBS
- Compile benchmarks using the compiler distribution comes along with the Scala standard library which is used to run the suite. Users can specify whether to (re) compile the benchmarks. The effect of the changes made to the compiler is reflected through the performance of the bytecodes compiled with it
- Run benchmarks selectively to obtain benchmarking results, such as performance numbers, number of times a method has been called, amount of memory consumed, etc. The metrics which regard performance are measured separately with the others to prevent them from being influenced. Users have the ability to specify more than one metrics at a time during a run. A specific metric can be recorded independently or together with the others

- Have benchmarks divided into groups and being able to run groups selectively and automatically, e.g. in nightly builds.
- Obtain benchmark results and keep benchmark results histories for comparison in the future. This is the main feature to keep track of the performance along the growth of the Scala standard library and compiler
- Using histories, automatically detect a failing benchmark, using statistical analysis. Performance histories are kept from previous SBS run on the accepted revisions of Scala.
- Be able to specify the used JVM when running a benchmark/benchmark group to detect the different performance of the same benchmark on various environment
- Benchmarks have default arguments, but when running them selectively, they can be passed in additional args (for instance, an array buffer is benchmarked with 10000 elements, but the user could run this specific benchmark with -Dsize=5000 if he so desires) - these arguments are defined on a per benchmark basis
- Comparative benchmarking - compare 2 approaches lively to point out the better one.
- Produce reports about the benchmark results, to send this through e-mail or be available through a web interface - report percentage losses and improvements in performance
- Various kinds of reports - first implement just text-only, but it is left extensible for various graphical representations
- Have verbose and debug output options
- Allow interfacing the suite through command line, ant and sbt[18]. Make it easy to continue developing with sbt project manager, integrate to Scala nightly build with ant and for the normal users to use as a jar package through command line

Usage: sbs [<options>] [<benchmark> <benchmark> ...]

<benchmark>: a path to a benchmark, typically a .scala file or a directory. All the per-benchmark <options> will be overridden by corresponding ones in .arg file with the same name with the snippet benchmark or values overridden from templates in the case of initializable benchmark. Following is the comprehensive list of all the possible arguments:

Benchmarking modes:

[18] https://github.com/harrah/xsbt/wiki

--steady-performance	SBS runs in steady state benchmarking mode
--startup-performance	SBS runs in start-up state benchmarking mode
--memory-usage	SBS measures benchmarks' memory usage in steady state
--profile	SBS profiles activities of benchmarks' runs (class loading, method invocations, etc)
--pinpoint	SBS runs pinpointing regression detection mode
--all	SBS runs all current supported benchmarking modes

Statistics metrics:

--least-confidence-level <value>	smallest acceptable confidence level (default: 90)
--precision-threshold <value>	% (default: 2%)
--timeout <value>	maximum time for each measurement (ms)
--noncompile	if set, SBS will not re-compile the benchmarks

Arguments necessary for performance benchmarking (see section 2.3.3 for the meaning of these arguments):

--measurement <value>	number of measurements (sample size) - default: 11
--multiplier <value>	number of benchmark run repetitions per measurement - default: 1
--sample <value>	number of pre-created samples used for statistically rigorous regression detection - default: 0
--re-measurement <value>	maximal number for re-measurements a metric in case the measurement result is not acceptable (too much noise for example) - default: 1

--warm-repeat <value>	maximal multiplier number of measuring repetitions for warming up. For example, if user specified --measurement 10 --warm-repeat 10, SBS at most repeat the benchmark running for 100 times at warming up phase. Default: 5

Arguments necessary for profiling:

--profile-classes <classes>	classes to be profiled - split by ; - default: <empty>
--profile-exclude <classes>	classes to be ignored - split by ; - default: <empty>
--profile-method <method>	the method to be profiled - default: <empty>
--profile-field <field>	the field to be profiled - default: <empty>
--profile-gc	if set, SBS will profile the running of the garbage collectors
--profile-boxing	if set, SBS will profile the number of boxing - unboxing
--profile-step	if set, SBS will profile the number of steps performed

Arguments necessary for pinpointing regression detection:

--pinpoint-class	name of the class contains the method to be regression detected - default: <empty>
--pinpoint-method	name of the method - default: <empty>
--pinpoint-bottleneck	if set, SBS will detect the bottleneck using performance regression pinpointing methodology (see section 4)
--pinpoint-previous <location>	the location of the previous build, should not be included in classpath - default: <empty>

--pinpoint-exclude	classes to be ignored - split by ; - default: <empty>

Specifying paths and additional values, ~ means SBS root:

--benchmarkdir	path from ~ to the working directory contains mostly log files and report files - default: .
--bindir	path from ~ to the directory contains binary files of the benchmarks - default: <empty>
--history	path to measurement result histories - default: .
--classpath	classpath for benchmarks running - default: <empty>
--scala-library	path to scala-library.jar - default: <empty>
--scala-compiler	path to scala-compiler.jar - default: <empty>
--javaopts	flags to java on all runs - default: JAVA_OPTS environment variable - currently unset
--scalacopts	flags to scalac on all tests runs - default: JAVA_OPTS environment variable - currently unset
--java-home	path to java

Options influencing output:

--show-log	if set, SBS will show the log message on the console
--verbose	verbose logging output
--debug	debugging logging output
--quiet	no console output

Other options:

--cleanup	delete all stale files and dirs before run

--noclean-log	do not delete any logfiles
--help	print usage message

3.1.1. Benchmark taxonomies

With respect to metric:

- Running time - easy to measure but unreliable, and has to be measured on the same platform to analyze history - no profiler should be used here
- Profiling a specific value - using a profiler with the JVM invocation
  - Classes that loaded
  - Number of times specific field(s) accessed/modified
  - Number of times specific method(s) invoked
  - Number of steps performed
  - Memory consumption
  - Number of GC cycles
  - Number of boxings/unboxings

A benchmark can possibly specify more than a single metric. In this case, the running time and profiled values are not measured during the same run. We separate runs for different profiled metrics.

With respect to measurement type

(This mainly concerns running performance benchmarks):

- Startup - perform measurements only once during JVM warm-up time and record them - what's measured may include both JIT compiled and interpreted code, along with the compilation time, class loading etc.
- Steady-state - run the benchmark code multiple times during a single JVM invocation until steady-state is detected (using the coefficient of variation), then do K iterations and measure the observed value - compute the mean of these K measurements
- Comparative - compares 2 programs (snippets, functions) and measures relative performance
- Performance regression pinpointing – compares 2 versions of the same program to detect performance regression and to point out the bottleneck (if any)

The taxonomies above may dictate how benchmarks are divided into logical groups, which makes them easier to select all at once. A group of benchmarks all are in a certain directory corresponding to their benchmarking mode.

With respect to how and when they're run:

- Individual - for individual use and parameter tweaking, these are not run on a nightly or regular basis, but on demand - the developer can play around with the parameters to test the code or changes he made
- Nightly - these are run on a regular basis to detect regressions on various revisions of Scala, based on statistical analysis the difference in performance with the performance histories kept from earlier builds

Benchmark directory structure

The working directory consists of the log files, report files and sub-directories:

- `bin` – holds binary (i.e. `.class`) files of benchmarks. All the bytecodes compiled from benchmark sources go here
- Mode directories – a directory exists for each benchmarking mode. Their names depend on the definition of the corresponding `BenchmarkMode`. Each mode directory holds the source files and the argument files of its benchmarks. Additionally, each benchmark has its own directory for generated histories (which are typically `.xml` files)

For example, a typical benchmark directory is shown below

```
\benchmark
        \steady
            \Benchmark_1
                    \history_1.xml
                    \history_2.xml
             \Benchmark_1.scala
        \startup
        \pinpoint
```

This design of directory structure is convenient to run selectively benchmarks depending on their mode of run all at once.

3.1.2. Measurement methodologies

There are several ways to run a benchmark:

- Run-once - such a benchmark is run once and the measured times are reported (e.g. printed on the screen) - typically the individual benchmarks are run this way
- Statistically analysis - the benchmark is run N times and the results are analyzed - a mean value, variance and a confidence interval for the value are computed. Confidence interval should be specified (perhaps only in strict steps, e.g. 90%, 95%, 99%)

3.1.3. Failure reporting methodologies

Clearly, this involves comparing the result of the benchmark against results from the previous runs – at least a history of previous performance measurements should be kept. Ways to discriminate failing benchmarks:

- Confidence interval - if the mean value fell out of the confidence interval of the last result, report an error
- Difference - compare the difference of the current and the previous result - if the confidence interval of the difference does not include zero, report an error
- Difference with up to M previous - same as difference, but compare against M previous results – using analysis of variance

3.1.4. Statistical analysis

In memory consumption, steady-state performance and pinpointing benchmarking mode, methodologies described in section 2.3 is applied. Typically:

- In memory consumption and steady-state performance mode, when no history available, no statistical analysis. When there is a single xml history, applied methodology described in section 2.3.4. Otherwise, methodology described is section 2.3.5 is used

- In pinpointing mode, methodology in section 2.3.4 is used in every comparison between current and previous performances (see section 4)

3.2. Use case

Here we describe the parameters for running SBS, the typical use case and an example of a typical benchmarking suite run. All of these considerations will reflect the architecture of the suite.

A benchmark suite run requires a set of parameters to be specified:

- A set of benchmarks to be run, specified as a list of benchmark groups and/or individual benchmarks
- The JVM options to use to run the benchmarks
- Optionally a set of JVM parameters, unless the per-benchmark defaults are to be used
- If a single benchmark was specified, its default arguments may be overridden (if the benchmark has any arguments specified - e.g. collection size or number of actors used)
- Measurement type, unless the default per-benchmark type is to be used (e.g. startup, steady-state)
- Measurement methodology, unless the default per-benchmark defined methodology is to be used (e.g. run-once, statistical analysis)
- For statistical analysis, various parameters such as confidence intervals or coefficients of variance, unless per-benchmark defaults are to be used
- Location where to run these - locally or on a remote machine
- Whether to compare results to previous results and the failure discrimination strategy
- Where is the history for previous runs kept - needed if results are to be compared against previous results
- Other options (e.g. verbosity level, influencing logging…)

A typical benchmark suite run is as follows:

1. Parse input arguments
2. Compile the sources of the benchmarks if necessary

3. Run all the benchmarks with the specified arguments
4. Load the previous results if necessary
5. Run comparisons to previous results if necessary
6. Post-process results if necessary (e.g. preparing reports, storing run results, etc.)

3.3. Design description

For convenience purpose, all the package and type names mentioned in the following sub-sections are originally prefixed by `scala.tools.sbs`.

3.3.1. Main architecture

Figure 3.1 depicts the main activities of a SBS run session, which are, with respect to the order:

- Parsing user arguments using `ArgumentParser`. An instance of the class `Config` is created representing the environment and user's requirements of the run. Logging activities also starts at this point, their instances and I/O format depends on user's requirements
- Reading information about the benchmarks to be run from user arguments or from per-benchmark specific argument files. Users are able to specify one or more benchmark at a time with command line - this ability is convenient for running selective benchmarks. They also can specify several directories contain lots of benchmarks inside in the case SBS is used in nightly builds, in grouping benchmarks or the number of benchmarks is just too large to fit in.

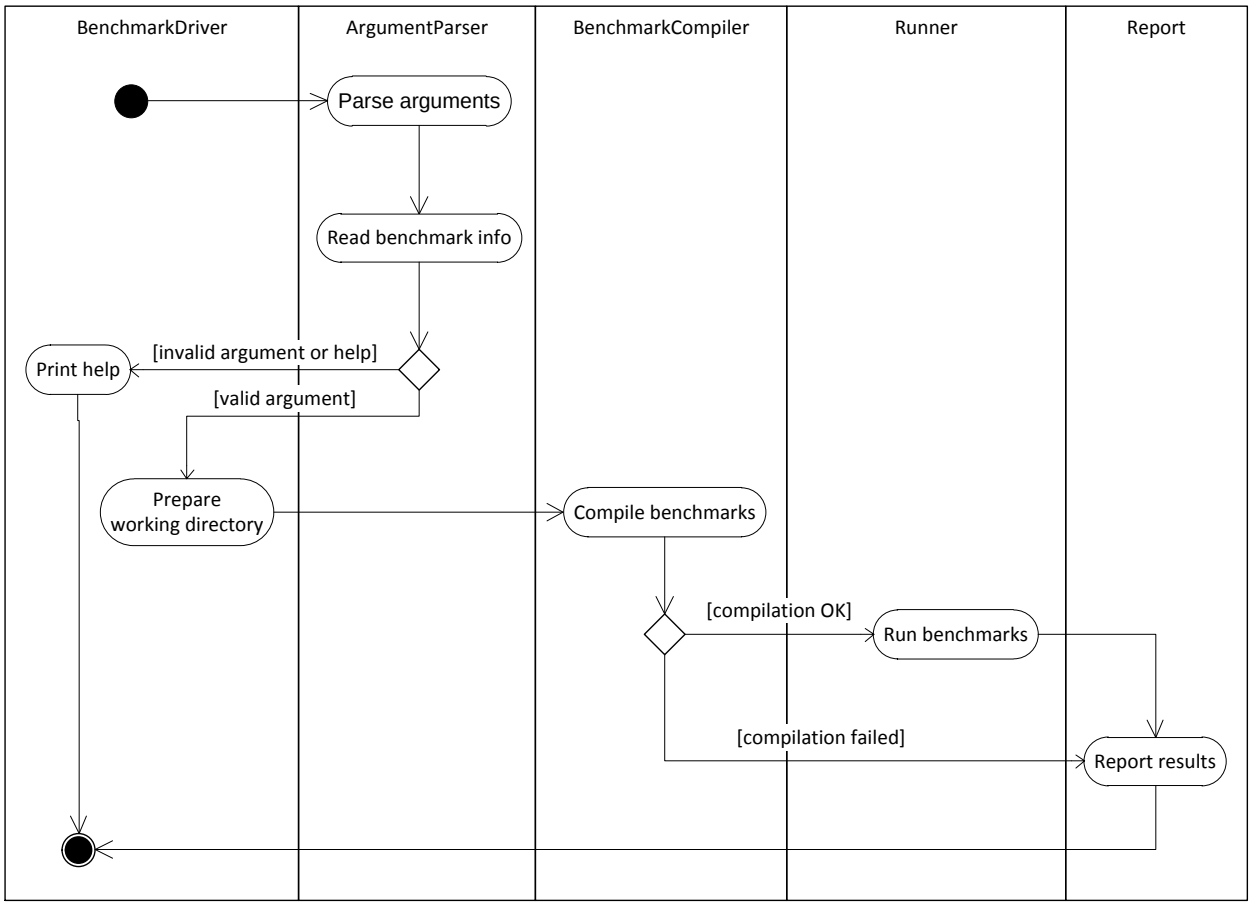

Figure 3.1 – Main activity diagram

- Preparing the working directory structure, including creating places for each mode to store their histories and logging, cleaning unwanted files, etc.
- Compiling all the benchmarks needed using `BenchmarkCompiler`. If a benchmark has already been compiled before and the change (if any) does not concern it implementation, it is not necessary to re-compile the benchmark (i.e. changes were made to the standard library, and are reflected through the re-compilation of the standard library, which has taken its place before the SBS run session).
- For each selected `BenchmarkMode`, do benchmarking on all the list of the benchmarks specified to run in the mode using the corresponding instance of `Runner`
- Finally processing (typically is reporting) the generated benchmarking results

Object `scala.tools.sbs.BenchmarkDriver`

`BenchmarkDriver` is the central back bone of SBS. Its `main` method is the entry point and controls the work flow of every SBS run session. `BenchmarkDriver` receives user arguments and instantiates the necessary objects.

Class `scala.tools.sbs.ArgumentParser`

`ArgumentParser` receives the `Array` of user's arguments to create the `Config`, the log file and read the information of the to-be-run benchmarks.

A typical benchmark is stored in local disk in the form of a Scala source file or a directory of Scala source files. It may come along with an argument file – a file with the same name and the extension `.arg`. This file contains the additional per-benchmark information such as the number of iterations, whether to recompile, the time out, etc. All the benchmarks' information read is represented as a set of `benchmark.BenchmarkInfo`.

`ArgumentParser` runs only once for each SBS run session and has no state. So, `ArgumentParser` is implemented to be an object with a small number of static methods.

Class `scala.tools.sbs.Config`

An instance of `Config` reflects the benchmarking "environment". It consists of the user arguments has been parsed and the constant values, such as precision threshold etc., used by most of all classes in SBS.

