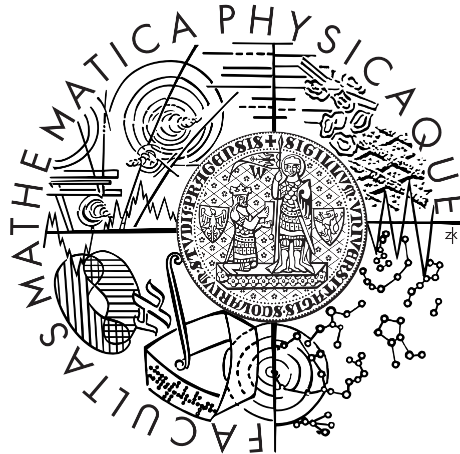
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Jaroslav Kotrč

Run-time Performance Testing in Java

Department of Distributed and Dependable Systems

Supervisor of the master thesis: Vojtěch Horký
Study programme: Informatics
Specialization: Software Systems

Prague 2015

I would like to thank my supervisor Mgr. Vojtěch Horký for his helpful guidance and advice.

I owe my special thanks to those closest to me for their endless patience and moral support.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague date 5.5.2015

Název práce: Testování výkonu za běhu v Javě

Autor: Jaroslav Kotrč

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí diplomové práce: Mgr. Vojtěch Horký, Katedra distribuovaných a spolehlivých systémů

Abstrakt: Práce je zaměřena na relativní porovnávání výkonu jednotlivých metod. Základem je Stochastic Performance Logic, která například umožňuje vyjádřit, že běh jedné metody trvá nejvýše dvakrát déle než běh jiné metody. Tyto výsledky jsou přenositelnější než absolutní hodnoty. Standartní testy jednotek jsou rozšířeny o výkonnostní předpoklady a vyhodnoceny za skutečného běhu reálné aplikace. Instrumentace kódu je dynamicky přidána a odebrána kvůli automatické úpravě produkčního kódu. Pro instrumentaci je použit nástroj DiSL, což umožňuje hladce měřit i systémové třídy Javy. Metody jsou měřeny postupně, počet souběžně měřených metod se dynamicky mění a měřící kód je odstraněn, jakmile jsou získána potřebná data kvůli snížení vlivu měření. Výsledky ukazují, že pro aplikace náročné na procesor lze takto dosáhnout až 3-krát nižšího maximálního okamžitého vlivu měření než při měření všech metod najednou.

Klíčová slova: testování výkonu, testování jednotek, Stochastic Performance Logic, Java

Title: Run-time performance testing in Java

Author: Jaroslav Kotrč

Department: Department of Distributed and Dependable Systems

Supervisor: Mgr. Vojtěch Horký, Department of Distributed and Dependable Systems

Abstract: This work focuses on relative comparisons of individual methods performance. It is based on Stochastic Performance Logic, which allows to express, for example, that one method runs at most two times longer than another method. This results are more portable than absolute values. It extends standard unit tests with performance assumptions, which are evaluated during actual run-time of a released application. Dynamically added and removed instrumentation is used for automatic modification of the production code. Instrumentation part uses DiSL framework to be able to seamlessly measure even Java system classes. Methods are measured sequentially, number of concurrently measured method is dynamically changed and measurement code is removed as soon as required data are obtained to avoid high overhead. The results show that for processor demanding application this approach may bring up to 3-times lower overhead peaks than measuring all methods at once.

Keywords: performance testing, unit testing, Stochastic Performance Logic, Java

Contents

Introduction **3**

1 Analysis **4**
1.1 Measuring process . . . 4
1.1.1 Profiling . . . 4
1.1.2 Instrumentation . . . 4
1.2 Measurement . . . 5
1.2.1 Conditions of measurement . . . 5
1.2.2 Effects on the application . . . 6
1.3 Modification of loaded classes . . . 6
1.3.1 Java agent . . . 7
1.3.2 Native agent . . . 7
1.4 Determining workload size . . . 8
1.5 Measurement control . . . 8
1.5.1 Evaluation . . . 9
1.5.2 Adding measurement . . . 9
1.5.3 Suspending measurement . . . 10
1.5.4 Controller . . . 11

2 Instrumentation technology **12**
2.1 ASM . . . 12
2.1.1 ASM representation . . . 12
2.1.2 Using transformed classes . . . 15
2.2 Javassist . . . 21
2.2.1 Load time modification . . . 21
2.2.2 Static modification . . . 23
2.3 AspectJ . . . 25
2.3.1 AspectJ overview . . . 25
2.3.2 Weaving . . . 26
2.3.3 AspectJ configuration . . . 27
2.4 DiSL . . . 28
2.4.1 DiSL overview . . . 29
2.4.2 DiSL instrumentation process . . . 31
2.5 Technology summary . . . 32

3 Implementation **34**
3.1 Measuring agent . . . 34
3.1.1 Measurement . . . 35
3.1.2 Client listener . . . 36
3.1.3 Native agent . . . 37
3.2 Measuring controller . . . 37
3.2.1 Identification and storage . . . 38
3.2.2 Instrumentation . . . 39
3.2.3 Server listener . . . 40
3.2.4 Scheduling . . . 41

3.2.5 Controlling . . . 42
3.2.6 Size of method workload . . . 44
3.2.7 Evaluation . . . 45
3.3 Configuration . . . 45
3.3.1 Configuration of the agent . . . 46
3.3.2 Configuration of the controller . . . 46
3.3.3 Measuring tasks . . . 50
3.4 Usage . . . 51
3.4.1 Compilation . . . 51
3.4.2 Running . . . 52
3.4.3 Examples . . . 52

4 Evaluation **55**
4.1 Sorting . . . 55
4.1.1 Measurement of sorting . . . 55
4.1.2 Measurement influence . . . 55
4.2 ROME . . . 60
4.2.1 Measured methods . . . 60
4.2.2 Configuration . . . 61
4.2.3 Overhead . . . 61
4.2.4 Results . . . 61

Conclusion **66**

Bibliography **67**

List of Tables **68**

List of Abbreviations **69**

Attachment A **70**

Introduction

Performance of an application is an important property which has great influence to the application success. One method of performance testing is to use Stochastic Performance Logic (SPL) [1].

SPL is a formalism that allows to express expected differences in performance of individual parts of a software. Generally there exist number of tools that capture run time of application methods to express their performance. However, when performance is described in absolute bounds of the execution time, it has limited scope. Running the same code on different machine may lead to different results and the previously measured values became profitless.

SPL expresses performance assumptions as a relative comparisons of run times of two methods. This allows to express, for example, that one method runs at most two-times longer than another one.

There already exist tools for Java which allows performance regression testing based on SPL. This is useful during development time because it allows to monitor performance over the time. The tools use artificial environment to run tests and it has the advantage that the measurement can run without other interferences. However, the real applications never run in this environment, so there may be some differences when the application is practically used.

Goal of this thesis is to explore ways how to monitor application's performance using SPL based assumptions about its methods in a production environment.

The main difference against existing tools is that the measurement interference to the application should be as small as is possible. It will help both to keep the overhead minimal and to obtains more precise data, because if the application is lesser affected, then the impact to the measurement is also lesser.