Trait `scala.tools.sbs.Runner`

`Runner` is one of the most important `traits` in SBS. Each of its concrete implements and their supporting types should represent the necessary activities to produce the benchmarking results. An important part of Runner is shown in Listing 3.1.

With the purpose to have SBS easily extensible, `Runner` is designed to represent a vast various ways and metrics of measurement. Depends on the running mode, a suitable `Runner` is instantiated. Each implementation of `Runner` has its own sub-type of `benchmark.Benchmark` which it can process (for example, a `profiling.Profiler` (see section 3.3.3) can only run a `profiling.ProfilingBenchmark`). At the first step of running a benchmark, a `Runner` checks whether the benchmark is suitable (see Listing 3.1). That can be fulfilled using the method `check()`, which is inherited by `Runner` from the `trait common.RuntimeTypeChecker`. `check()` tests whether the type

represented by the `Runner`'s field `upperBound` is a super-type of the benchmark's type by using `scala.reflect.Manifest`.

```
trait Runner extends Configured with RuntimeTypeChecker {

  def benchmarkFactory: BenchmarkFactory

  def run(benchmark: Benchmark): BenchmarkResult =
    if (check(benchmark.getClass)) {
      val result = doBenchmarking(benchmark)
      result.toReport foreach log.info
      result
    }
    else {
      throw new
        MismatchBenchmarkImplementationException(
          benchmark,
          this)
    }

    ...

}
```

Listing 3.1 – Trait `scala.tools.sbs.Runner` (simplified)

A little thing about Scala – Java *Generics*[19] is necessary to be described here to for one to be able to understand the role of `scala.reflect.Manifest`. *Generics* is a facility of generic programming that allows a type or method to operate on objects of various types while providing compile-time type safety (in our case these types are the sub-types of `benchmark.Benchmark`s). Java's, therefore Scala's, approach to implement Generics is *type erasure*[20] that removes the *type parameter* information at compile time. So that, every objects at runtime have the type of `java.lang.Object`. That makes the operation `isinstanceof T` (`T` is a *type parameter*) always yields `true` and thereby meaningless. With Scala, one can work around the *type-polymorphism* problem using the `scala.reflect.Manifest`. `Manifest`s are descriptors for types which can be used in runtime to test type-relating constraints and are our solution applied in SBS.

In addition, a runner must define its own factory to generate the concrete benchmarks. The factory is hold in the field `benchmarkFactory` at SBS runtime. For instance, `performance.Measurer` has its fields defined as follow:

[19] http://en.wikipedia.org/wiki/Generics_in_Java
[20] http://en.wikipedia.org/wiki/Type_erasure

```
protected val upperBound = manifest[PerformanceBenchmark]
val benchmarkFactory =
  new PerformanceBenchmarkFactory(log, config)
```

The harnesses

Harness is the common term for every `object` named suffixed by "`Harness`" in SBS.

To satisfy the constraints that require benchmarks run in a clean JVM, a harness is a controller for running benchmarks in a separated JVM and is a sub-type of the trait `common.ObjectHarness`. A harness in general is the main Scala class in its JVM and has its own `main()` function as the entry point.

A harness typically does the following steps:

- Recreating the `Config` and the log
- Loading the benchmark classes and iterating its runs using *reflection*
- Reporting measurement result to the main JVM

A new JVM is needed for a harness to run to satisfy the constraints of benchmarking environment. It is launched using `common.JVMInvoker` (see section 3.3.6). The hierarchy of harnesses is illustrated in Figure 3.2.

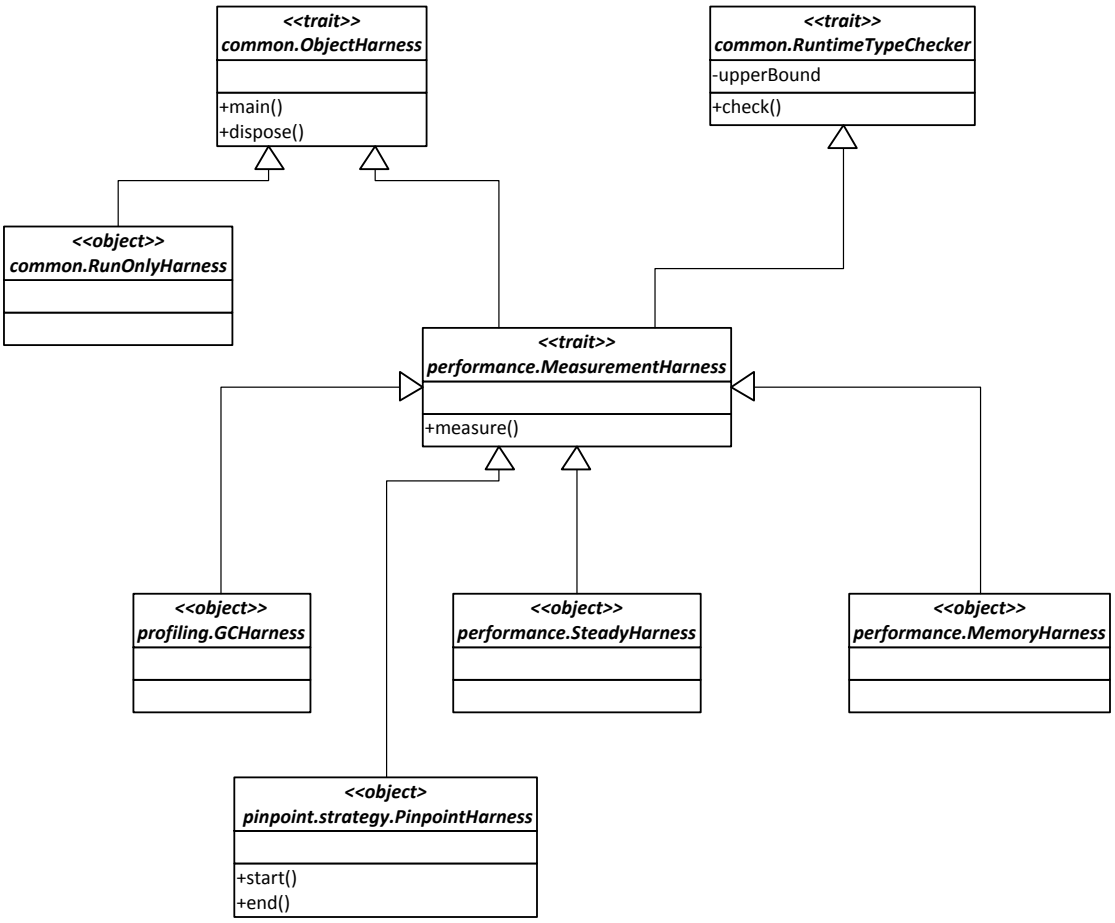

Figure 3.2 – The harnesses' class diagram

Adding new benchmarking mode into SBS

SBS is designed to be easily extendable in order to meet all the requirements may appear in the future. To extend the abilities of SBS to be able to do some additional kind of benchmarking, follow these steps:

- Create an `object` which extends the `trait BenchmarkMode`. It represents the newly added mode and lets `BenchmarkDriver` be able to switch onto the new benchmarking mode in a SBS run session
- Create a command line option for users to be able to select the new mode. The option must be declared in the trait `BenchmarkSpec` so that a `var` named `_modes` appends the new mode `object` whenever the option is specified. All other options' definitions and metrics that the new mode needs also go here
- Create a new `package` in package `scala.tools.sbs` for all the implementations of the new benchmarking mode (not necessary, this is just for convenience and neatly looking code purpose)

- Define a new kind of benchmark provides all the necessary information to the new benchmarking mode and a new result type. The implement of the benchmark kind has to be a sub-type of `benchmark.Benchmark` for the `Runner` to accept it, the result type is a sub-type of `BenchmarkResult`
- Create a new factory class for creating instances of the new benchmark type from user arguments. This factory must be a sub-type of `benchmark.BenchmarkFactory` (see section 3.4.5)
- Create a sub-type of `Runner` which is used to run in the new mode. The field `upperBound` must hold the `manifest` of the new benchmark type and the field `benchmarkFactory` holds an instance of the new benchmark factory

When all of the above are complete, the new benchmarking mode is ready to run. Following is an example. Supposed that we want to add a new benchmarking mode called `NewMode` to SBS. The steps to have it ready in SBS are described below:

- Define the mode object

```
object NewMode extends BenchmarkMode {
  val location = "newmode"
  override val toString = "NewMode"
  val description = "for example purpose"
}
```

- Insert an option (as the following line of code) to `BenchmarkSpec`

```
"newmode" / "run in the new mode" --> (
  _modes ::= NewMode)
```

- Create the new package `scala.tools.sbs.newmode`
- Define the new type of benchmarks, which is the following class:

```
package scala.tools.sbs
package newmode

class NewBenchmark extends Benchmark {

  def name = "NewBenchmark"
  def arguments = List[String]()
  def classpathURLs = List[URL]()
  def sampleNumber = 0
  def createLog(mode: BenchmarkMode): Log = null
  def timeout = 10000
  def init() = ()
  def run() = ()
```

```
  def reset() = ()
  def context: ClassLoader =
    Thread.currentThread().getContextClassLoader()
  def toXML: scala.xml.Elem = <newbench />

}
```

- Define class `NewBenchmarkFactory` which is used to create `Benchmark` instances which actually are `NewBenchmark`s for them to be able to run by the new runner

```
package scala.tools.sbs
package newmode

class NewBenchmarkFactory(val log: Log,
                          val config: Config)
  extends Configured
  with BenchmarkFactory {

  /* In real life, createFrom() should use method
   * load() inherits from BenchmarkFactory to read
   * .arg files or instantiate an
   * InitializableBenchmark.
   * See the pre-created BenchmarkFactory-s for
   * more details.
   */
  def createFrom(info: BenchmarkInfo): Benchmark =
    new NewBenchmark

}
```

- Modify method `apply()` of object `BenchmarkFactory` to have it create `NewBenchmarkFactory` in the case `--newmode` is selected

```
object BenchmarkFactory {

  def apply(log: Log,
            config: Config,
            mode: BenchmarkMode): BenchmarkFactory =
    mode match {
      case DummyMode =>
        new DummyBenchmarkFactory(log, config)
      case Profiling =>
        new ProfilingBenchmarkFactory(log, config)
      case Pinpointing =>
        new PinpointBenchmarkFactory(log, config)

      //insert this
      case NewMode =>
        new NewBenchmarkFactory(log, config)
```

```
      case _ =>
        new PerformanceBenchmarkFactory(log, config)
  }

}
```

- Define new runner which has field `upperBound` holds the manifest of `NewBenchmark` and field `benchmarkFactory` holds an instance of `NewBenchmarkFactory`. These fields will be later used to create (load) the benchmarks

```
package scala.tools.sbs
package newmode

class NewRunner extends Runner {

  protected val upperBound = manifest[NewBenchmark]

  val benchmarkFactory =
    new NewBenchmarkFactory(log, config)

  protected def doBenchmarking(benchmark: Benchmark):
    BenchmarkResult = {
    // do things here
  }

}
```

- Modify the factory object for the runners: Insert a case of `NewMode` to method `apply()`

```
object RunnerFactory {

  def apply(config: Config,
            log: Log,
            mode: BenchmarkMode): Runner =
    mode match {
      case Profiling =>
        ProfilerFactory(config, log)
      case Pinpointing =>
        ScrutinizerFactory(config, log)
      case StartUpState | SteadyState | MemoryUsage =>
        MeasurerFactory(config,
                        log,
                        mode,
                        MeasurementHarnessFactory)
      case Instrumenting =>
        InstrumenterFactory(config, log)
```

```
      // Insert this line
      case NewMode => new NewRunner

      case _ =>
        throw new NotSupportedBenchmarkMode(mode)
  }

}
```

The new mode is now completely added to SBS. Users can have SBS run the new benchmarking mode with the command option `--newmode`.

3.3.2. Package scala.tools.sbs.performance

This package implements the most important and heavily-used benchmarking modes: benchmarking steady-state performance. Besides, it comes along with the ability to do benchmarking start-up performance and memory consumption in steady-state.

The central unit of package `performance` is the trait named `Measurer` which extends `Runner`. `Measurer` has two concrete classes, `StartupHarness` and `SubJVMMeasurer`.

- `StartupHarness` measures the running time in start-up state of the benchmark, not necessary for it to run in a new JVM.
- `SubJVMMeasurer` uses the corresponding harness to measure the steady-state performance or the memory consumption

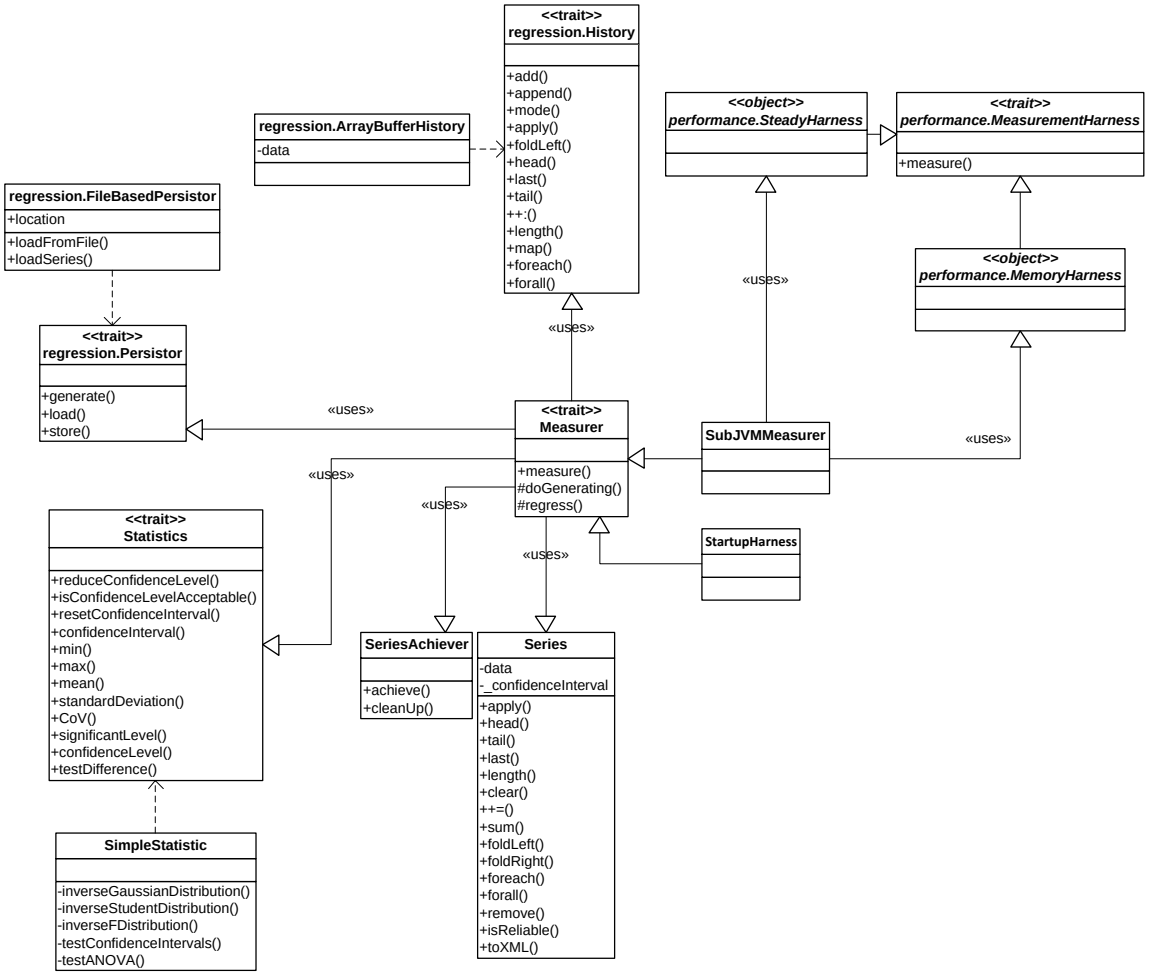

Figure 3.3 – Package `performance`'s class diagram

Figure 3.3 illustrates the static structure of the most important elements of package `performance`, includes:

- Class `Series` represents a series of performance measurements.
- Class `SeriesAchiever` is the controller of benchmarking iterations. `SeriesAchiever` guarantees that the benchmark has reached steady-state after warming up phase and redoes the whole measurement if the final series is not reliable
- Package `regression` – contains classes support *statistically rigorous performance regression detection*:
    - Trait `Statistics` provides the operations concern statistical metrics. It computes CoV of a series as well as its mean and sample standard deviation. The most important functionality of `Statistic` is checking

the statistically significant difference among the current series and histories using *statistically rigorous performance detection* methodology