As there is no control how the application will run and what data it will process, it is also needed to distinguish different data size to compare methods with similar workload. For example, comparing two sorting methods when one sorts ten numbers and other sorts million numbers is unhelpful.

This work is structured into four main chapters. Chapter 1 explores different aspects of the problem and shows possible solutions. Chapter 2 describes number of technologies which can be used to add measuring code to the application. Chapter 3 defines structure of a prototype solution of the problem in Java and its usage is presented on examples in chapter 4.

1. Analysis

There already exist tools to express expected differences in performance of individual parts of a software using SPL. They are designed to be used during application development and are used similar to unit tests independently of other application parts in an artificial environment. This thesis aims to explore ways how to use similar SPL expressions to capture performance assumptions for existing applications on their regular running conditions.

This goal needs to change the attitude towards the process of measurement and evaluation of data, which is described in this chapter. The difference is that measurement should not interference the application to keep the overhead minimal and also to be able to measure data without additional errors.

Because the goal is focused on performance testing in Java, the analysis takes into consideration specific Java features that may be useful for the solution.

1.1 Measuring process

There are two basic ways how to measure application performance. One way is to use profiler and another one is to use custom instrumentation of measured application. This section aims to describe both ways and their difference.

1.1.1 Profiling

Profiling means to collect informations about application performance. One way is to use sampling profiler which probes the target program's program counter at regular intervals using operating system interrupts. Sampling profiles are typically less numerically accurate and specific, so these profilers will be not mentioned further.

Other profilers can use instrumentation or may respond to events from the Java Virtual Machine (JVM) like JPROF [7] to capture data about application execution.

These tools can be used to measure run time of application methods, but they brings additional application slowdown for the whole time the application runs. Another disadvantage is that they do not use any continuous evaluation so the measurement cannot be finalized until the application exits.

Profilers may produce various outputs but it is mainly low level informations which has to be post processed and only small part of it is really needed to evaluate SPL relative comparisons.

1.1.2 Instrumentation

Instrumentation means adding code for measurement start at the beginning and code for measurement finalization to the end of a measured method.

It is more difficult to implement, but it yields significant advantages and can be applied only to classes needed to measure. Code for measurement start and finalization is added only to demanded methods and do not affect other parts of the application.

Generally the instrumentation process can be done before the application starts or it can be postponed until the application is loaded to the JVM.

Instrumentation done in advance of an application start needs to store modified classes until they are loaded to the machine, but then the start-up of the application is not delayed by the instrumentation process.

Instrumentation done as a part of an application loading brings additional time to the start-up, but it does not need to store modified classes, because they are immediately loaded to the JVM. Another advantage is that it is not needed to explicitly run instrumentation and then the modified application as two following processes, so it is more user-friendly.

The main advantage of custom instrumentation is that it allows full control over the measuring process. It is not needed to add measuring code to all methods that should be measured in advance, because Java has a capability to retransform already loaded classes, so the instrumentation process can rerun the similar way like it runs during load time. This brings possibility to add and remove measuring code whenever it is needed.

This way only part of demanded methods can be modified for measurement at once and another methods wait unchanged to be measured after the running measurements finish. When measurement of the method is completed, its class can be retransformed to its original state and another method can be instrumented with measuring code. After all measurements are completed and all modified classes retransformed to their original state, the application continues like it was started without measurement.

1.2 Measurement

The base thing that does not change is to measure time that takes method to complete its run. In the artificial environment it is simple, just start measuring time, run the method and stop whenever the method returns from its call. This cannot be used for application regular run and so its is needed to add code for measuring run time of selected method to the code of an existing application.

1.2.1 Conditions of measurement

Without any assumptions about the measured application it is possible that measured method can be called recursively and even from multiple threads. To work out with this possibilities it is needed to distinguish which thread calls the method and how deep is the recursion.

If this recognition is omitted, then the method measurement can be started by one thread and stopped by another thread which coincidentally calls the same method or method measurement may be started by first call of the method, then the method is called recursively several times and when the deepest recursive call returns it can stop measurement but on another recursion level than the measurement was started.

Both cases result in a measurement of the same method but in an unexpected and unwanted manner which gives erroneous values.

1.2.2 Effects on the application

Since measuring time of method run adds some code to the measured method, it is unavoidable that measurement affects the method execution time. It is possible to organize the code so the start-up instructions are completed before the start time is measured and stop time is measured before finalizing instructions of the measurement are called, so the overhead has as low effect on the measured time as is possible, but it still causes application slowdown

This slowdown is caused not only by the measurement itself, but also by detecting the thread and recursion depth, storing of measured values, evaluating the values and storing results. It is easy to see that the more methods are measured the more significant is the application slowdown.

But there may be no reason to measure all selected methods at once. Furthermore, if one measured method calls another measured method, then the overhead of the called method measurement adds error to the measurement of the caller method. This can be eliminated by measuring methods one by one. It means that in any time at most one method is measured and another methods are waiting in a queue to be measured.

This approach eliminates correlation between two measured methods and yields as low application slowdown as is possible without measurement switch-off.

On the other hand some rarely called method can be selected for measurement and it can take a lot of time before enough data are collected. This may cause that another waiting methods may not be measured at all.

Each application may have its own requirement to the number of methods which should be measured at once. Some applications may want to measure methods one by one, because they are frequently called methods which will cause high application slowdown if they are measured simultaneously and another may want to measure all methods at once because they are not called frequently and their simultaneous measurement does not bring significant application slowdown.

The solution for both cases and any other requirement between them is to use configurable set of measured methods. User can specify number of methods that can be measured simultaneously and the application will control the measurement process and selection of methods that are measured simultaneously.

1.3 Modification of loaded classes

Java provides instruments how to modify classes which are already loaded to the JVM and this can be used to rerun instrumentation process. The modification of already loaded classes is restricted, for example, it must not add new method to the class, but adding and removing measuring code is allowed. There are two ways how the original class can be modified by user created transformer:

Definition

Class definition is made by class loader and it means that array of bytes is converted into an instance of the class *Class* that is then used by the JVM. Redefinition of a class can be made too and that means to replace the definition of a class with a new definition. Transformer can be used to modify class during its definition or redefinition by replacing the definition

of a class without reference to the existing class file bytes, as might be needed in fix-and-continue debugging.

Transformation
Class transformation is made whenever class is initially loaded, redefined or retransformed. Unlike definition which does not need initial class file bytes, transformation starts with the initial bytes and returns modified version of a class. Transformation can be easily used to run instrumentation to add measuring code and when it is needed to remove the code all what is needed to do is to call retransformation of the class without instrumentation. The retransformation process starts with original state of the class and returns it without changes so the measuring code is not added.

In Java there are special system classes, for example, from package *java.lang* which may be difficult to modify, because they are loaded first of all and they can be loaded only by the system class loader.

Retransformation process can be called by agents added to the JVM as a command line arguments. Agents are software components that provide instrumentation capabilities to the application.

1.3.1 Java agent

Java agent is simple a class with method *premain* which takes *Instrumentation* instance from package *java.lang.instrument* as an argument. Agent is packaged as a Java Archive (JAR) file which must include manifest file with *Premain-Class* attribute that specifies the fully qualified name of the agent class containing the premain method. Virtual machine parameter *-javaagent* is then used to add agent to the machine.

Agent's *premain* method is called by the JVM before *main* method of the starting application. It is possible to register a transformer that will be automatically called whenever class definition, redefinition or retransformation is made. This may not work for system classes, before they may be already present in the JVM when the agent registers the transformer.

1.3.2 Native agent

Native agent is a program written in any native language that supports C language calling conventions and C or C++ definitions which use JVM Tool Interface (JVM TI). Agent is platform specific and can be deployed like a dynamic library.

An agent may be started at VM start-up by specifying the agent library name using a command line option *-agentlib*. Another option is to statically link agent with the VM.

The virtual machine starts agent by invoking a start-up function. What function is used depends on the phase when the agent is started and if it is statically linked with the VM. Function *Agent_OnLoad* is used for agents that are not statically linked and are started during the loading phase of the VM which is the most common way.

Pointer to the object used for getting JVM TI environment is passed as an argument of the start-up function. Each agent has its own JVM TI environment and uses it for access to JVM TI functions.

Native agent is started before any bytecode is executed and before any class is loaded, so, unlike Java agent, it can modify even system classes which are loaded first of all. Unlike Java agent which is just a regular class, the native agent does not need the VM to run its code.

During agent's start-up function it should store the environment pointer for later use, set capabilities and register callbacks. Enabled capabilities change which JVM TI functions can be called, what events can be generated, and what functionality these events and functions can provide. Callbacks are functions to be called for each event by the VM.

Capability to retransform classes is needed to be set to enable class retransformation by the agent. Calling class retransformation generates event Class File Load Hook. This event can be received by any agent which set proper capabilities and callbacks. As a response the callback function can transform given class and return its modified version.

1.4 Determining workload size

Because measuring application without artificial environment should not have any effect to the application behaviour, there is no way how to ensure that compared methods will run with similar size of their workload. However, when run times of two methods are compared, only measurements where both methods have similar size of workload should be used.

For example, when comparing two sorting functions they should be compared only when the numbers of sorted values have the same order of magnitude. Otherwise, the comparison of their measurement does not have any significant value.

Often the workload size can be denoted from method arguments, but the instance of method's class may be important too. Example of this case is adding element to a sorted collection. The time depends upon the number of presented elements rather than to the element itself.

The size of method's workload can be represented as a single integer. But there can be more cases where this is not enough. If method works with data with more than one dimension, there may be need to record more than single value. It means that also marking two workloads similar may vary for different applications.

All these cases shows that it is not sufficient to restrict the way how to compute size of the method's workload and aggregate measured times to parts with similar workload. Simple way of sorting measurements by single size of method's workload may be used, but it is also needed to allow user defined computing of workload size and its aggregation for specific cases.

1.5 Measurement control

This section aims to describe controlling the measuring process and evaluation of results.

1.5.1 Evaluation

Student's t-test is used to compare two sets of measurements with similar size of method's workload. This test has some requirements to hold like it requires normality of sample means, which may not hold for measured data. However, it can be solved using moderately large samples. If tens or hundreds of measurements are used, then the t-test generally gives usable results [5].

The larger the sample is the more accurate is the result, but it also extends a period for which is the application affected by measurement. Even when the application is measured for the whole time it is running, there is no guarantee that enough measurements of a method will be made. On the other hand fast and frequently called methods can produce millions of samples in a moment.

It is needed to monitor number of method's measurement to ensure the best results and try to do enough measurements to produce usable result. This can be done by specifying the minimal number of measurements to be collected for similar size of method's workload. The comparison requirement is accomplished when there are enough measurements for both methods of a comparison with the same range of method's workload.

With this approach the overall number of method's measurement can be larger than the number of measurements used for comparison, because the method can be measured with different size of its workload. To limit the number of method measurements, it may be useful to specify maximum number of method's measurement regardless of the size of its workload. It helps to prevent that only some methods will be measured all the time with different workload while other methods wait until they are in order.

With these two limits the method's measurement is done where it accomplishes requirements for all comparisons where it is involved or if it exceeds maximum number of measurement, whatever comes sooner.

1.5.2 Adding measurement

Measurement of a new method can be added when measurement of other method is finished or if there is long delay from the last measurement. Long delay indicates that the measured methods does not run frequently, so it is possible to add new measurement with low risk of suddenly adding high application overhead.

Selecting class

If selecting class of method which measuring will be added, then it may be benefiting to choose class that has not been measured. This may lead to lower correlation between running measurements if we assume that class is independent module of an application. It is more probable that methods in the same class calls one another which brings measuring error if both methods are measured at the same time.

However, this may disadvantage classes where more methods are marked to be measured. Such classes may bring important results for the user so they should be preferred for measurement.

To solve the problem the priority of a class can be computed as the difference between number of methods marked to be measured in the class and number of

methods of the class which measurement has been started. This approach prioritizes classes with higher number of methods to be measured at the beginning of measuring process, but their priority decreases with the number of their methods which measurement has been started, so the number of classes with measured methods will increase over time.

Selecting method

When a class with more than one method to be measured is selected to add new measurement, it is needed to select method of the class which measurement will be started. Method can be prioritized by the number of comparisons where it is involved.

This priority allows to finish some measurements faster, because if there are enough samples for a method, then comparison requirements can be accomplished for more methods compared with this one, so their measurement can be finished.

It has also the natural meaning that the more the method is compared, the more important its performance is for the user and so it should be prioritized for measurement.

When the number of method's comparison is the same, then number of remaining measurements can be used. The method with lower number of remaining measurements can have higher priority which prioritizes methods whose measurement may be finished sooner.

1.5.3 Suspending measurement

Since there is demand for low overhead of the measured application, there is also need to be able to suspend some measurement if the overhead rises more than it is allowed.

This may happen when new measurement of frequently called method is added or when application suddenly rises frequency of calling a measured method.

Storing time between two measurements of any methods can be used to detect such cases. We assume that single measurement adds constant overhead for the measured application regardless which method was measured. If time between two measurements is lower than its limit, then some measurement should be suspended to lower the overhead.

Method with the highest frequency of measurements may be selected to be suspended, because this may lower the overhead more than selecting other measured method. If method runs frequently during last measurement, it is probable that this method will run frequently even after suspension of its measurement and so the application overhead should decline.

If the overhead is still higher than it is allowed even after the suspension of a method, next method may be selected to be suspended from measurement until the application overhead is low enough.

Suspended method can be added for measurement again in the same cases like adding new method for measurement. That is when another method measurement is finished or when delay between measurements is longer than the maximum. Both cases has low chance to add excessive overhead.

1.5.4 Controller

The measuring process and evaluation can be controlled in the same JVM like the measured application, but there is also possibility to move it to another machine. It brings disadvantage of communication overhead, but the advantages are significant.

The controlling process in another JVM is separated from the application machine and so it may realize more expensive operations with less effect to the measured application. Especially running the measuring on a hardware with multiple processor cores or processors may lower the overhead of controller in another JVM.

There can be used just simple agent running on the machine of the measured application and all other components with higher complexity can run on a separated machine. This agent task is only to send measured data to the controller and receive messages to retransform classes.