- Trait `History` consists of a list of performance Series from previous benchmarking runs
- Trait `Persistor` loads and stores Series into the storage. Currently, `Persistor` is implemented to work with measurement histories in the form of `.xml` files

A typical run in steady-state benchmarking mode is described in Figure 3.4

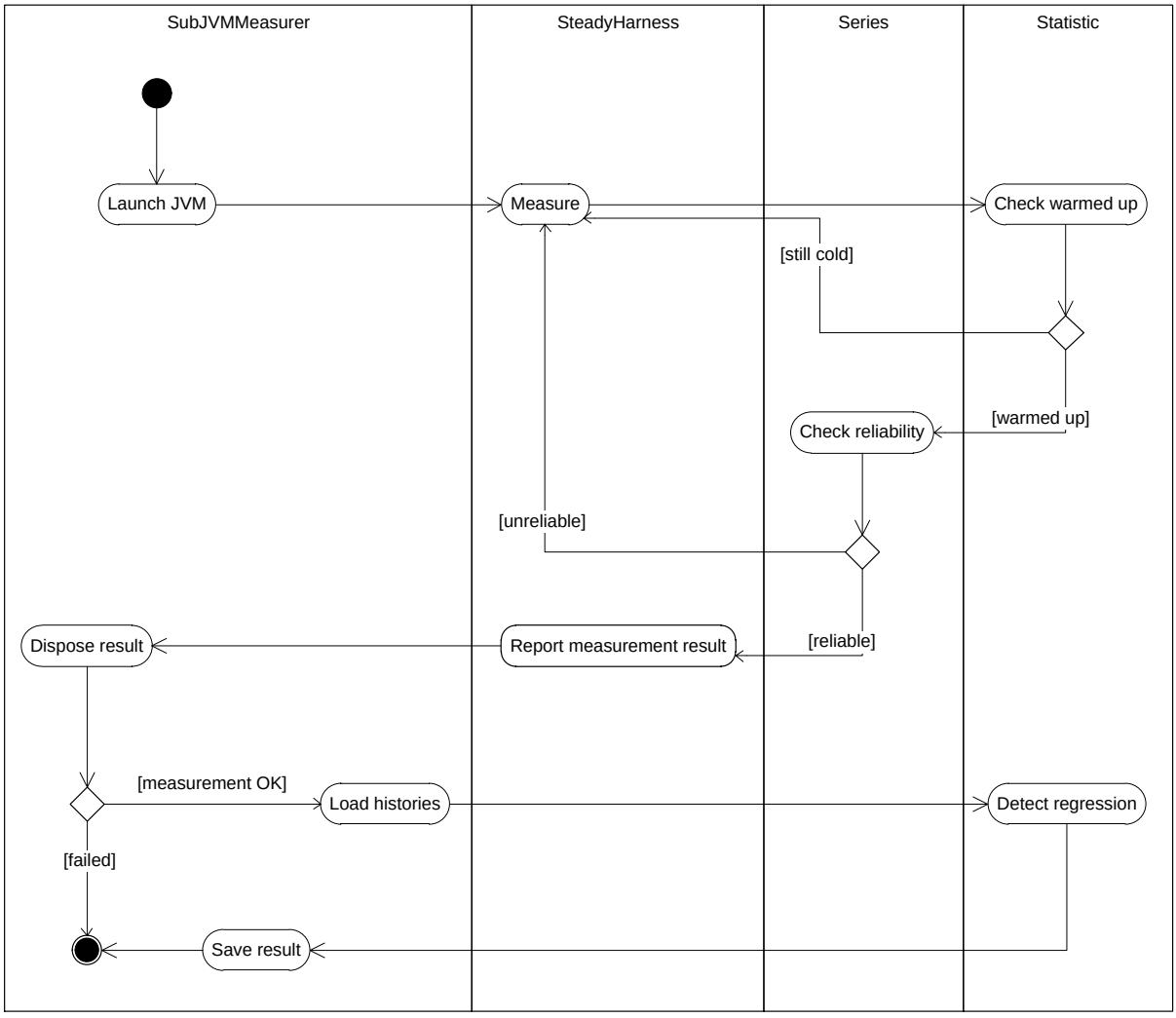

Figure 3.4 – Steady-state performance benchmarking activity diagram

3.3.3. Package scala.tools.sbs.profiling

Package `scala.tools.sbs.profiling` implements a benchmarking mode under the form of a profiler using the high level API Java Debug Interface (JDI) which is a

part of Java Platform Debugger Architecture (JPDA). The profiler is used to record the activities of a benchmark running session and profile typical metrics specified by user, includes:

- Classes that are loaded together with its methods that ran and/or fields accessing and modifying
- Number of boxings/unboxings
- Number of steps performed
- Memory activities that consists of types of garbage collectors have been run, their number of cycles and time spend. Also, memory usage on heap and non-heap memory, their `init`, `used`, `committed` and maximal memory fragments

Interaction to this package i.e. the profiler is done through a `trait`, sub-type of `trait Runner`, named `Profiler`, which has its implementation to be `JDIProfiler`. Figure 3.5 depicts the relations among the types in package `profiling`.

The architecture of the profiling process follows the guide from Sun to write a debugger based on the JDI API. When starting running, a `JDIProfiler` launches a new JVM running the user program and creates a mirror of that JVM for management representing as an instance of class `com.sun.jdi.VirtualMachine`. The mirror of the JVM is then passed to be processed by a `JDIEventHandler`. All the requests for event generating are registered. Finally, a `JDIThreadTrace` is created for each of the JVM's threads recording the events generated.

The result created using `JDIProfiler` is contained in an instance of class `Profile`. It is then passed to `MemoryProfiler` to continue recording the activities of memory usage. The `ProfilingBenchmark` is run in a new JVM under the control of `GCHarness` to make sure the values recorded are achieved satisfied all the constraints of benchmarking environment.

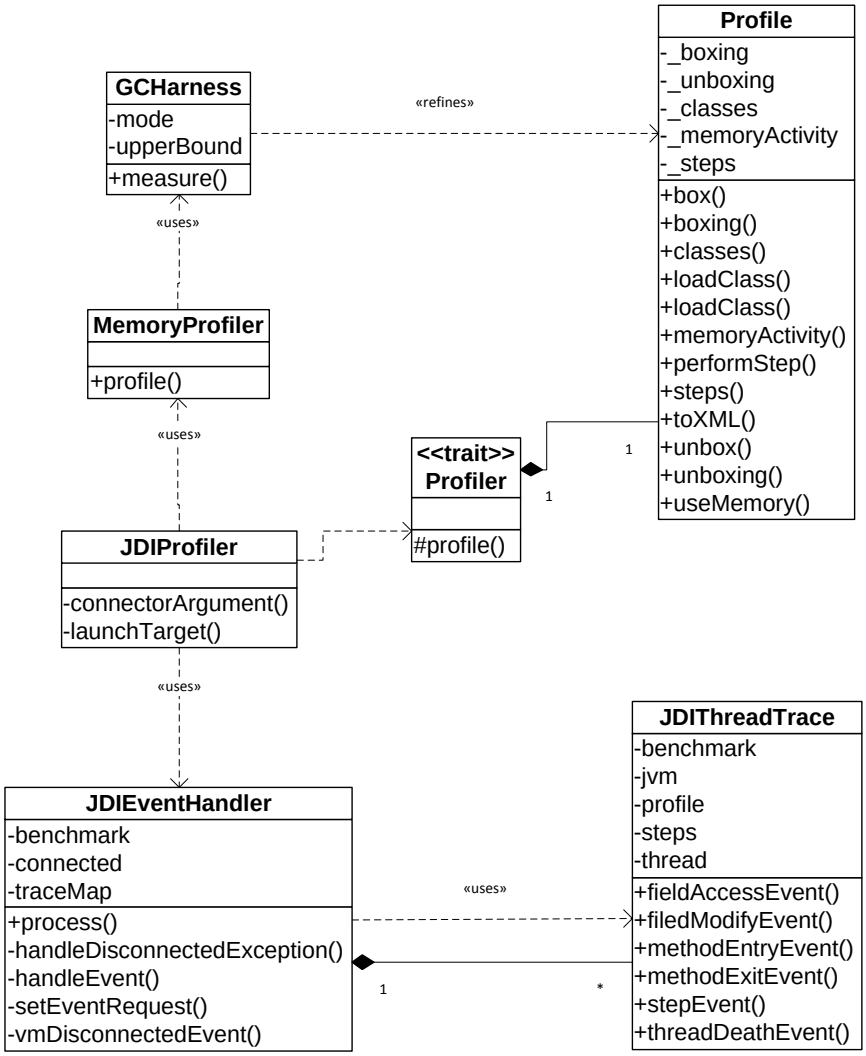

Figure 3.5 – Package profiling's class diagram

3.3.4. Package scala.tools.sbs.pinpoint

This package implements the bottleneck finding methodology named *performance regression pinpointing*. Package `scala.tools.sbs.pinpoint` heavily depends on the package `scala.tools.sbs.performance` since it also uses the same process of measurement and regression detection. Many of its important classes extend classes from package `scala.tools.sbs.performance` including the harnesses and benchmarks.

Package `scala.tools.sbs.pinpoint` will be described in details in section 4.8.

3.3.5. Package scala.tools.sbs.benchmark

This package contains all of the class definitions to represent the information about user benchmarks.

`BenchmarkInfo` stores the basic information about a benchmark: its name, where to find the source code, whether to compile…In addition, it composes the corresponding concrete benchmark object using a concrete factory.

The snippet benchmarks and initializable benchmarks are the two supported kind of a benchmark implementation. That may be

- A snippet benchmark – is a standalone Scala program that can run on Scala independently from SBS. User defined it with a `main` method in an `object`
- An initializable benchmark – is mainly a class which implements a special trait from SBS – a sub-type of `BenchmarkTemplate` which is provided to the user and has the interface depends of the benchmarking mode. The initializable benchmark is used when a lot of data has to be generated before starting the benchmark. In practice, to be precise in performance measurements, we do not want the cost of data preparation to influence the overall running time. In that case, a initializable benchmark is used and its method named `init()` runs all the initializations (things like loading data from files, creating large arrays or instantiating data-heavy classes, etc.)

The benchmark templates are the interfaces provided to the user to implement their initializable benchmarks. Those are sub-type of `BenchmarkTemplate` and are defined depends on the benchmarking mode they are intended to run in.

A BenchmarkFactory using reflection to load the classes corresponds to the benchmark. It defined the loading process from `classpath` but leaves the creating of benchmark instances to the concrete sub-type.

The current benchmark hierarchy of package `benchmark` is depicted in Figure 3.6:

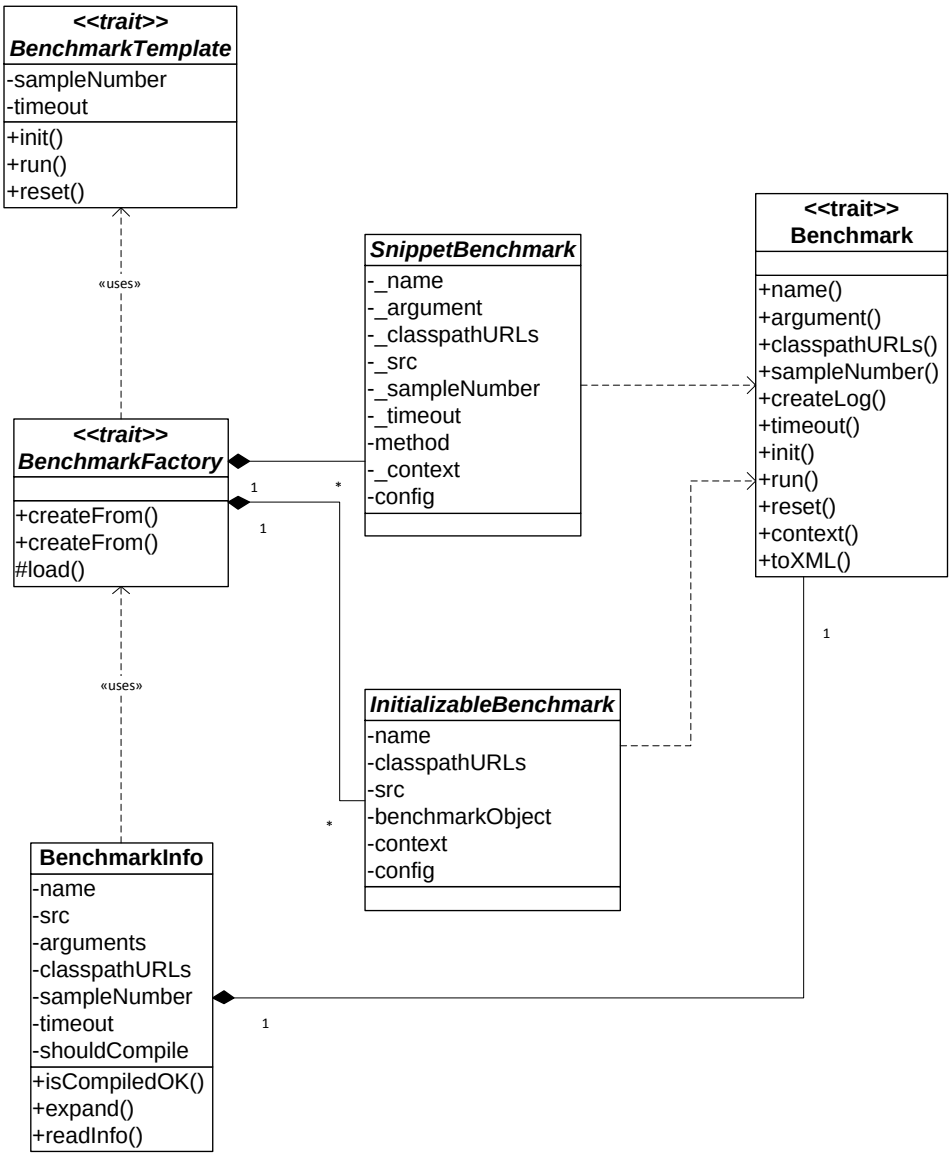

Figure 3.6 – Package `benchmark`'s class diagram

3.3.6. Other supporting packages

Package `scala.tools.sbs.common`

Consists of several very important `trait`s, some of them define a number of the core operations in SBS

- `Reflector` – has a simple implement to be `SimpleReflector`. A `Reflector` provides the ability to dynamically load the definition of classes, create class instances and find the location where the given class locates in `classpath` using *reflection*[21]. `Reflector` is used in order to create

[21] http://en.wikipedia.org/wiki/Reflection_(computer_programming)

`benchmark.Benchmark` instances or support instrumentation phase of *performance regression pinpointing* (see section 4.5)

- `ObjectHarness` – trait `ObjectHarness` is the super-type of all the harnesses in SBS (see section 3.3.1 – small topic The Harnesses). An `object` sub-type of `ObjectHarness` reports the result from a measurement by printing the `xml` element representing the result to its standard output, which is transferred back to the main JVM as a data stream[22]
- `RuntimeTypeChecker` – uses `scala.reflect.Manifest` to test for type-related constraints at runtime. It is mixed-in the implements of `Runner` and the harnesses
- `Backuper` – backups and restores the unwanted files form their original locations. `Backuper` is mostly used in the instrumentation phase of *performance regression pinpointing* (see section 4.5)
- `BenchmarkCompiler` – compiles the benchmark source files into Java byte codes. Its single implement, `BenchmarkGlobal`, uses the standard built-int Scala compiler which is `scala.tools.nsc.Global` to do the compilation and reporting on benchmarks that cannot compile
- `JVMInvoker` – has an implement to run Scala in new JVM, named `ScalaInvoker`. The most important operation of `JVMInvoker` is invoking a new JVM for some purpose and then using the two argument functions `String => E`, `String => R` (`E` and `R` are type parameters) to process each line of the standard output and standard error, producing the two `ArrayBuffer[R]` and `ArrayBuffer[E]` as the return values

Package `scala.tools.sbs.io`

`io` package defines the `traits` `Log` and `Report` which currently simply write plain texts to `.txt` files. A special sub-type of `Log` is the `object UI`. It prints messages directly to the `console` to interact with users.

Package `scala.tools.sbs.util`

Package `util` consists of a few utilities

- `object Constant` holds the platform-specific pre-computed values such as `path.separator`, `file.separator`, etc.

[22] http://www.scala-lang.org/api/current/scala/sys/process/ProcessIO.html

- `object FileUtil` implements all the file-system-related operations including file reading and writing, preparing working directory structure etc.