All other work can be done by the separated controller. It can choose proper method to be measured and send message to the agent to retransform class of the method, so it can be instrumented with measuring code. It will receive measured data and watch the overhead of the measurement to keep it below maximal limit but also ensure that the delay between measurements is not to long. The controller can add or suspend measurement if the overhead is out of the limit and remove measurement of finalized methods. After the method measurement is completed the controller will evaluate comparison and generate results.

Separated controller also allows easy future extension. For example, if there arise a demand for additional evaluation, then it can be easily added to the controller without changes in the agent code.

2. Instrumentation technology

There is number of tools that can be used to manipulate Java code. This chapter describes some of the most known tools with focus on adding measuring procedures to the existing code. Attention is also paid to the ability to modify Java system classes which can be found hard to edit.

2.1 ASM

"ASM is an all purpose Java bytecode manipulation and analysis framework. It can be used to modify existing classes or dynamically generate classes, directly in binary form. (...) ASM offer similar functionality as other bytecode frameworks, but it is focused on simplicity of use and performance." [3]

ASM is not as simple to use as other tools. When adding measuring method is intended, the use of low-level ASM framework is more difficult than using other technology. Moreover other tools use ASM internally and provide its functionality in more user-friendly manner. For example, ASM is used in AspectJ and DiSL described below.

ASM is designed to work with compiled Java classes. This has the advantage that source code is not needed and thus it can be used on applications in binary form. ASM can be used to generate, transform and analyse classes represented as byte arrays as they are stored on disk and loaded in the JVM, but the class loading process is out of its scope.

2.1.1 ASM representation

ASM library provides two ways of class representation. The core application programming interface (API) provides event based representation and the tree API provides object based representation. These kinds of representation can be compared to the representations of Extensible Markup Language (XML) data. The event based API is similar to the Simple API for XML (SAX), while the object based API is similar to Document Object Model (DOM). Regardless of the used representation only one class is managed at the time and independently of the others.

Event based representation

This model allows to represent class as a sequence of events. Each event represents element of the class, for example, its header, field, method declaration or instruction. ASM provides class reader that generates events as it reach them during class parsing and class writer that generates classes based on sequence of events.

This representation is faster and requires less memory than the object based representation, because it is not needed to create and store whole class representation at once. However, some transformation can be more difficult and even may be necessary to use several passes, since only one element of the class is available at any given time. This element corresponds to the currently processed event.

```java
// Create class writer for generating modified class
ClassWriter writer = new ClassWriter(0);
// Create own adapter for instrumentation
// adapter forwards all events to writer
MeasureAdapter adapter = new MeasureAdapter(
        Opcodes.ASM5, writer);
try {
  // Create reader that reads class test.Main
  ClassReader reader = new ClassReader("test.Main");
  // Makes the given adapter visit the Java class of
  // this reader
  reader.accept(adapter, 0);
  // Get the bytecode of the class that was build
  // with writer
  byte[] code = writer.toByteArray();
} catch (IOException e) {}
```

Listing 1: Transformation with ASM event based representation.

The class generation and transformation is based on the abstract class *ClassVisitor*. It has methods for each class file structure section which is used for visiting corresponding section. Sections whose content can be of arbitrary length and complexity are visited by auxiliary visitor class, for example, method is visited by *MethodVisitor* class.

ASM provides three core components based on the *ClassVisitor*.

ClassReader parses a byte array conforming to the Java class file format and calls appropriate visit methods of a given class visitor. It can be seen as an event producer.

ClassWriter generates classes as a byte array conforming to the Java class file format. It can be used alone to generate whole new Java class or to generate a modified class from one or more existing Java classes. It can be seen as an event consumer.

ClassVisitor delegates all the method calls it receives to another *ClassVisitor* instance. Subclasses of this class can overwrite some if this methods to make additional actions during class visiting. For example, when it visits method it can print the method name. It can be seen as an event filter.

Transformation for the purpose of method time measurement can be done with the utilisation of *ClassReader* as an event producer, subclass of *ClassVisitor* that filters events important for measurement and *ClassWriter* to generate modified class as is shown on listing 1 for the measurement of method *void test.Main.run()* and the code of *MeasureAdapter* is on listing 2.

To this point the example does not modify the class, it only prepares the code for the use of own method visitor which adds instructions. To add measuring code it is needed to add events to the method that starts the visit of the method's code and to the method which terminates method's execution. The start of

```java
public class MeasureAdapter extends ClassVisitor{
  // Implicit super constructor ClassVisitor()
  // is undefined for default constructor.
  public MeasureAdapter(){
    super(Opcodes.ASM5);
  }
  // It is needed to overwrite only method visiting
  @Override
  public MethodVisitor visitMethod(
    int access, String name, String desc,
    String signature, String[] exceptions) {
    // Get default method visitor from visitor which
    // this visitor must delegate method calls.
    MethodVisitor mv = cv.visitMethod(access, name,
      desc, signature, exceptions);
    if(name.equals("main")){
      // Use own method adapter for particular method
      // and default adapter otherwise
      mv = new MethodMeasureAdapter(mv, "main");
    }
    return mv;
  }
}
```

Listing 2: MeasureAdapter code for transformation with ASM.

method's code visiting is only one, but the method execution can be terminated by more than one instruction. All these instructions do not have any parameter, so they are all visited by the method for visiting a zero operand instruction, but it is needed to check also the opcode of the instruction to identify only method terminating instructions.

Invocation of methods for measurement start and stop can be added to the picked up points by calling method for visiting a method instruction. That creates new event of the same type. It is also needed to pass string constant to the measuring instructions because they take single string parameter with the name of the measured method. This can be done by calling method that creates event of loading constant on the stack before the measuring method is invoked.

The listing 3 shows method visitor code.

Object based representation

With this model the class is represented as a tree of objects. Each part of the class such as class itself, field, method or instruction is represented as an object that has references to its constituents. Object based representation of class is equivalent to the event based representation of the same class and ASM provides a way how to convert between them.

Some transformation can be simpler when this representation is used, because whole class is available in the memory. The tree representation of a class can

be traversed in any order as it is needed for the transformation, however, it takes more time and consumes more memory for creation and storing of the tree. This representation is useful especially for transformations where several passes are necessary for the event based representation. For example, to add digital signature it is needed first to visit class and compute the digital signature based on the class content and then to do a second visit to actually add the signature. This increases transformation code complexity and requires to store the state between visits which can be as complex as a full tree representation.

The tree representation is based on *ClassNode* class. It extends *ClassVisitor* and for every event it sets fields of the tree representation, so it can be created with the help of *ClassReader*. It also provides inverse operation when for every field of the tree representation it generates events that can be used by *ClassWriter* to generate class as a byte array.

MethodNode class is used for method representation ant method transformation consist of modifying its fields. Example on listing 4 shows how tree representation is created, *void test.Main.run()* method is chosen for transformation and the result class is generated.

Method code is represented as a linked list of instructions. All instructions are represented by the class extended from *AbstractInsnNode*. To transform the method code it is needed to alter this list of instructions. Measuring transformation can be done by adding new instructions to the beginning and before the end of the list. In the example the transformation of the method is done by method *transformMethod* which code is on listing 5.

2.1.2 Using transformed classes

At this point only the transformation process is described, but the transformed classes are not used yet, they are only obtained as byte arrays. The way of using generated classes depends on the context and is out of scope of the ASM. One way is to load them dynamically with a class loader. However, class transformation done inside a class loader can only transform classes loaded by this class loader.

If the modification of other classes is needed, then the transformation can be done inside a *ClassFileTransformer* from package *java.lang.instrument*. This is used by Java agents to transform classes before they are defined by the JVM.

Java agent is a class with method *public static void premain(String agentArgument, Instrumentation instrumentation)* which is called when the JVM is launched in a way that indicates an agent class. In that case an *Instrumentation* instance is passed to the premain method of the agent class. *ClassFileTransformer* can be registered to the instrumentation instance and is used to transform any class loaded after the transformer was added.

Listing 6 shows code of such premain method for transforming *test.Main* class.

This example uses the same *MeasureAdapter* as is used in the example of transformation with event based representation. To create a Java agent it is needed to create JAR file containing the agent class and manifest with *Premain-Class* attribute set to the class with premain method. The agent is used with *-javaagent:agent.jar* JVM argument where *agent.jar* is path to the agent JAR file.

Java agent can transform most of the classes, but there are some exceptions

which may be not instrumented this way. For example, *java.lang.Integer* class may be not transformed before it is loaded to the JVM. Some classes can be loaded to the JVM before agent starts, so they can be omitted.

Class can be also transformed and stored to the local disk. It can be done by storing byte array of the transformed class as a class file on the local disk. This is applicable for every class including any Java system class. To use this class it is needed to prepend it in front of the bootstrap class path, so the system class loader would find them before it reaches the original class files thus only the modified classes are used.

```java
public class MethodMeasureAdapter extends MethodVisitor{
  // Name of the measured method
  private String name;
  //Create new adapter for method with given name.
  public MethodMeasureAdapter(
    MethodVisitor mv, String name) {
    super(Opcodes.ASM5, mv);
    this.name = name;
  }
  // Override method that starts the visit of
  // the method's code
  @Override
  public void visitCode() {
    // Call visiting on the visitor to which this
    // visitor must delegate method calls.
    mv.visitCode();
    // Load constant with method name on the stack
    mv.visitLdcInsn(name);
    // Add instruction invoking static method
    // DataStore.startMeasurement. It starts measurement
    // and has name of measured method as a parameter
    mv.visitMethodInsn(Opcodes.INVOKESTATIC, "DataStore",
      "startMeasurement", "(Ljava/lang/String;)V",
      false);
  }
  // Override visiting a zero operand instruction
  @Override
  public void visitInsn(int opcode) {
    // Check if the instruction terminates method's
    // execution
    if ((opcode >= Opcodes.IRETURN
      && opcode <= Opcodes.RETURN)
      || opcode == Opcodes.ATHROW) {
      // Load constant with method name on the stack
      mv.visitLdcInsn(name);
      // Add instruction invoking static method
      // DataStore.stopMeasurement
      mv.visitMethodInsn(Opcodes.INVOKESTATIC,
        "DataStore", "stopMeasurement",
        "(Ljava/lang/String;)V", false);
    }
    // Delegate visitig when it is done
    mv.visitInsn(opcode);
  }
}
```

Listing 3: Method visitor code for transformation with ASM.

```java
try {
  // Create reader for given class
  ClassReader reader = new ClassReader("test.Main");
  // Create blank node
  ClassNode node = new ClassNode();
  // Makes node visit class of the reader
  reader.accept(node, 0);
  // Iterate over methods of the class
  List<MethodNode> methods = node.methods;
  for(MethodNode method : methods){
    // Choose method "run" and transform it
    if(method.name.equals("run")){
      transformMethod(method);
    }
  }
  // Create writer for generating modified class
  ClassWriter writer = new ClassWriter(0);
  // Makes writer visit tree representation
  node.accept(writer);
  // Get the bytecode of the class
  byte[] b = writer.toByteArray();
} catch (IOException e) { }
```

Listing 4: Transformation with ASM tree representation.

```java
private static void transformMethod(MethodNode method) {
  // Get list of method instructions
  InsnList insns = method.instructions;
  // Iterate over the list to find instruction which
  // terminates execution
  Iterator<AbstractInsnNode> it = insns.iterator();
  while (it.hasNext()) {
    AbstractInsnNode in = it.next();
    int op = in.getOpcode();
    if ((op >= Opcodes.IRETURN && op <= Opcodes.RETURN)
      || op == Opcodes.ATHROW) {
      // Create temporary list of instructions that stops
      // measurement
      InsnList stop = new InsnList();
      // Add instruction to load constant with method
      // name on the stack
      stop.add(new LdcInsnNode(method.name));
      // Add instruction invoking static method
      // DataStore.startMeasurement
      stop.add(new MethodInsnNode(Opcodes.INVOKESTATIC,
        "DataStore", "stopMeasurement",
        "(Ljava/lang/String;)V", false));
      // Insert the temporary list before the terminating
      // instruction
      insns.insert(in.getPrevious(), stop);
    }
  }
  // Create temporary list of instructions that starts
  // measurement
  InsnList start = new InsnList();
  // Add instruction to load constant with method name
  // on the stack
  start.add(new LdcInsnNode(methodName));
  // Add instruction invoking static method
  // DataStore.startMeasurement
  start.add(new MethodInsnNode(Opcodes.INVOKESTATIC,
    "DataStore", "startMeasurement",
    "(Ljava/lang/String;)V", false));
  // Insert the temporary list at the beginning
  insns.insert(start);
}
```

Listing 5: Transformation method code for transformation with ASM.

```java
public static void premain(String agentArgument,
  Instrumentation instrumentation){
  instrumentation.addTransformer(
    new ClassFileTransformer() {
    // Overwrite method for transforming classes
    public byte[] transform(ClassLoader l, String name,
      Class<?> c, ProtectionDomain d, byte[] b)
      throws IllegalClassFormatException {
      // Transform only "test.Main" class
      if(name.equals("test.Main")){
        // Create reader which uses provided byte array
        // as an input
        ClassReader cr = new ClassReader(b);
        // Create writer to generate modified class
        ClassWriter cw = new ClassWriter(cr, 0);
        // Create class to filter events and add
        // measurement code
        ClassVisitor cv = new MeasureAdapter(cw);
        // Make the filter to visit class of the reader
        // It also delegates the visits to the writer
        cr.accept(cv, 0);
        // Return modified class as an array
        return cw.toByteArray();
      }
      // It is not required class, no transformation is
      // needed
      return null;
    }
  });
}
```

Listing 6: Premain method code.

2.2 Javassist

"Javassist (Java Programming Assistant) (...) enables Java programs to define a new class at runtime and to modify a class file when the JVM loads it." [2]

Javassist provides two levels of API:

- Source level allows editing a class file without knowledge of the specifications of the Java. User can even specify inserted bytecode in the form of source text and Javassist compiles it on the fly. This is useful for adding measuring code just as plain source text without a precedent compilation.
- Bytecode level allows to directly edit a class file.

The instrumentation can be made during the application load time or statically before it starts.

2.2.1 Load time modification

The simple way to modify class is during its load time. That can be used for most common cases. However it does not support modification of Java system classes like *java.lang.String*, because it cannot be loaded by a class loader other than the system class loader.