3.4. Experiment

In the next sub-sections, we will evaluate the performance and try detecting regression on the benchmark named ArrayCopy (described in section 3.4.1) using statistically rigorous data analysis. For doing so, we consider an experiment in which we compare the statistically significant difference in the performances measured from different iterations of the benchmark. Section 3.4.1 discusses the experimental setup: the benchmark, the configurations of the virtual machine and the hardware platform. In section 3.4.2, we illustrate the process evaluating the performance of the benchmark which includes the warming up phase. Finally, sub-section 3.4.3 uses the figures included to depict the comparing results.

3.4.1. Experimental setup

This section describes the implementation and running environment for the benchmark ArrayCopy.

The benchmark ArrayCopy

The benchmark used in this experiment is called ArrayCopy. The main activity of the benchmark is to clone a set of arrays. The arrays to be cloned consist of arrays of types: `Object`, `Boolean`, `Byte`, `Char`, `Double`, `Float`, `Int`, `Long` and `Short`, each has size of 48000.

```
val size = 48000

val objectArray  = new Array[Object](size)
val booleanArray = new Array[Boolean](size)
val byteArray    = new Array[Byte](size)
val charArray    = new Array[Char](size)
val doubleArray  = new Array[Double](size)
val floatArray   = new Array[Float](size)
val intArray     = new Array[Int](size)
val longArray    = new Array[Long](size)
val shortArray   = new Array[Short](size)

val lst = List(
  objectArray,
  booleanArray,
  byteArray,
```

```
  charArray,
  doubleArray,
  floatArray,
  intArray,
  longArray,
  shortArray
)
```

Listing 3.2 – Benchmark ArrayCopy – arrays to be cloned

When run, as shown in Listing 3.2, benchmark ArrayCopy clones every element of the `scala.List` of `scala.Array` named `lst`. It generates the new `scala.List` by using the higher-order function `map`. `map` maps each element of `lst`, which is an array, into the new element of `newLst` by applying the first-order function passed to it, in this case, the function `clone` of the `scala.Array` elements.

```
val newLst = lst map (_.clone)
```

Listing 3.3 – Benchmark ArrayCopy – operations when run

The precision threshold is set to 2% by default (user can also modify the threshold with SBS option --precision-threshold, see section 3.1 for the comprehensive list of user arguments).

The number of measurements (the number of sample to apply statistical analysis) retained to be kept is set to 13. This means that, for a performance measuring to be considered success, it achieves a series of running time which has the length of 13 and its confidence interval is less than 2% of its mean (we have described confidence interval statistics in section 2.3.1).

The iteration of the benchmark is initially set to 41. This means that a running time achieved is the time needed for ArrayCopy to run 41 times. This makes sure the running time measured is large enough (in general cases, up to several hundreds of milliseconds) thereby meaningful.

Java Virtual Machine

We use the JVM distributed by Sun, Java SE 6 build 29: Java version 1.6.0_29, Java SE Runtime Environment (build 1.6.0_29-b11), Java HotSpot Server VM (build 20.4-b02, mixed mode). We consider running the Server VM of Java HotSpot Virtual Machine with the default garbage collection strategy: Copy and MarkSweepCompact.

Hardware platform

We considered a single hardware platform in our performance evaluation experiment: Intel Core 2 Duo CPU T5800 2.00GHz (2 CPUs) has 2048MB of main memory. The machine runs the Windows 7 ThinPC operating system. In all the performance evaluations, we consider the machine either unloaded or idle.

3.4.2. Warming up

As explained in section 2.2.1 and 2.3.3, for a measurement to be complete in steady-state, the benchmark will have to run through its warming up phase which makes the bytecodes fully optimized and perform the highest performance for the measurement.

Figure 3.7 illustrates the first measuring process of benchmark ArrayCopy to achieve a series of running time performance and store it as a history for detecting regression in the future.

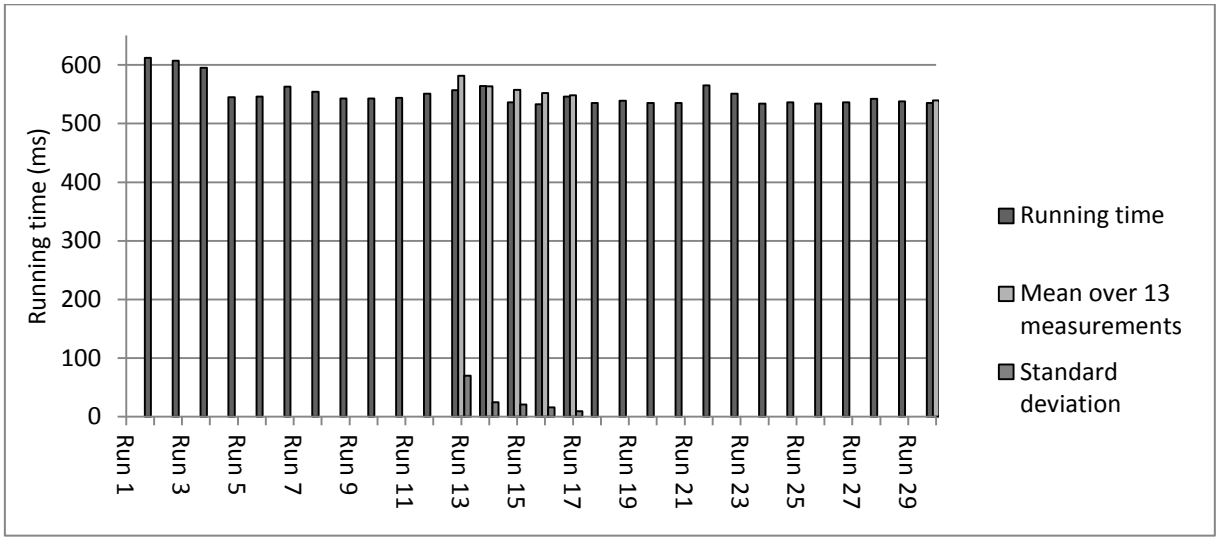

Figure 3.7 – All measurements, including warming up phase,
of a measuring process on the benchmark ArrayCopy

The graph shows that SBS run through 30 measurements in total to retain the final series of 13 running time samples. And therefore, SBS used the first 17 measurements to warm up the benchmark. To be more precise in material data, we can explain the detection of reaching steady-state as follows:

- First, measure the running time to get a series of 13 samples. The series has its length exactly is equal to the length we want to retain. At the time, the mean and sample standard deviation of the current series is computed, which are

respectively 581.46 and 69.92. That makes the coefficient of variation 12%, much larger than we expect. The conclusion here is that the benchmark has not reached its steady-state

- The measuring is continued. For each new measurement, the oldest sample in the series is removed and the new one is appended. The measuring and computing are repeated until the CoV of the current series becomes less than 2%. The computed metrics for the next runs, respectively are the mean, the sample standard deviation and the CoV, are listed below:

run	mean	standard deviation	CoV
14	563.3846154	24.82451229	4.4%
15	557.5384615	21.08955265	3.7%
16	551.8461538	15.9991987	2.9%
17	548.0769231	9.393805924	1.7%

- At run 17, the CoV has become less the 2%, the steady-state is detected. The final series retained is composed by the next 13 samples measured from run 18 to run 30

These measurement results are kept and used as the history to detect performance regression in the future.

3.4.3. Statistically significant difference detection

The steps performed in the experiment as follow:

- A series was initially stored as the first history. That was the series we had achieved by the measuring described in section 3.4.2
- The benchmark iteration was changed from 41 to 45 for the benchmark to have a worse performance. That caused the performance regression that we intend to detect using confidence interval test. This is considered as a failing benchmark and the measurement results were not stored as history. Illustrated by Figure 3.8
- The benchmark iteration was changed from 45 back to 41. That caused the benchmark to perform its original performance so that no regression can be detected by confidence interval test. The measurement results are stored as another history. Illustrated by Figure 3.9
- The benchmark iteration was changed to 45 once again. The benchmark then had a worse performance. Since we already had 2 series as persisting histories,

the ANOVA test was performed instead of the confidence interval test. Illustrated by Figure 3.10

- The benchmark iteration was changed back to 41 to illustrate the case ANOVA test detects no performance regression. Illustrated by Figure 3.11

All the series are achieved in steady-state (all the measurements taken in steady-state, the benchmark had run through warming up phase like ones in section 3.4.2).

Performance regression detected using confidence interval

After changing the benchmark iteration to 45, the performance measuring was performed and the result series is depicted in Figure 3.8 in its comparison with the history.

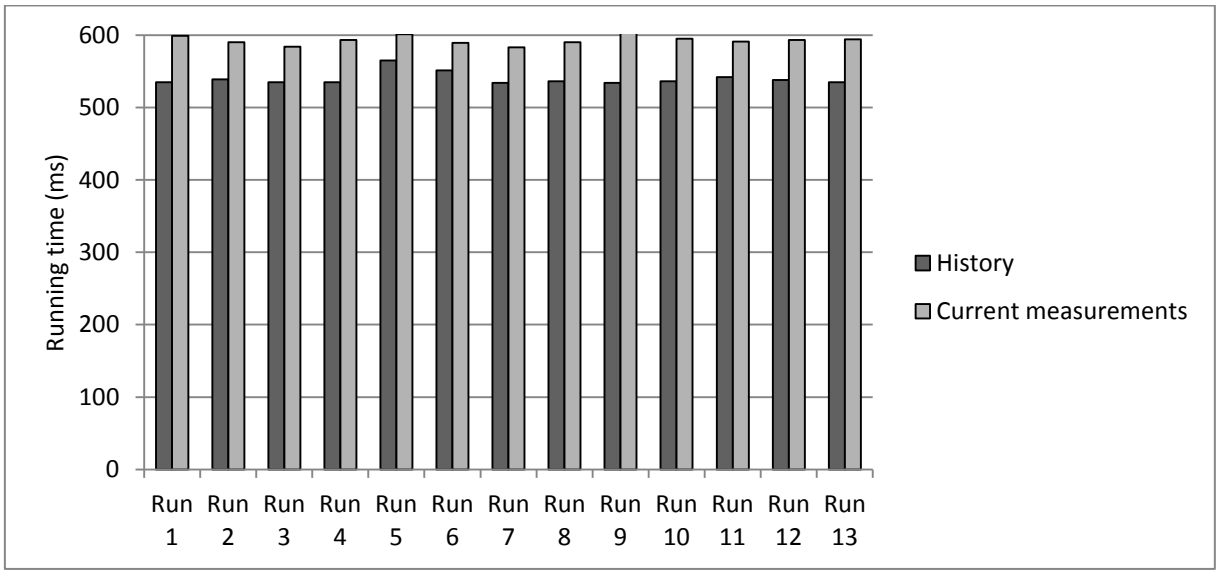

Figure 3.8 – Performance regression detection using confidence interval – regression detected

It can be seen with naked eye that the new performance is almost 600 milliseconds and significantly worse than the original one which even cannot reach 550 milliseconds. The confidence interval computed is [-58.21; -47.95] at the confidence level reduced to 90%. Since the confidence interval did not contain zero, there is performance regression detected.

No performance regression detected using confidence interval

The benchmark iteration was changed back to 41, causing the benchmark performs its original performance as depicted in Figure 3.9.

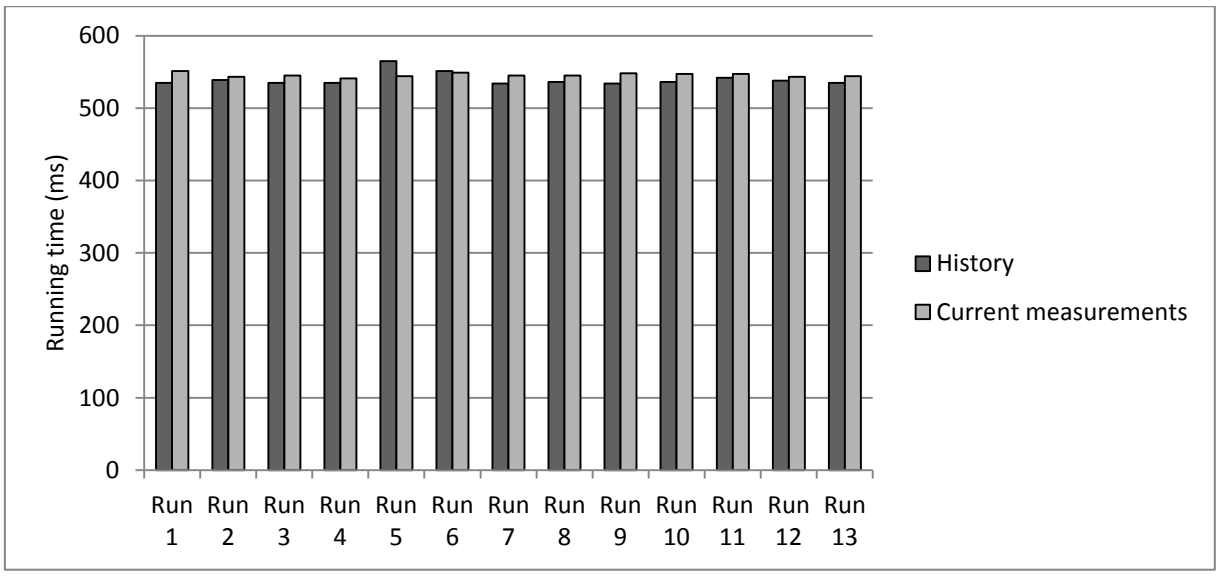

Figure 3.9 – Performance regression detection using confidence interval – no regression detected

The new performance is now about 550 milliseconds just like the samples stored in history. The confidence interval computed is [13.64; 1.79] at the confidence level 99%. The confidence interval did contain zero, so there is no performance regression detected and the measurement results are stored as another history.

Performance regression detected using ANOVA

After once again changing the benchmark iteration to 45, the performance results are measured and shown in Figure 3.10 in its comparison with the histories.

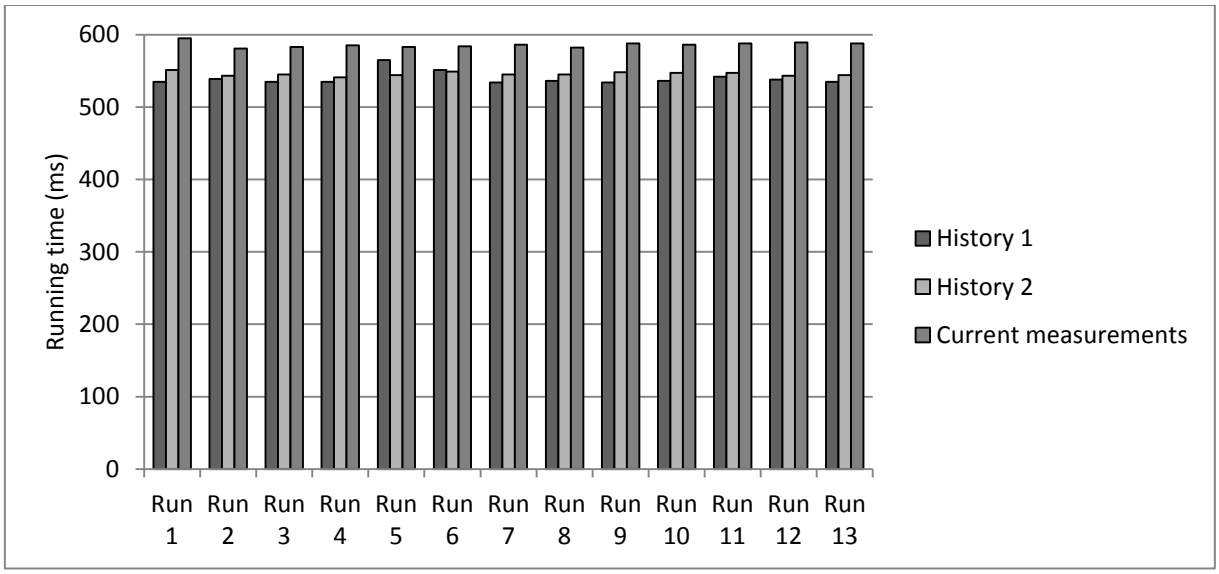

Figure 3.10 – Performance regression detection using ANOVA – regression detected

The running time of the benchmark is now high again. The statistics computed as follow:

- F-value computed 137.36
- $F_{12;\ 36}$ according to F-distribution 5.25

Since F-value is much larger than $F_{12;\ 36}$ from the distribution, there is performance regression detected.

No performance regression detected using ANOVA

Finally, Figure 3.11 depicts the series achieved after changing benchmark iteration back to 41 along with the series from histories.

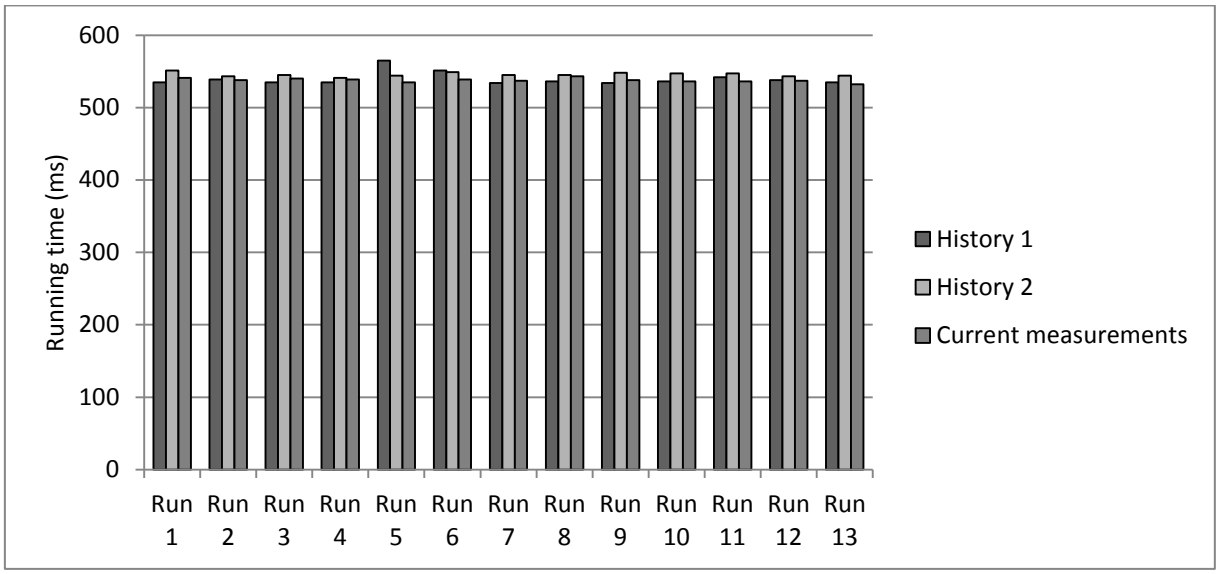

Figure 3.11 – Performance regression detection using ANOVA – no regression detected

The benchmark performance changed back to its original value which is about 550 milliseconds. The statistics computed at this point:

- F-value computed 3.99
- $F_{12;\ 36}$ according to F-distribution 5.25

In this case, F-value is smaller than $F_{12;\ 36}$ from the distribution, and so, no performance regression detected.

Chapter 4

Performance Regression Pinpointing

Despite the fact is that *statistically rigorous performance regression detection* methodology is trustworthy and reliable, by using the benchmarking tools provided by SBS introduced so far, we can only detect whether there is performance regression exists inside our program. In particular, when having a large codebases and a big number of committers to a project (like, for example, an OS, language or some framework) it is sometimes hard to find out where is the change that causes the performance regression. In fact, just one very little false change in deep down to the standard library implementation may result in a largely significant lost in overall performance. Such cases actually exist in reality and many of them have been found so far.

Those changes are called *performance regression* and should be considered as implementation errors, not causing the system to crash but heavily influencing speed and load when they are left unchecked. Scala (or any other language and/or framework) developers and library designers may need a tool to debug those errors to help them get the regression point(s) fixed before delivering the product to their clients.

In this section, we would like to introduce a regression detecting method, called *performance regression pinpointing*, which is already implemented as a benchmarking mode run in SBS. This methodology should be used to point out the as-small-as-possible piece of code that causes the performance regression. The main idea is using bytecode instrumentation to measure the performance of a piece of code and statistically detect its difference from the performance of a previous version implementation kept from earlier builds.

Starting from here, we will call a performance regression point a *bottleneck*.

4.1. Prevalent bottleneck finding methodologies

4.1.1. Profiler

When it comes into bottleneck finding inside a code snippet, the most preferred solution is to use a profiler. With an enough-powered profiler, one can measure the performance of a single method during a running pass to use for comparing in the future. This solution is not statistically rigorous thereby may lead to miss-judging and wrong conclusions as described in section 2.3.

Another drawback of this approach is the way a profiler is implemented. In Java world, a profiler is written based on a native API called Java VM Tool Interface (JVMTI) and packed as a program called an *agent*. At the profiling time, the agent is required to run along in the same JVM with the code snippet. It influences the run of the snippet to produce useful data and reports them in some I/O heavy way. That significantly violates the constraints that require the benchmarks to run on a clean runtime environment.

4.1.2. Further benchmarking

Another approach is writing more benchmarks for typically suspicious part of the benchmark that caused the performance regression. This may lead to miss-conclusions if done in "native" ways – methodologies that are not statistically rigor (see section 2.3). In statistically rigorous methodology, this requires addition manual effort to generating sample histories to compare with the performance of those parts mentioned above. With this approach, benchmarkers/developers have to repeat the process: writing more and more benchmarks, generating more and more sample histories and hoping the bottleneck will soon be found.

4.2. Main work flow

Rather than using approaches mentioned in section 4.1, we intend to dynamically programmatically point out the bottleneck lies inside the interested code snippet. Our approach is to find the bottleneck by detecting the difference in performance between two versions of Scala `.class` files:

- The current version – the newest built classes which cause performance drop

- The previous version – the classes from one of the earlier builds, which are well-tested and have an accepted performance

In compare to the prevalent methodologies described in section 4.2, our approach has several fundamental differences and advantages:

- Programmatic – not only measures the performance of a piece of code, also finds the bottleneck inside it if exists. Without any manual effort required from user, uses a binary-search like algorithm to narrow the piece of code that causes the performance drop. In the best case that the narrowed piece is a single function call expression, the whole process is recursively applied to the just-collected function.
- Clean JVM – the inspected code snippet is run in a clean new JVM without any agent and event processing comes along. More precise performance measurement, no halting in method invocations to generate events or process requests.
- Automatically performance drop detection – runs once and compares the performance difference from an earlier accepted version of `.class` files.
- *Statistically rigorous difference detection* – difference in performances is detected using *statistically rigorous performance regression detection* methodology rather than comparing derived metrics such as average, maximal, minimal running time...

The algorithms shown in following sub-sections illustrate the main flow of our technique. For convenience, we represent the inspected source code (typically a method) by "slicing" it into lists of function call expressions, called layers. A layer consists of all the function call expressions which have a same property: when any of them is called, the call stack will have the same height. The layers are numbered by their relative depth to the inspected method, which means that the layer 0 consists of only one method: the inspected method itself. For example:

```
def foo {
  bar
  baz
}

def bar {
  one
  two
```

```
}

def baz = zero
```

- layer 0: foo
- layer 1: bar, baz
- layer 2: one, two, zero

In the following sub-sections, we introduce three algorithms. One of those is called Digging Finding. It is a recursive algorithm used to navigate the overall finding process into one-level-deeper layer when the bottleneck found at the current layer is a single function call. The other two are the finding algorithms applied on the function call list of the inspected method at each layer `DiggingFind` (see section 4.2.2) visits.

4.2.1. Method body as listing function call expressions

A fact should be noticed is that mere computations (such as arithmetic operations +, -, *, / etc.) are not likely causing the performance drop because they are compiled directly into simple Java bytecodes. If we put them aside, all left in a method body is only method or function call expressions. A method body might also contain loops which cause a performance regression, but in this case the enclosing method is detected as the bottleneck. A function call expression becomes a basis unit that may produce the bottleneck.

With the instrumentation approach (see section 4.4) the body of a method is read from a `.class` file that contains the compiled bytecodes. Therefore, the method body is represented as a flat list of function call expressions with no respect to other bytecode instructions corresponding to basic operations such as adding, loading, jumping etc. For example, a **`def`** `foo = bar(baz)` is translated as `List(baz, bar)` just like a **`def`** `foo = baz + bar`.

In the next sections, we use the term *function call expression list* to address all of the lists of function call expression described above.

4.2.2. Digging finding

In the first step, the inspected method, which is the method we intend to locate the bottleneck inside, is checked whether to have the same list of function call expressions for both the current and the previous version of implementation. One big note is that, the inspected method body is not expected to be changed between the two versions; otherwise, an

exception is raised. The changes, if any, should occur in the implementation of layers deeper than layer 0. If any change happens at layer 0, probably it is the cause of regression, the process is decided to stop and return the inspected method as the bottleneck.

Algorithm: DiggingFind

```
Input: the method to inspect
Output: the bottleneck if exists
if (method.currentCallList matches method.previousCallList) {
  val found = binaryFind(method.callList)
  found match {
  case bn: Bottleneck => if (bn.length == 1) {
    val newFound = diggingFind(bottleneck.method)
    newFound match {
      case NoBottleneck => found
      case something    => newFound
    }
    else found
  case _ => NoBottleneck
}
else throw Error
```

Algorithm 4.1 -Algorithm bottleneck digging finding

After the function call expressions list have been checked, a narrowing algorithm, that may be Linear Finding or Binary Finding in the following sections, is applied to find the bottleneck. If the bottleneck exists, and happens to be a single function call, it will be recursively inspected by `diggingFind()` to find the inner bottleneck.

4.2.3. Linear finding

This algorithm together with `BinaryFind` (described in section 4.3.3) is used to narrow the length of the function call list which causes the performance regression.

Algorithm: LinearFind

```
Input: a list of function call expressions
Output: the bottleneck if exists
if (list.length == 0)
  NoBottleneck
else if (list.head.currentPerf == list.head.previousPerf)
  Bottleneck(callList.head)
else
  linearFind(callList.tail)
```

Algorithm 4.2 -Algorithm bottleneck linear finding

In algorithm 4.2 and algorithm 4.3, `currentPerf` and `previousPerf` respectively are the performances performed by the two, current and previous, versions of implementation. They are obtained using *statistically rigorous performance regression* methodology (see section 2.3); each running time value of the piece of code is measured by instrumenting the bytecodes complied from the program (see section 4.4 for more detail). Algorithm `BinaryFind` is a simple recursive algorithm which does comparing the performances performed by the two, current and previous, versions of implementation. It returns the first bottleneck found when the corresponding function call is detected that currently performed statistically worse than the previous version. In the case there is no bottleneck lies inside the inspected method, the algorithm returns `NoBottleneck` when it receives an empty list as the argument `callList`.

4.2.4. Binary finding

The function `BinaryFind` defined by the pseudo code in Listing 4.2 briefly describes how we can "precisely" find out the performance bottleneck inside a piece of code which is simply represented as a list of method call expressions.

Algorithm: BinaryFind

```
Input: a list of function call expressions
Output: the bottleneck if exists
if (callList.currentPerformance ==
    callList.previousPerformance)
  NoBottleneck
else if (callList.length == 1)
  Bottleneck(callList)
else {
  val (first, second) = binaryDivide(callList)
  try {
    val firstFound = binaryFind(first)
    firstFound match {
      case _: Bottleneck => fisrtFound
      case _             => binaryFind(second)
    }
  }
  catch { case Error => Bottleneck(callList) }
}
```

Algorithm 4.3 - Algorithm bottleneck binary finding

The algorithm input is initially a list of all of the method call expressions in the whole method body. Actually it is not as simple as a relation operation == (equivalent to `equals`)

like the one in the pseudo code, at the first step of the algorithm, we use *statistically rigorous performance regression detection* methodology to achieve the runtime performance and detect whether the difference exists between the performances of the current version and the previous one. We will describe the process in more details in section 4.5 and 4.6.

When there is no statistically significant difference detected, we return the result indicates there is no bottleneck found inside the inspected piece of code. (Note that type `NoBottleneck` and type `Bottleneck` are both subtypes of type `Found` in the pseudo code).

Otherwise, the inspected piece of code is a bottleneck itself, and it alone causes a significant drop in the overall performance so that it can be detected using *statistically rigorous performance regression detection* methodology. One may satisfy with this result and happily return the snippet itself to the user. But with our methodology, we intend to be more precise and specific. We try to narrow the range of the bottleneck as small as possible, and in the best case, to a single method call expression.

Pseudo function call `binaryDivide()` is used for splitting the original list into two function call expression lists, both have their length which are equivalent to each other. The narrowing operation is recursively applied again and again until no statistically significant difference found, or, the function call expression list becomes a single function call expression.

4.3. Bounds on running time

The whole bottleneck finding process will terminate normally on the cases specified below:

- when reaches the depth user specifies
- when reaches a method specified to be ignored
- when reaches a native method
- when reaches a method has been inspected before

In the case there is some error occurs when running performance measurement process or there is no statistically significant difference detected inside the inspected method, the

whole finding process will stop also. And finally, the current bottleneck collected is reported to user.

Given the instrumentation, backup-ing and difference detecting processes run in constant time. Let:

- *t* is the time needed to run the inspected piece of code for both the current and previous versions
- *D* is the maximal depth of the digging process, specified by user
- *d* is the actual depth of the digging process, $d \leq D$
- s_i is the length of the list of function call expressions of the inspected method at layer *i* $(i \leq d)$

Suppose the length of the function call expression list at layer *i* is $s_i = 2^{k_i}$, the binary finding algorithm runs at most k_i times to narrow the list to a single function call expression. Therefore, maximal running time at layer i is

$$B_i = t \cdot log_2(s_i)$$

The maximal total running time (which also is the normal one) for all *d* layers is

$$B = \sum_{i=0}^{d} B_i = t \cdot \sum_{i=0}^{d} log_2(s_i) = t \cdot log_2(\prod_{i=0}^{d} s_i)$$

In the function call expression list at layer i, the probability of a function call is the bottleneck is the first bottleneck is $\frac{1}{s_i}$. With linear finding algorithm, the average running time at layer i is

$$L_i = t \cdot \frac{1}{s_i} \cdot \sum_{i=1}^{s_i} i = t \cdot \frac{1}{s_i} \cdot \frac{s_i(s_i + 1)}{2} = \frac{t(s_i + 1)}{2}$$

The average total running time for all *d* layers with linear finding algorithm is

$$L = \sum_{i=0}^{d} L_i = \frac{t}{2} \cdot \sum_{i=0}^{d} (s_i + 1) = \frac{t}{2}(\sum_{i=0}^{d} s_i + d + 1)$$

The maximal running time at layer i, which will be taken when the bottleneck is the last function call in the function call list at layer i, is

$$L_{max;\,i} = t \cdot s_i$$

The maximal total running time for all d layers with linear finding algorithm is

$$L_{max} = \sum_{i=0}^{d} L_{max;\,i} = t \cdot \sum_{i=0}^{d} s_i$$

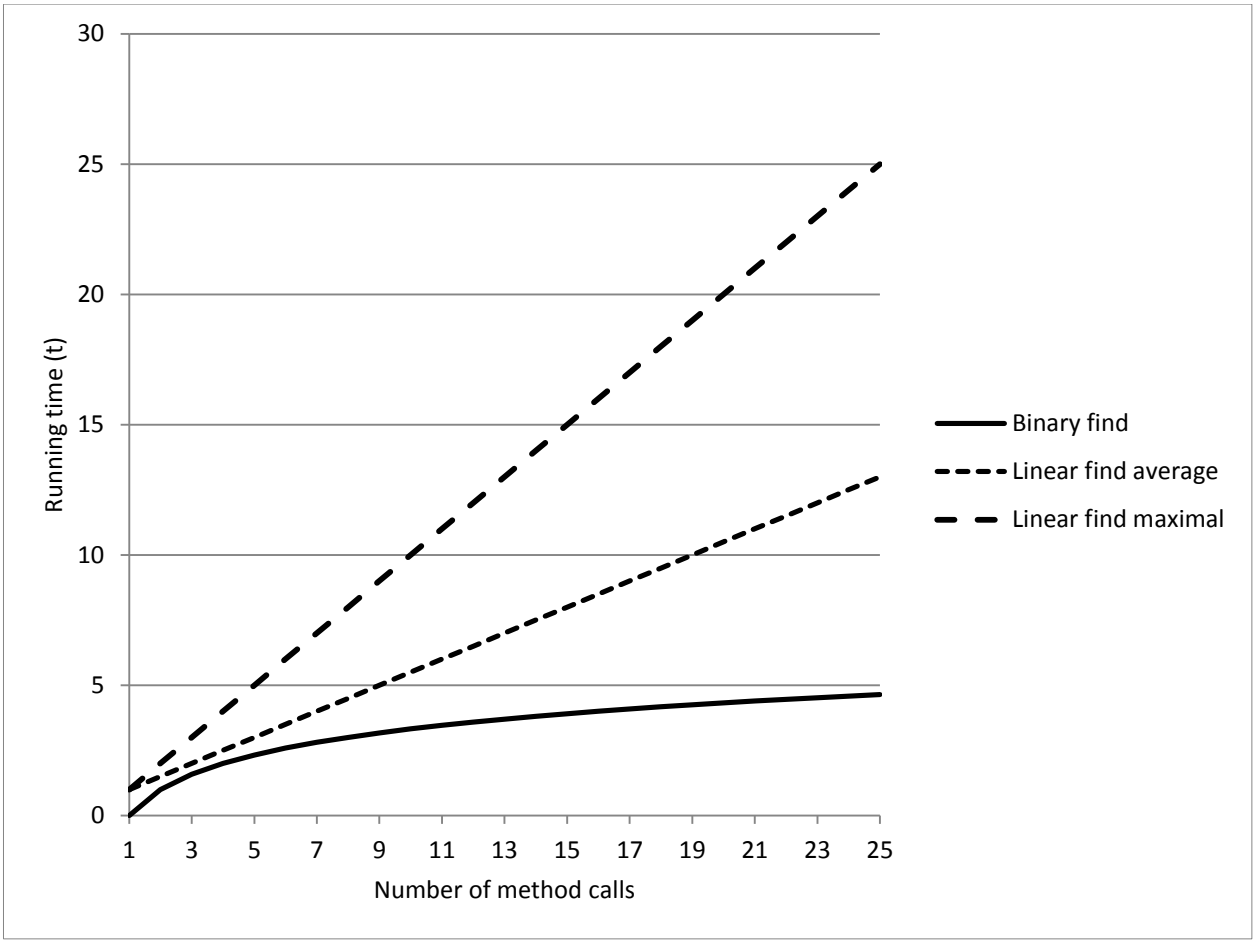

Figure 4.1 – Running time at one layer with increasing length of code

Figure 4.1 visually illustrates the comparison of the complexity – running time of Binary Finding and Linear Finding in its average case and worst case. The time consumed by Binary Finding becomes significantly small in compare to Linear Finding when the length of the function call list increases. This can be explained by the o-notation of Binary Finding is $O(\log_2(n))$ while Linear Finding is O(n) with n is the length of the function call list of the inspected method at some of the layers.