Load on demand

First approach to modify class is to obtain the class representation on demand before the application is started. Javassist reads a class file from the source and returns a reference to the *javassist.CtClass* object representing that class file. Then instrumentation can be done using this object and at the end the class is converted to a *java.lang.Class* instance. After this conversion further modifications are not allowed any more.

When all classes are modified according to the required measurements, the application is started in the same JVM. During its run these modified classes are used because they have already been loaded in the machine.

This way of instrumentation depends on the fact that the modified classes are never loaded in the machine before their representation is converted to a *java.lang.Class* instance. If not, the JVM would load the original classes before the modified classes are converted, so the conversion would fail since the class loader cannot load two different versions of the same class at the same time.

This fact is not a problem for common cases, because the modification runs before the application itself, so the application classes should not be present in the JVM. But it cannot be used in general. Different class loaders can solve some issues, however, Java system classes can be loaded only by the system class loader. That is why they cannot be modified this way.

Listing 7 shows simple code of Javassist utilisation to add measuring code for method *void test.Main.run()*.

```
// Get default class pool
ClassPool pool = ClassPool.getDefault();
try {
  // Get class of the method
  CtClass cc = pool.get("test.Main");
  // Get method with specific name and descriptor
  CtMethod cmethod = cc.getMethod("run", "()V");
  // Insert code that starts measurement
  cmethod.insertBefore(
    "DataStore.startMeasurement(\"test.Main#run\");");
  // Insert code that stops measurement
  cmethod.insertAfter(
    "DataStore.stopMeasurement(\"test.Main#run\");");
  // Convert instrumented object to the Class instance
  cc.toClass();
} catch (Exception e){}

// Instrumentation is done
// The measured application can be started now
test.Main.main(args);
```

Listing 7: Javassist usage.

Translator

Second way how to modify class during load time is to use a translator, which is called whenever a class is loaded. The translator can decide whether to translate the class or not by the class name.

To use it it is needed to implement interface *javassist.Translator* provided by Javassist. This implementation can be added as an event listener to the class loader *javassist.Loader* which is also part of the Javassist.

This loader can be used for loading and running a particular class, for example, the main class of the measured application. The added event listener is notified when the class loader loads the main class and any other application classes and can modify them before they are converted to *java.lang.Class* instance and used by the application.

However, either using the translator does not work for Java system classes, because they can be loaded only by the system class loader.

The difference between this two ways is that when using load on demand all classes, that should be modified, are modified at once before the measured application starts. On the other side using the translator postpones the loading and modification of the particular class to the time, when the application first needs it.

Translator usage is shown on listing 8 and on the listing 9 is the code of user specific translator *MyTranslator* for the same method *void test.Main.run()*. The instrumentation part is the same as in the load on demand example without converting to Class instance.

```
// Create provided javassist Loader
Loader cl = new Loader();
// Create user specific translator
Translator t = new MyTranslator();
// Get default class pool
ClassPool pool = ClassPool.getDefault();
try {
  // Add translator
  cl.addTranslator(pool, t);
  // Run the application with provided loader
  cl.run("test.Main", args);
} catch (Exception e) {}
```

Listing 8: Javassist translator usage.

2.2.2 Static modification

The main problem, why Java system classes cannot be modified, is that they are already present in the JVM when the instrumentation runs and they can be only loaded by the system class loader. To modify them it is needed to use static modification.

The process is similar to the load on demand approach. First it is needed to obtain class representation and modify it. The only difference is that after modification the representation is not converted to a *java.lang.Class* instance, but it is written to a class file matching the representation on the local disk. The difference in code is on listing 10.

To run the application it is needed to prepend this modified class files in front of the bootstrap class path. System class loader would find these files before it reaches the original class files thus original classes are not used and the application should use only the modified classes.

This approach finally allows to modify system classes at the expense of two runs. First run instruments the classes and creates new class files and second run starts the application with the modified classes.

```java
public class MyTranslator implements Translator {
  // It is invoked when the object is attached to
  // the Loader object. Not needed for the example.
  @Override
  public void start(ClassPool pool)
    throws NotFoundException, CannotCompileException {}

  // It is invoked by a Loader for notifying that
  // a class is loaded.
  @Override
  public void onLoad(ClassPool pool, String classname)
    throws NotFoundException, CannotCompileException {
    // Check if the loaded class is the demanded one
    if("test.Main".equals(classname)){
      // Get the class from class pool
      CtClass cc = pool.get("test.Main");
      // Get method with specific name and descriptor
      CtMethod cmethod = cc.getMethod("run", "()V");
      // Insert code that starts measurement
      cmethod.insertBefore(
        "DataStore.startMeasurement(\"test.Main#run\");"
        );
      // Insert code that stops measurement
      cmethod.insertAfter(
        "DataStore.stopMeasurement(\"test.Main#run\");");
    }
    // Not needed to convert to Class instance
    // since Loader do it after this method returns.
  }
}
```

Listing 9: Javassist translator code.

```java
// Instead of a converting to the Class instance by
// cc.toClass();
// write it to local file
cc.writeFile("outDir");
```

Listing 10: Writing class to the file with Javassist.

2.3 AspectJ

Aspect-oriented programming is a style of computer programming that aims to increase modularity by allowing the separation of cross-cutting concerns. These concerns are parts of a program that rely on or must affect many other parts of the system. That would be any kind of code that is repeated in different methods and cannot normally be completely refactored into its own module.

Measuring method time can be considered as a kind of cross-cutting concern. The code responsible for providing measurement of the method time has to be normally inserted into each measured method. Using aspects it allows to separate this code in its own module and appropriate code is automatically inserted on the right place into all required methods.

2.3.1 AspectJ overview

"[AspectJ is] a seamless aspect-oriented extension to the Java programming language (...) [it enables] clean modularization of crosscutting concerns, such as error checking and handling, synchronization, context-sensitive behavior, performance optimizations, monitoring and logging, debugging support, and multi-object protocols." [4]

AspectJ is well known and easy to learn and use. It is also supported in some integrated development environments, for example, Juno version of Eclipse.

AspectJ uses some specific concepts:

Join point is a well-defined point in the program flow. For example, they consist of things like method calls, method executions, object instantiations, constructor executions, field references or handler executions. The join point model provides the common frame of reference that makes it possible to define the dynamic structure of crosscutting concerns.

Pointcut picks out certain join points in the program flow. A pointcut can be also built out of other pointcuts with logical conjunction, disjunction or negation.

Advice brings together a pointcut to pick out join points and a body of code to run at each of those join points. There are several kinds of advice

- **before** advice runs as a join point is reached, before the program proceeds with the join point.
- **after** advice runs after the program proceeds with the join point.
- **around** advice on a join point runs as the join point is reached and has explicit control over whether the program proceeds with the join point.

Inter-type declarations are declarations that may declare members that cut across multiple classes, or change the inheritance relationship between classes. This declarations operates statically at compile-time.