4.4. Scala class instrumentation

The most important step of *performance regression pinpointing* is to achieve the series of running time of a small piece of code snippet. Each performance measurement (which is one element of the series) needs to be as accurate as possible, so we do not allow any influence to the measurement process. Those activities such as: generating event/request, I/O, interacting with other OS processes… should be prevented.

In prevalent methodologies, one may create samples by interrupting the execution at some instruction, using the old Java Virtual Machine Profiler Interface (JVMPI) introduced in Java 1.4.2. JVMPI has been removed since JDK 6 and replaced by Java Virtual Machine Tool Interface, which is a part of Java Platform Debug Architecture (JPDA). JPDA defines an event/request based interacting protocol between the host and client JVMs. Probably these are not the best choices to be applied in our approach.

Instead, we directly instrument the Scala `.class` file in which the inspected method is declared. The most basic implementation to do instrumentation to get the running time is to inject two instructions, say, `start()` and `stop()` at the beginning and the end of the inspected code:

```
start() // injected using instrumentation
// do something really costs much time here
stop()  // injected using instrumentation
```

Listing 4.1 – Basic instrumentation to measure running time

What `start()` and `stop()` do are essentially get the two values of current system time with `System.currentTimeMillis()` (written with Java) and calculate their subtraction to achieve the running time. But the problem is that, `start()` and `stop()` are not available in the constant pool of the inspected class. That leads to the necessary of re-compiling and loading the class again, what is, by specification, not allowed by Sun's JVM. In fact, class reloading is allowed by Sun's implementation of JVM, but the JVM has to be invoked with the option `--debug`, which enables JPDA in the JVM working session with other agents run along. That certainly violates our requirements about a clean JVM for measuring performance.

Fortunately, we have solved those problems completely with our combination of measurement methodology and a Java bytecode instrumentation library call Javassist[23].

Javassist is a class library for editing bytecodes in Java. It provides an API level of Java source code, which means, user can simply manipulate a Java class with its high level objects such as class, constructor, field, method… without the need of knowledge of the specification of the Java bytecode. It comes along with a Java compiler of itself to recompile the class after being modified.

After instrumentation phase, the inspected class is loaded and run in a clean brand new JVM invocation to be measured the performance. It is run under the control of a *harness* (which is described in section 3.3). The harness class should implement the two method `start()` and `stop()`, and the path to it had been provided to Javassist Java compiler to have `start()` and `stop()` available.

Another difficulty is to find the implementation of the interested piece of code in the classpath. Scala has lots of automatically generated `.class` files for `trait`s, `object`s, anonymous functions… and their actual body probably is not located in the class file which has the name of the class they were declared in. Even though one might have written the entire Scala program himself, at the first glance he may not know which `.class` file contains the body of the method he is interested in at all.

Finally, to make things even more complicated, currently there is no tool or library supports Scala class instrumentation. Javassist has no specific component to be used on Scala class files. Essentially, at bytecode level, Scala class is just an ordinary Java class. But there some important transitions from Scala to Java happen at compile time: Scala class hierarchy, `val` and `var` implementation, function value, `trait` and `object`…

To overcome those problems above, the knowledge of how Scala codes compile into Java bytecodes is necessary. We explain some of those along with the process to get the implementation of the inspected method can be instrumented using Javassist as follows:

- `val` is translated into two components: a `private final` field together with an accessor method which has the same name, same access level with the `val` and an empty parameter list. `val` accessing through Java is done by using the accessor method. For example:

[23] http://www.csg.is.titech.ac.jp/~chiba/javassist/

```
// defined in Scala
class A { val a = 1 }

// translated in Java
class A {
  private final int a = 1;
  public int a() {
    return this.a;
  }
}

// access in Scala
val x = (new A) a

// access in Java
int x = (new A).a();
```

Listing 4.2 – Scala to Java example – `val`

- `var` is translated with the same name convention as `val`, but the generated private field is not final. In addition, `var` has a modifier method which has the name composed using the name of the var prefixed by `_$eq`. This modifier method in general has one parameter named `x$1` which has the same type with the `var`. For example:

```
// defined in Scala
class A { var a = 1 }

// translated in Java
class A {
  private int a = 1;
  public int a() {
    return this.a;
  }
  public void a_$eq(int x$1) {
    this.a = x$1;
  }
}

// modify in Scala
(new A).a = 0

// modify in Java
(new A).a_$eq(0);
```

Listing 4.3 – Scala to Java example – `var`

- `class` and `abstract class` is compiled just like Java classes

- `trait` is a special mechanism in Scala, used for *multiple inheritance* (*mix-in* composition) and implementing *rich interfaces* (described in more details in section 2.1). For more than one trait to be able to be mixed-in together, the only way in Java is to be represented as interfaces. In fact, a trait is compiled into two class files:
    - one has the original name and is an interface with all the method prototypes included.
    - one has the original name suffixed by `$class` and is an `abstract class`. All the methods that have already been implemented in the trait definition are re-defined in this abstract class as `static` methods. These static methods' respective parameter lists are prepended by a parameter name `$this` which has the type which is the generated interface. Also, all the references to `this` in the methods' bodies are changed into `$this`. Classes that have the trait mixed-in are compiled implementing the generated interface and forward all the calls to methods in the trait to the corresponding static method in the generated abstract class with the first arguments always are `this`.

```
// defined in Scala
trait T { def foo = 1 }

class C extends T

// translated in Java
interface T {
  int foo();
}

abstract class T$class {
  public int foo(T $this) {
    return 1;
  }
}

class C implements T {
  public int foo() {
    return T$class.foo(this);
  }
}
```

Listing 4.4 – Scala to Java example – `trait`

- `object` is another special type definition in Scala, it supposed to be the singleton class – a class that always have a single instance at runtime. An `object` is compiled into two `.class` files:
    - one has the original name and is a `final class` with all the methods from the object definition implemented as `static final`. Everything that they do is to, from the other generated class, first get the value of a special static field named `MODULE$` (described below) and then invoke the corresponding method on it.
    - one has the original name appended by $ and is also an final class. All the methods that were in the object definition are re-defined in this final class. It has no constructor and has a special additional field named `MODULE$` which has its type this class itself. This field is the only instance of the "object class". For example:

```
// defined in Scala
object O { def foo = 1 }

// translated in Java
final class O {
  public static final int foo() {
    return O$.MODULE$.foo();
  }
}

final class O$ {
  public static final O$ MODULE$ = new O$;
  public int foo() {
    return 1;
  }
}
```

Listing 4.5 – Scala to Java example – `object`

We do not expect that it is necessary for users to have the knowledge about the back-end design of Scala programming language and the instrumentation approach. Therefore, method bodies inside class definitions is looked for based on our inference and assumption. Typically, the class looking process first looks for the desired method in class that one's name is prefixed by `$class` and `$`. If nothing found in there, it then will look for method body in the class that has the name exactly matches what users specified. The reason of doing that is, the fact that the true definition of a method is not lies inside the class users expected it to do is

inferred. What is defined in user specified class is typically an interface or a bridge method, which we do not want (nor are able to) do instrumentation on.

So, following is the four-step process we advocate to do instrumentation on a method – specified by user that – named `foo` and defined in the class (or trait or object) named `Clazz`:

- Look in classpath for the class named `Clazz$class`. If it can be found, it is the class we want. Otherwise, look for the class named `Clazz$` with the same expectation. If `Clazz$` can neither be found, the target class is `Clazz`. `Clazz` surely exists for user program to be able to run
- Backup a copy of `.class` file contains the class has just been found, away from `classpath` because we do not want a `ClassLoader` to load it instead of the instrumented one
- In the class found, look for the method named `foo`. Do instrumentation on `foo` using Java bytecode instrumentation library
- After running, remove the instrumented class from `classpath` and restore the original version from backup place for future use

This process has been successfully applied in practice within the implementation of Scala Benchmarking Suite and worth to try until an instrumentation library specific for Scala programming language comes into place.

4.5. Backup .class files

As mentioned in the end of the previous section, some activities of moving `.class` files around are necessary for the measurement phase to be correct. This section gives a brief introduction to Java class loading and describes the need of backup-ing `.class` files in or out the `classpath` as well as our approach to provide the right version of a `.class` file to the JVM.

With our approach to do comparing over two versions of `.class` files to detect the bottleneck, it is necessary to point out that, at a typical point in time during the bottleneck finding process, there are maybe up to three versions of the same `.class` files:

- The current version – a set of `.class` files contain the definitions of current build

- The previous version – a set of `.class` files contain the definitions of previous build
- The instrumented version – a set of `.class` files which can originally be the current or the previous classes. These `.class` files contain the class we have instrumented to achieve its measurements

It is very important to be able to load the right version during measurement phase. Or we will end up, without any idea that we do, measure the wrong performance or even be able to measure nothing at all. To explain the solution to this problem, a general knowledge of Java class loading may be necessary.

Essentially, a Scala runtime session is just a Java runtime session with the main class which is `scala.tools.nsc.MainGenericRunner` (if Scala is about to run on JVM and compiled to Java bytecodes instead of CLR). This main class loads and runs users' Scala programs using *reflection*. Scala has its own class loader, but the class loaders are implemented by inheriting from Java class loaders, thereby leaves all the native activities to Java.

A class is known by Java runtime environment when it is referenced by its name by a class that has been already loaded. As referenced in Oracle's documentations, the order of searching locations for a class loader is as follows:

- *Bootstrap classes*: the runtime classes in `rt.jar`, internationalization classes in `i18n.jar`, etc.
- *Installed extensions*: classes in JAR files in the `lib/ext` directory of the JRE, and in the system-wide, platform-specific extension directory.
- *The user class path*: classes, including classes in JAR files, on paths specified by the system property `java.class.path`. If a JAR file on the class path has a manifest with the `classpath` attribute, JAR files specified by the `classpath` attribute will be searched also. By default, the `java.class.path` property's value is the current directory. It can be changed by using the `-classpath` or `-cp` command-line options, or setting the `CLASSPATH` environment variable.

Accordingly, given a class name at runtime, the class locates in `rt.jar` is loaded no matter whether another one with the same name exists in user `classpath`. But we could not

find the search order rules for the user-defined `classpath`. In many tests and tries on Sun's JVM, we can see that classes are searched following the first-come-first-serve order. That means the class locates in the first location specified in classpath will be loaded. But there is no evidence that we know of, and it may depend on the implementation of the JVM. So, no conclusion at all.

Instead of relying of luck, we decided to move the undesired classes out of the `classpath`. This is done by (i) loading the class (ii) tracing back to its location and (iii) moving it into the backup location with the respect to platform-specific directory structure for packages. It requires that the locations of the previous version and the instrumented version are not included in the original `classpath`. Finally, to make it sure, the path to the previous version is prepended to classpath during its measuring performance while the path to the instrumented version is placed at the first location in both of the measurement phases.

There exists another solution, which seems better, which is defining a custom class loader. This class loader is supposed to find, instrument and transform the original classes at measurement phase. The reason why that is not preferred in this case is that the SBS is supposed to work with problems in the Scala standard library. It suggests that the class is going to be transformed likely has been loaded. The approach to load and redefine a class at runtime is not interesting with the constraints about the clean JVM environment for benchmarking.

4.6. Package scala.tools.sbs.pinpoint

Performance regression pinpointing has been implemented as a benchmarking mode run in the Scala Benchmarking Suite, which will soon be integrated into Scala's trunk. This section describes the structure and working flow of that implement – package `scala.tool.sbs.pinpoint`.

The important components consists of

- Trait `Scrutinizer` – the central trait of the package. Trait `Scrutinizer` extends trait Runner and is implemented by class `MethodScrutinizer`
- Trait `ScrutinyRegressionDetector` detects the statistically significant running time difference between the two versions of the benchmark classes. It is implemented by the class `MethodRegressionDetector`

- Package `strategy` holds the traits which define all the higher-order function which are factored out to do their specific work:
    - Trait `TwinningDetector` – serially runs the benchmark on its two versions of classes. After the performances is achieved, detects the difference between those
    - Trait `PreviousVersionExploiter` – backups the `.class` files corresponding to the current version and run the benchmark on its previous version
    - Trait `InstrumentationRunner` – does some kind of instrumentation and run the benchmark to achieve the desired metrics
- Package `instrumentation` consists of the trait `CodeInstrumentor` and its Javassist-based implement, class `JavassistCodeInstrumentor`. Package `instrumentation` defines all the necessary operations to accomplish our goal measuring the performance of a specific piece of code
- Package `bottleneck` holds the trait `BottleneckFinder` and its sub-classes `BottleneckDiggingFinder` – `BottleneckBinaryFinder` which implement the algorithm `DiggingFind` and `BinaryFind` described in the sections 4.2.2 and 4.2.4 (we did not implement algorithm `LinearFind` in SBS)
- Besides, there are several supporting traits and classes which define benchmarks, exceptions, results, etc.

In Figure 4.2, we visually illustrate the simplified static structure of the package pinpoint. Components included are only traits and classes which define operations controlling the finding process

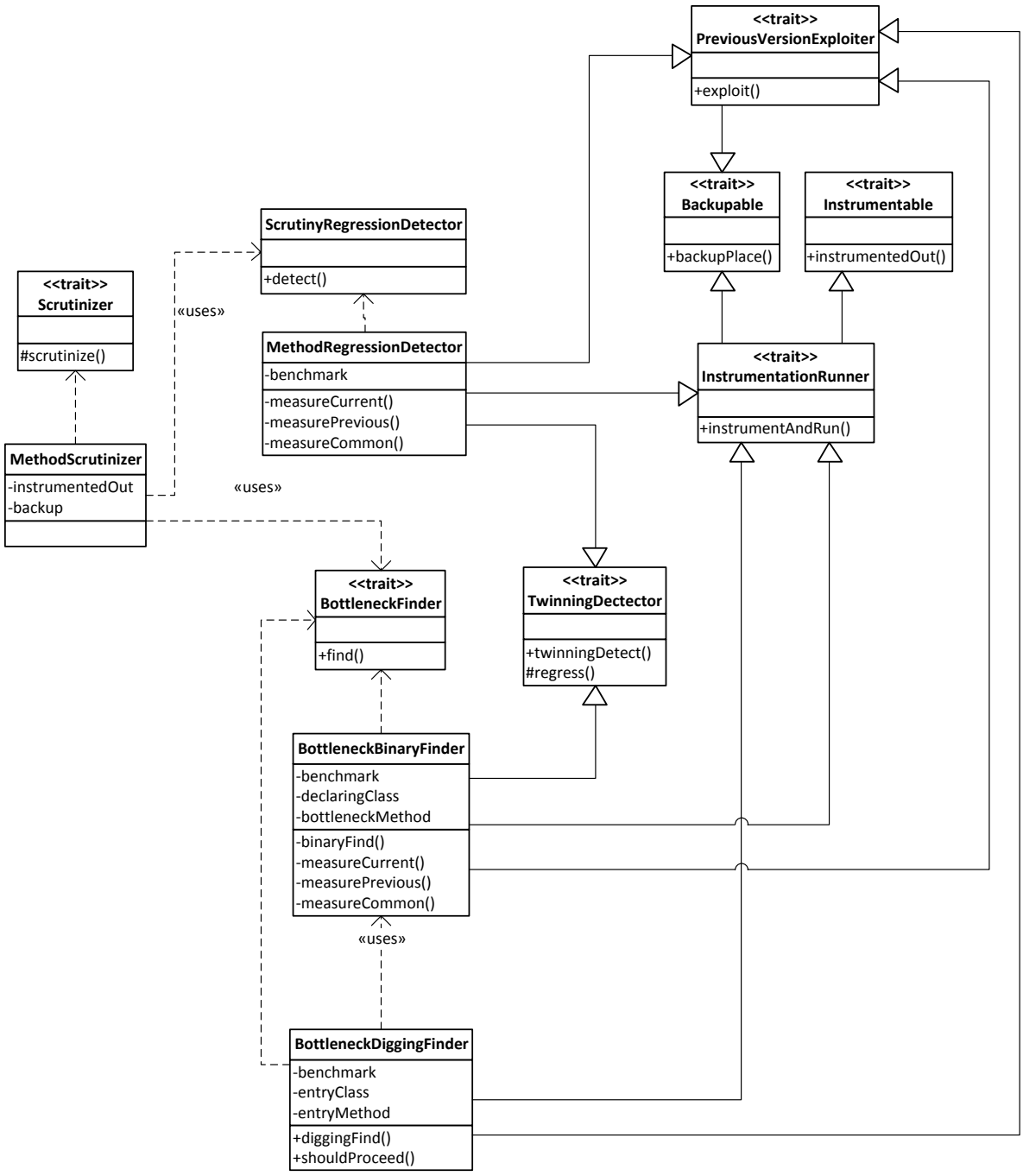

Figure 4.2 – Package pinpoint's simplified class diagram

The work flow in a pinpoint benchmarking mode run is already described in section 4.2 except the first step to compare the two, current and previous, performances of the inspected method. Following is the illustration for a run process finding the bottleneck

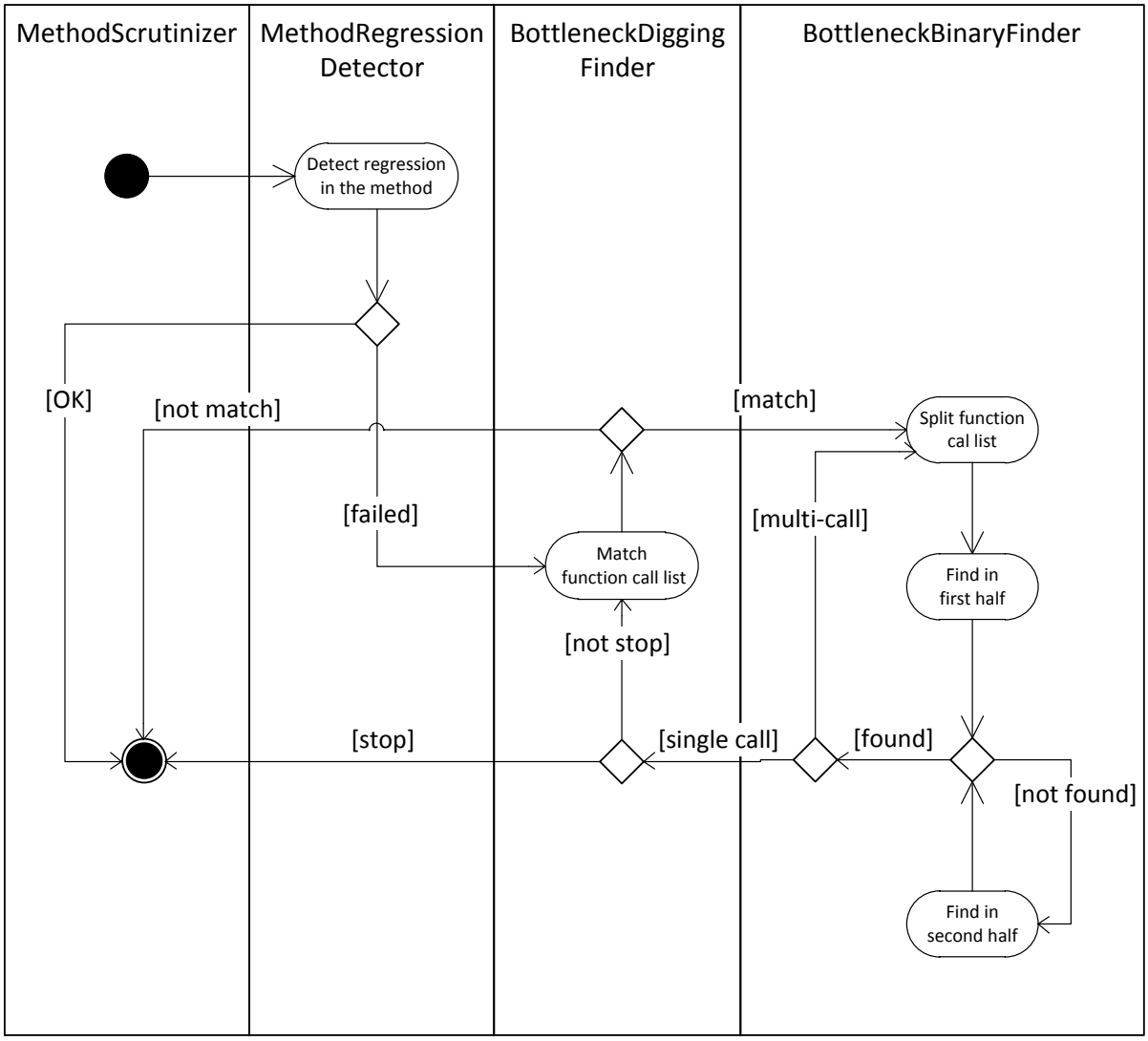

Figure 4.3 – Bottleneck finding activity diagram

4.7. Case study

This section explains the benchmark design and implementation to illustrate the process of finding the bottleneck by performance regression pinpointing. Firstly, we introduce a real life performance problem found in the Scala bug tracking system which is easy to reproduce and understand. Next is the implementation of a benchmark that has the problem intended to appear in. Finally, we briefly explain and illustrate the process of finding the bottleneck inside the benchmark.

4.7.1. Problem with scala.collection.mutable.ListBuffer.size

The problem raised by the ticket number SI-4933 in the Scala bug tracking system. It stated that the operation of method `size` from class `scala.collection.mutable.ListBuffer` took the complexity of O(n) instead of O(1) like method `length` although they two had the same meaning in the case of sequences. The reason why that happened was that `length` returned the value of a private variable which was updated every time elements were added or removed but in the meantime, `size` was inherited from `scala.collection.TraversableOnce` counting all the elements currently in the list (more details about `scala.collection` in Chapter 27 – [1]).

The problem fixed in Scala revision r25684 by overriding `size` to return the result of `length`. In the next section, we use the latest Scala revision as the previous version which has `ListBuffer.size` run in O(1). To produce the current version that drops in performance, we comment out the overriding of `size` to have it run in O(n). The benchmark is composed in the way which does lots of call to `ListBuffer.size` on large length lists so that the difference in performance of O(1) and O(n) can be detected.

4.7.2. The pinpointing benchmark

The benchmark used in this case study is called `PinpointDemo`. It runs lots of invocations of `ListBuffer.size` to reflect the problem described in the previous section along with some other costing time activities. The important parts of the benchmark implement is shown in Listing 4.6

```
class PinpointDemo {

  val failure = ListBuffer_size
  val ok      = Iterator_flatten

  def run() = {
    bridge
    ok.main
  }

  def bridge = {
    foo
    failure.run
  }

  def foo = Thread sleep 50
```

```
}

object ListBuffer_size {

  val lb =
    for (_ <- 1 to 100000) yield ListBuffer((0 to 50): _*)

  def run() {
    val ls = (1 to 15000) map (lb map (_ size))
    ls foreach (_ => ())
  }

}

object Iterator_flatten {

  def main /* does something costing time here */

}
```

Listing 4.6 – Pinpointing benchmark `PinpointDemo` – simplified

Class `PinpointDemo` defines 3 methods `run`, `bridge` and `foo` together with 2 fields `ok` and `failure`. Their roles and operation as follow:

- `run` calls `bridge` before calling the method `main` of the object stored in field `ok`
- `bridge` calls `foo`, then calls the method `run` of the object stored in field `failure`
- `foo` makes the whole benchmark sleep for 50 milliseconds
- `ok` holds the object `Iterator_flatten`
- `failure` holds the object `ListBuffer_size`

The two objects `ListBuffer_size` and `Iterator_flatten` are put in fields to force their data initialized before running to avoid the data initialization influencing the performance.

Object `ListBuffer_size` makes sure that the performance regression exists. It consists of a very large number of `scala.collection.mutable.ListBuffer` objects and performs operations based on lots of invoking the method `size`. The `object` is defined in this way so that when the overriding of `size` forwarding to `length` is commented out, `PinpointDemo.run` will introduce a significant lost in performance. The last line of `ListBuffer_size.run` is used to fool the JIT compiler of the JVM not to

optimize away the cloning process. The method `foreach` applies the anonymous function `_ => ()` to each element of `ls`.

The inspected method – entry of the finding process – is set to be the method `run` of class `PinpointDemo`.

The depth of the finding process is set to 2.

The number of measurements to be kept for each series is set to 13.

4.7.3. Bottleneck finding process

This section follows the finding of the bottleneck inside `PinpointDemo.run` step by step and explains what happening at the time.

The inspected method is method `PinpointDemo.run` as specified in the previous section. The method body is translated into a function call expression list as shown in Listing

```
// method definition
def run() = {
  bridge()
  ok.main()
}

// function call expression list
PinpointDemo.bridge:()V
PinpointDemo.ok:()LIterator_flatten$;
Iterator_flatten$.main:()V
```

Listing 4.7 – Function call expression list of method `PinpointDemo.run`

It consists of 3 function call expressions:

- `PinpointDemo.bridge` – call to method `bridge()` of the class `PinpointDemo`
- `PinpointDemo.ok` – the getter of field `ok` which holds the `object Iterator_flatten`
- `Iterator_flatten$.main` – call to method `main()` of the object `Iterator_flatten`

The first step is to detect performance regression on the piece of code represented by the function call expression list above. The result is displayed in Figure 4.4:

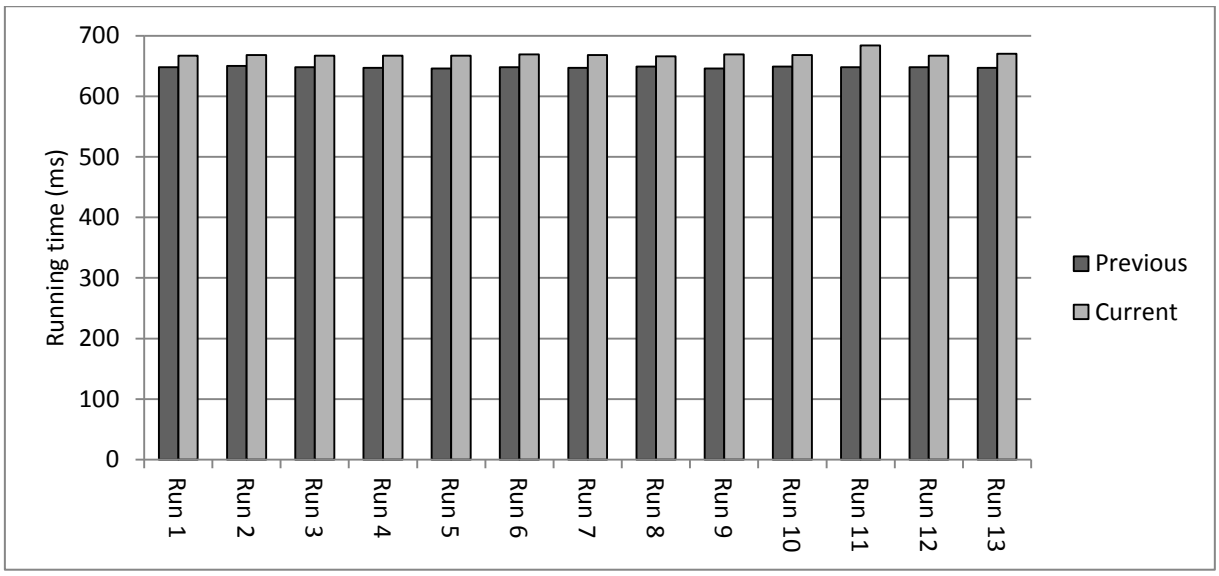

Figure 4.4 – Pinpoint performance comparison – `PinpointDemo.bridge` to `Iterator_flatten$.main`

The confidence interval computed from the two series of running time is [-23.58; -18.88] at the confidence level reduced to 90%. This means that the overall performance drops about 20 milliseconds and the performance regression does exist.

In the next step, the function call expression list is split into two shorter lists to perform the `BinaryFind` algorithm. The first list consists of only one function call – `PinpointDemo.bridge`. Its performance comparison is shown in Figure 4.5.

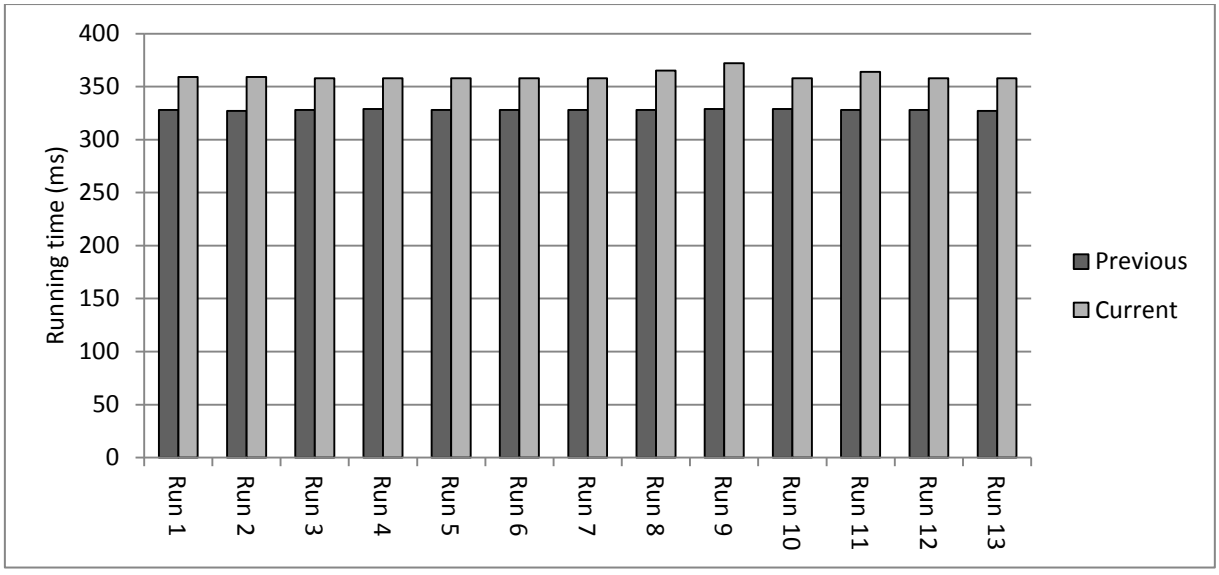

Figure 4.5 – Pinpoint performance comparison – `PinpointDemo.bridge`

The confidence interval computed is [-34.29; -30.02] at confidence level 90%. It indicates that `PinpointDemo.bridge` is actually a bottleneck causing the performance drop about 30 milliseconds to the method `PinpointDemo.run`.

The finding process now ignores the second half of the list and digs into method `PinpointDemo.bridge` hoping to find the bottleneck inside it. The inspected method is now `PinpointDemo.bridge` and its body is translated into a function call expression list as shown in Listing 4.8

```
// method definition
def bridge = {
  foo
  failure.run
}

// function call expression list
PinpointDemo.foo:()V
PinpointDemo.failure:()LListBuffer_size$;
ListBuffer_size$.run:()V
```

Listing 4.8 – Function call expression list of method `PinpointDemo.bridge`

Figure 4.6 depicts the performance comparison for the whole list.