Aspect wraps up pointcuts, advice, and inter-type declarations into modules. It is defined like a class and can have methods, fields, and initializers in addition to the crosscutting members.

```
// Aspect wrapping crosscutting members
public privileged aspect Measure {
  // Define pointcut for desired method
  pointcut main() : execution(* test.Main.run(..));
  // Define advice that starts measurement
  before() : main(){
    DataStore.startMeasurement();
  }
  // Define advice that stops measurement
  after() : main(){
    DataStore.stopMeasurement();
  }
}
```

Listing 11: Aspect definition for AspectJ.

The purpose of method run time measurement can be achieved by the usage of pointcut which picks out particular method execution. The measurement can be implemented by *before* advice which starts time measurement and *after* advice which stops measurement and stores the result. The definition of the pointcut and advices can be wrapped up in a single aspect outside of the measured application. Listing 11 shows aspect for method *void test.Main.run()* measurement.

2.3.2 Weaving

Because aspects are written in a source code which is not compatible with pure Java code, it cannot be directly used in a program compiled with Java compiler. It is needed to create source code of the aspects first and then use *ajc*. This tool compiles the aspect and uses it to modify other classes. It can be run from command line or as an Ant task and is implemented completely in Java. The modification is called *weaving*.

The weaving process can take place at one of three different times:

Compile-time weaving

It is the simplest approach which can be used when the application source code is available. The source code is not compiled before it is weaved. The *ajc* compiles from source and produce woven class files as output. Aspects used for the weaving may be in source or compiled to binary form in advance. This is useful mainly for measuring application during development time. However, if the affected classes need aspects to compile, then compile-time weaving has to be used. For example, this happens when aspect adds some members to a class and other classes references the added members.

Post-compile (binary) weaving

It is used to weave existing class files and JAR files. Aspects can also be in source or binary form. This is very universal because it can be used to instrument existing application with measuring code. It can be also used to measure java system classes like *java.lang.Integer*. In this case the *rt.jar*

file which contains java system classes is used as input for weaving and AspectJ weaver produces new JAR file of the woven classes as output. To run the application it is needed to prepend the weaved JAR file in front of the bootstrap class path and the application will use the modified system classes.

Load-time weaving
It is binary weaving deferred until the point that a class loader loads a class file and defines the class to the JVM. To use it the *aspectjweaver.jar* library must be added to the class path. This approach does not produce any class files unlike the other two methods but uses the woven classes only for a single application run. Aspects used for weaving can be only in binary form. All load-time weaving is done in the context of a class loader and hence the set of aspects used for weaving and the types that can be woven are affected by the class loader delegation model. It implies that some classes like Java system classes in *java.** package and sub packages are not exposed to the weaving infrastructure and cannot be instrumented this way.

Regardless of the weaving process time the aspect code is still the same and the difference is only in the way how the code is weaved and started. To distinguish between sources *ajc* uses different arguments. For example *sourceRoots* argument is used for java or aspect source files, *aspectpath* argument for binary aspects and *inpath* for compiled class files.

For binary and load-time weaving the application class path has to include *aspectjrt.jar* library and then it can be started as any other application. To enable load-time weaving it is needed to add *aspectjweaver.jar* library to the class path. The mechanism is chosen through JVM start-up options and can be used by specifying the *-javaagent:pathto/aspectjweaver.jar* option to the JVM. The provided java agent then take care for the weaving process.

2.3.3 AspectJ configuration

The configuration of aspects to measure particular methods includes to create new aspect and specify the measured methods. This can be done by generating source code of the aspect. The other way is to use load-time weaving and configure the weaver with one or more *META-INF/aop.xml* files located on the class loader search path. Each file may declare a list of aspects to be used for weaving, type patterns describing which types should be woven, and a set of options to be passed to the weaver.

Aspects in the configuration file can be defined in two ways:

- Extending abstract aspect visible to the weaver. The abstract aspect includes an abstract pointcut and advice handling the measurement of method time attached to the abstract pointcut. It is needed to compile this aspect and concretize the abstract pointcut in the configuration XML file to include desired method. On the listing 12 is code of such abstract aspect and listing 13 shows how to concretize it for method *void test.Main.run()* in an XML file.

```
public abstract aspect AbstractMeasure {
  // Define abstract pointcut
  abstract pointcut measuredMethod();
  // Use abstract pointcut in advices
  before() : measuredMethod(){
    DataStore.startMeasurement();
  }
  after() : measuredMethod(){
    DataStore.stopMeasurement();
  }
}
```

Listing 12: Abstract aspect for AspectJ.

```
<aspectj>
  <aspects>
    <!-- Concrete aspect derived from abstract aspect.-->
    <concrete-aspect name="test._Measure0"
      extends="test.AbstractMeasure">
      <!-- Concretize abstract pointcut. -->
      <pointcut name="measuredMethod"
        expression="execution(* test.Main.run(..))"/>
    </concrete-aspect>
  </aspects>
</aspectj>
```

Listing 13: Concretizing abstract aspect in XML for AspectJ.

- Since AspectJ version 1.6.12 the concrete aspect can be written whole in the *aop.xml* file without need for the abstract aspect. This means that measurement can be done without single aspect source code so there is no need to use other than Java compiler because the weaving is done by weaver in the AspectJ *aspectjweaver.jar* library during load-time. Listing 14 shows how to write whole aspect in an XML file for method *void test.Main.run()*.

Since this configuration is simpler it cannot be used for system classes measurement because the load-time weaving relies on class loaders and system classes can be loaded just by system class loader. The only way to measure system classes is to use post-compile weaving to create modified classes which are used by the application afterwards.

2.4 DiSL

"DiSL is a new domain-specific language and framework for Java bytecode instrumentation. DiSL is inspired by AOP [aspect-oriented programming], but in contrast to mainstream AOP languages, it features an open join point model where any region of bytecodes can be selected as a join point (i.e., code location

```
<aspectj>
  <aspects>
    <!-- Concrete aspect using measuring methods. -->
    <concrete-aspect name="_Measure1">
      <before pointcut="execution(* test.Main.run(..))"
        invokeClass="test.DataStore"
        invokeMethod="startMeasurement(JoinPoint tjp)"
      />
      <after pointcut="execution(* test.Main.run(..))"
        invokeClass="test.DataStore"
        invokeMethod="stopMeasurement(JoinPoint tjp)"
      />
    </concrete-aspect>
  </aspects>
</aspectj>
```

Listing 14: Aspect written only in XML for AspectJ.

to be instrumented). (...) Thanks to the pointcut/advice model adopted by DiSL, instrumentations are similarly compact as aspects written in AspectJ. However, in contrast to AspectJ, DiSL does not restrict the code locations that can be instrumented, and the code generated by DiSL avoids expensive operations (such as object allocations that are not visible to the programmer). Furthermore, DiSL supports instrumentations with complete bytecode coverage out-of-the-box and tries to avoid structural modifications of classes that would be visible through reflection and could break the instrumented code." [9]

Because DiSL uses some native code the current version 2.0.1 can be used only for Unix system.

2.4.1 DiSL overview

As is described in [8] each instrumentation is defined through methods declared in standard Java classes so no additional compiler is needed to compile it into binary form in contrary to AspectJ advices. These methods are called *snippets* and are annotated to specify the join points where the code of the snippet shall be inlined. These annotations are also standard parts of Java language.