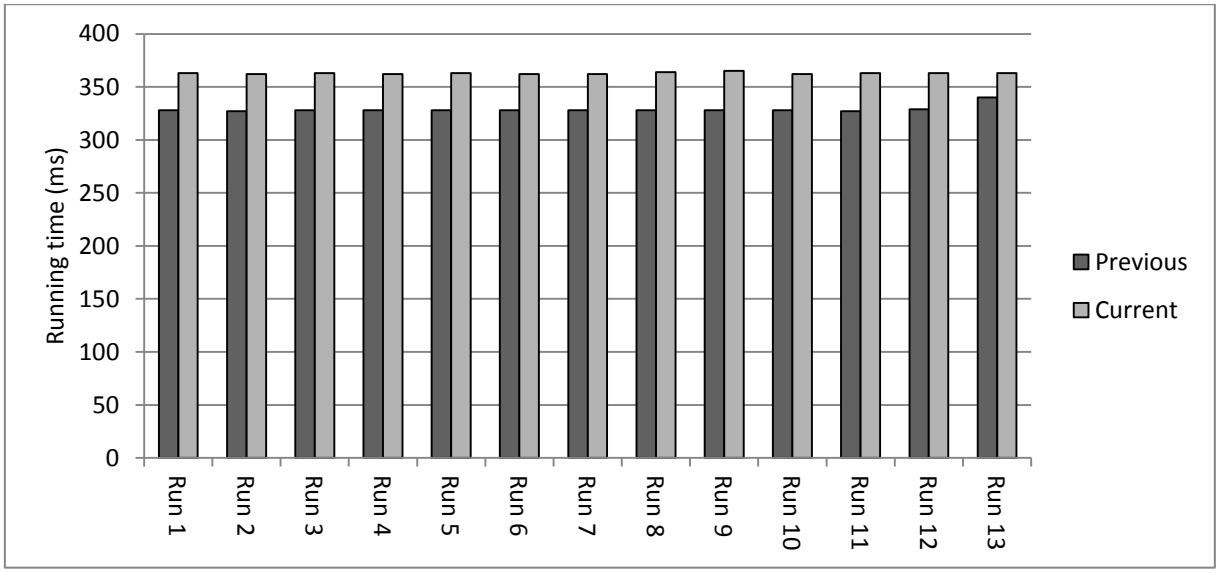

Figure 4.6 – Pinpoint performance comparison – `PinpointDemo.foo` to `ListBuffer_size$.run`

The confidence interval computed is [-35.72; -32.28] indicating the existence of performance regression. `BinaryFind` algorithm now splits the list into 2 halves, the first half has only one function call to `foo` and the other consists of the two remaining.

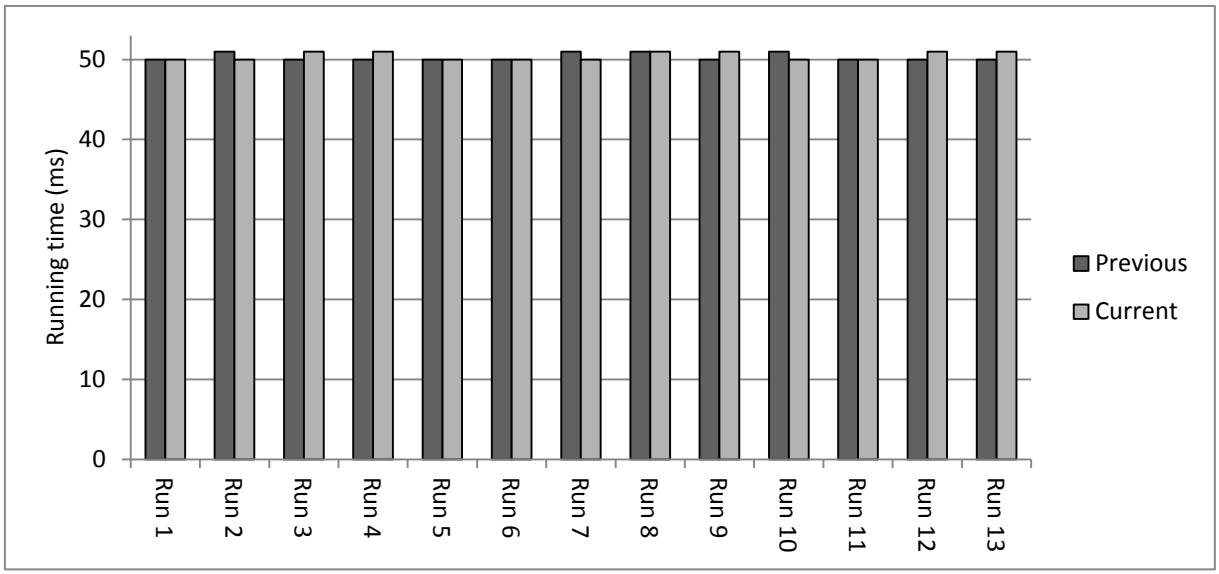

Figure 4.7 – Pinpoint performance comparison – `PinpointDemo.foo`

Figure 4.7 shows that the performance of `foo` is just about 50 milliseconds and no change exists between the two versions. The confidence interval is [-0.70; 0.40] at confidence level 99%, no performance regression detected. So, the finding process switches to the second half which is the list consists of the getter `PinpointDemo.failure` and `ListBuffer_size$.run`. Their performance comparison is depicted in Figure 4.8:

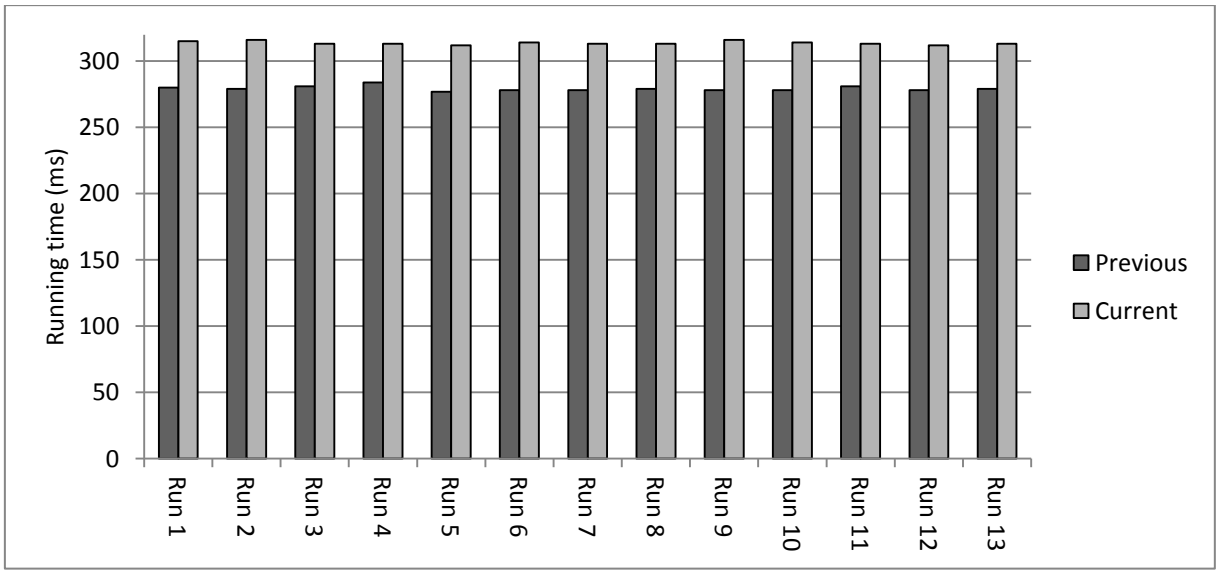

Figure 4.8 – Pinpoint performance comparison – `PinpointDemo.failure` to `ListBuffer_size$.run`

The current version shows its significantly lost in performance by over 30 milliseconds. Therefore, the function call expression list is split again into two. Each of the two consists of only one function call: `PinpointDemo.failure` and `ListBuffer_size$.run` respectively.

With the function call `PinpointDemo.failure`, this is the getter of the field `failure`. It does not consume much running time, so all measurements result in zeros – no performance regression detected.

In the contrary, `ListBuffer_size$.run` costs much running time of the benchmark and shows the poor performance. In the current version, it runs slower by 30 milliseconds in the comparison to the previous one as illustrated in Figure 4.9:

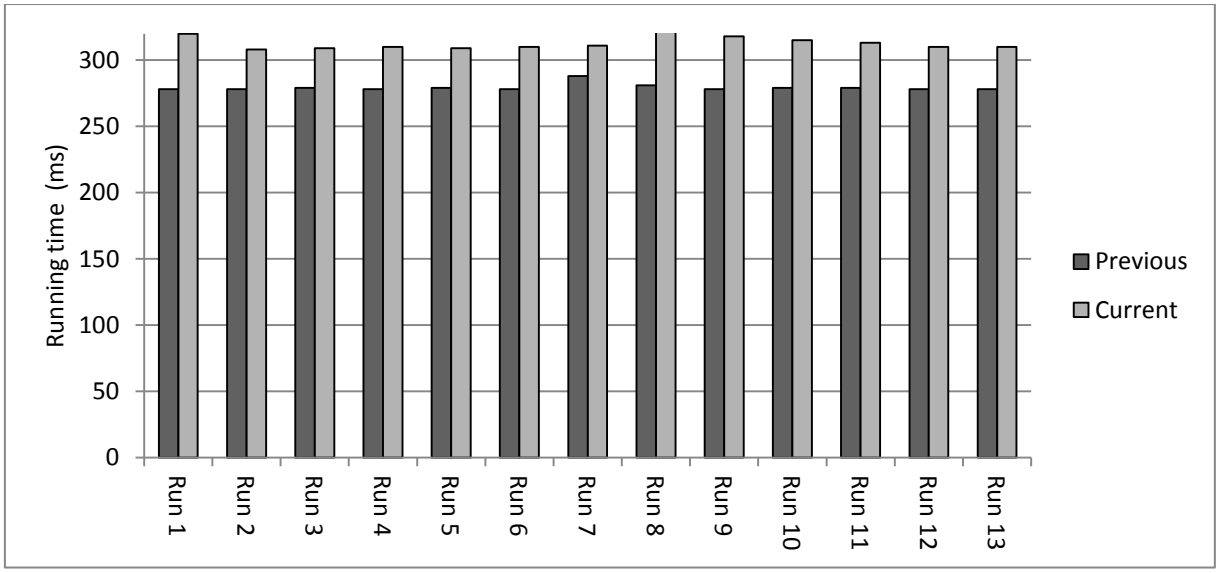

Figure 4.9 – Pinpoint performance comparison – `ListBuffer_size$.run`

The confidence interval computed is [-36.76; -30.78] at confidence level 90%. This indicates that `ListBuffer_size.run` is the bottleneck. Because the bottleneck locates at the layer 3, at this point, the finding process has reached the depth of 2 specified by user. The process stops here and `ListBuffer_size.run` is returned as the bottleneck which is found.

Chapter 5

Conclusions

In the preceding chapters, we have described the backgrounds which are an overview on the Scala programming language, issues concerning difficulties of benchmarking on dynamic compilation platform and the statistically rigorous methodology for qualifying Java performance (Chapter 2). In Chapter 3, we introduced the tool named Scala Benchmarking Suite – SBS which is designed mainly being intended to detect various kinds of regressions on Scala standard library and Scala compiler. Finally, in Chapter 4, we described our approach to find the underlying performance bottleneck by combining bytecodes instrumentation with statistically rigorous performance detection methodology. We next step back and reflect on the significance of this work.

5.1. Scala and dynamic language benchmarking

Scala is a programming language created in Programming Methods Laboratory – LAMP, EPFL - Switzerland. It supports both object-oriented and functional programming styles with a concise syntax and advanced features.

Scala can compile into Java bytecodes to be run on Java Virtual Machine. JVM is a dynamic compilation language runtime environment which does most of the code optimizations at runtime. It also comes with an advanced garbage collection mechanism. Those advanced features run unmanageable by user and thereby introduce noises and uncertainties to the performance measurements.

A methodology has been advocated to use statistics theory as a rigorous data analysis approach for dealing with the non-determinism altogether with the experiment designs to evaluate performances.

5.2. Scala Benchmarking Suite

Scala Benchmarking Suite is a tool used to evaluate the performance of programs written in the Scala programming language and to detect performance regressions caused by changes to the language. It is an extensive tool for tracking program performance and detecting performance regressions. SBS is integrated with Scala nightly build system as well as can be used as a standalone tool for statistically rigorous benchmarking on Scala.

Currently, SBS has the ability to measure and statistically detect performance regressions in start-up or steady-state of a Scala program. It can also profile a certain metrics during a benchmark run and the ability to point out the piece of code that causes the performance regressions.

5.3. Performance regression pinpointing

With benchmarking, we can only detect whether there is performance regression exists inside our program and have to do guessing on almost all of the post-process to find out the bottleneck.

Based on the fact that developers and/or library designer lack their useful tools to automatically locate the performance bottleneck, we introduced a methodology to programmatically point out the as-small-as-possible piece of code that causes the performance regression. We have described the main work flow which is recursively repeating the regression causer narrowing process in a method body. Also, we advocated the behind-the-scene trick which is using bytecode instrumentation to measure the performance of a piece of code and statistically detect its difference from the performance of a previous version implementation kept from earlier builds.

5.4. Future work

Being limited in time we have just been able to come so far and there are things have to be improved. The strengthens of *performance regression pinpointing* currently come up with the following two approaches

- One is to (re-) generate the AST for the inspected method and do instrumentation at the first level nodes of the tree. The AST can be achieved through the compilation of the benchmark or de-compilation `.class` file
- The other one is to record all of the function invocations through one running pass of the inspected method and use the invocation-dependency-graph as the input for narrowing algorithms

Another thing to do is continuing maintaining SBS. Not only because it is the biggest and most useful one among our projects, working on SBS also allows us to have more experience programming in the very interesting language Scala.

References

[1] M. Odersky, L. Spoon & B. Venners. *Programming in Scala, Second Edition*. Artima Press, Walnut Creek, California, 2010.

[2] A. Georges, D. Buytaert, L. Eeckhout. *Statistically Rigorous Java Performance Evaluation*. In OOPSLA, 2007.

[3] A. Georges, L. Eeckhout, D. Buytaert. *Java Performance Evaluation through Rigorous Replay Compilation*. In OOPSLA, 2008.

[4] B. Goetz. *Java theory and practice: Anatomy of a flawed microbenchmark*. IBM DeveloperWorks, 2005.

[5] B. Goetz. *Dynamic compilation and performance measurement*. IBM DeveloperWorks, 2004.

[6] A. Shipilev. *(The Art of) (Java) Benchmarking*. Java Platform Performance, Oracle, 2011.

[7] K. Hoste, A. Georges, L. Eeckhout. *Automated Just-In-Time Compiler Tuning*. In CGO, 2010.

[8] A. Sewe, M. Mezini, A.Sarimbekov, W. Binder. *Da Capo con Scala – Design and Analysis of a Scala Benchmark Suite for the Java Virtual Machine*. In OOPSLA, 2011.

[9] J. Ortiz. *Manifests: Reified Types*. http://www.scala-blogs.org, 2008

[10] I. Dragos, M. Odersky. *Compiling Generics Through User-Directed Type Specialization*. In ICOOOLPS, 2009.

[11] A. Sarimbekov, P. Moret, W. Binder, A. Sewe, M. Mezini. *Complete and Platform-independent Calling Context Profiling for the Java Virtual Machine*. BYTECODE 2011.

[12] J. Aarniala. *Instrumenting Java bytecode*. University of Helsinki, Finlan, 2010.

[13] S. Haines. *Byte Code Instrumentation Article Series*. http://www.stevenhaines.com, 2010.

[14] W. Binder, J. Hulaas, P. Moret. *Advanced Java Bytecode Instrumentation*. In PPPJ, 2007.

[15] S. Chiba, M. Nishizawa. An Easy-to-Use Toolkit for Efficient Java Bytecode Translators. In GPCE, 2003.

[16] S. Chiba. *Javassist - A Reflection-based Programming Wizard for Java*. In OOPSLA, 1998.

[17] T. A. Proebsting , S. A. Watterson. *Krakatoa: Decompilation in Java (Does Bytecode Reveal Source?).* In COOTS, 1997.

Coding counts LOC, thesis counts page.
In nowadays, it’s how things work.

January 2012