DiSL snippets can be inlined before or after each marked region like advices in AspectJ. However it lacks the usual form of the around advice. The type of inlining is distinguished by the name of the annotation. Annotation *@Before* is used to inline the snippet before each marked bytecode region and annotation *@After* places the snippet after exit of each marked region.

The regions, where the snippets shall be inlined, are specified with required parameter of the annotation called *marker*. The annotations have several optional parameters for more detailed configuration of the snippet scope, order and others.

In other aspect oriented programming languages the data can be exchanged between advices using local variable within around advice. This kind of advice gives the ability where the advice code can decide whether to skip or proceed with

```
// Code that starts measurement
@Before(marker = BodyMarker.class, scope = "Main.run")
public static void start(MethodStaticContext msc) {
  DataStore.startMeasurement(msc.thisMethodFullName());
}
// Code that stops measurement
@After(marker = BodyMarker.class, scope = "Main.run")
public static void stop(MethodStaticContext msc) {
  DataStore.stopMeasurement(msc.thisMethodFullName());
}
```

Listing 15: DiSL snippet for measurement.

the method invocation and thus it can change the instrumented code behaviour and break it.

Instrumentation code inserted by DiSL is intended only for application observation and it should never alter control flow of the observed application. To preserve the ability to share data without around advices DiSL uses *synthetic local variables*. They are static fields annotated as *@SyntheticLocal*. These are translated into local variables which have the scope of a method invocation and can be accessed by all snippets that are inlined in the method.

Because single snippet can be used for multiple methods measurement, there is need to distinguish between different methods. *Static context interfaces* can be used to obtain information about the instrumented class, method and bytecode region. They provide information that is already available at the instrumentation time. Interfaces can be accessed as a parameter of the snippet and DiSL replaces the calls to these interfaces with corresponding static context information which improves the efficiency.

The desired goal to measure method time can be achieved by creating two snippets. First snippet annotated by *@Before* annotation starts the measurement and second snippet annotated by *@After* annotation stops the measurement and stores the result. Both snippets use marker that identifies the method body as a join point where to inline code of the snippets. Optional annotation argument *scope* can be used to identify scope of methods, where the snippet is applied. Each snippet can have *MethodStaticContext* parameter and use it to obtain method name to identify different measured methods. Listing 15 shows the code of such snippets for method *Main.run()*.

This solution can be used for predefined set of measured methods. To change these methods there is need to create new snippet with different scope. However, DiSL allows to restrict the instrumentation scope using the *guard* construct. Guard is a user-defined class with a single guard method. This method determines whether a snippet matching a particular join point is inlined. The guard method can have parameters such as static context interface from which it can obtain information about the join point. With this information it can decide whether to realize the modification or not dynamically during instrumentation process.

With this approach the method measurement snippets can be altered not to

```
// Code that starts measurement
@Before(marker = BodyMarker.class,
  guard = MeasurementGuard.class)
public static void start(MethodStaticContext msc) {
  DataStore.startMeasurement(msc.thisMethodFullName());
}
// Code that stops measurement
@After(marker = BodyMarker.class,
  guard = MeasurementGuard.class)
public static void stop(MethodStaticContext msc) {
  DataStore.stopMeasurement(msc.thisMethodFullName());
}
```

Listing 16: DiSL snippet with guard for measurement.

```
// Guard method with proper annotation
@GuardMethod
public static boolean measuredMethod(
  MethodStaticContext msc) {
  String name = msc.thisMethodFullName();
  // Return true if method should be instrumented
  return "test/Main.run".equals(name);
}
```

Listing 17: DiSL guard method code.

restrain scope statically, but instead they can use annotation argument *guard* to identify custom guard. This guard can be the same for both snippets and its guard method annotated as *@GuardMethod* may have *MethodStaticContext* parameter to obtain method name like the snippet itself. Without restrained scope this guard method is called for every join point that matches marker of the snippet. For these snippets it means that the guard method decides if the method will be instrumented or not. The decision is made for each single method of the instrumented application separately and is based on name of the method which is examined. Example of guard usage is on listing 16 and listing 17 shows guard method code for method *void test.Main.run()*.

2.4.2 DiSL instrumentation process

One of the greatest difference between DiSL and other described technologies is the instrumentation process. DiSL uses two separate virtual machines to performs bytecode instrumentation. The purpose is to minimize perturbation in the observed program. In this way, class loading and initialization triggered by the instrumentation framework do not happen within the observed process.

It uses client-server architecture. Client side is a machine where the instrumented application runs and server side is a machine where the instrumentation is processed. The instrumentation starts on the client when the class is loading.

Native JVM TI agent captures all class loading events in the client JVM and sends every class as a byte array through a socket to the server JVM. On the server machine the class may be instrumented and it is send back to the client where it is linked so the instrumented code is used instead of the original one.

The code of native agent is the purpose why the current DiSL version 2.0.1 runs on Unix systems and is not as portable as other pure Java applications.

On the other hand it yields the advantage of easy instrumentation of every class. Because other instrumentation technologies depends mainly on class loaders, they cannot directly instrument system classes which can be loaded only by system class loader. The native agent can capture loading event of every class even the loading of *java.lang.Object*. Thus, the system classes can be measured using DiSL instrumentation without any additional effort just like every other class.

2.5 Technology summary

Every mentioned technology can be used for method run time measurement for every class. However, most of them requires two runs for special classes like Java system classes. First run generates modified classes and the second run uses them. The only exception is DiSL which is capable to transform every class on the first run.

ASM is powerful and simple but it works on the bytecode level, so it is not so easy to use it correctly. Javassist provides higher level of abstraction and easier way of instrumentation but it also needs two runs for instrumenting system classes. AspectJ is well known and simple to use, but its configuration is more difficult and it requires generation of either aspect or XML configuration file and for Java system classes it is also needed to use two runs to use the modified classes.

DiSL provides simplicity of usage like AspectJ and easy configuration and high level of abstraction like Javassist. It is also capable to instrument any class on the first run regardless if it is system class or just regular user created class due to its instrumentation process, which uses native agent and is divided into two JVMs. This can make retransformation of classes much easier, because all classes can be transformed during load time and calling retransformation of a class will just rerun the instrumentation.

Also this separated instrumentation process of DiSL leads to simple usage of separated controller of the measuring process which is described in section 1.5.4. When there is already another running JVM that communicates with the machine of the measured application, it is easy to add the controller to run on the instrumentation machine and use just simple agent for the machine of the measured application.

Because the instrumentation is done on the same machine where the controller runs, it is able to choose whether the method of a retransformed class should be instrumented or not using just controller. After it sends message to the agent to call class retransformation, the decision can be made without any additional dependency on the agent, which brings simplification of the communication between agent and controller.

Table 2.1 shows summary of the mentioned technologies.

Technology	Instrumentation Level	Configuration	System classes
ASM	bytecode	string in the code	two runs
Javassist	java code	string in the code	two runs
AspectJ	java code	generated file	two runs
DiSL	java code	string in the code	single run

Table 2.1: Technology overview.

3. Implementation

Part of this thesis is a prototype demonstrating described approach. This section focuses on implementation details of the prototype and explains its functions.

DiSL was selected as a technology for instrumentation. Its advantages are described in section 2.5.

Used technology allows to separate application into two functional parts like it is described in section 1.5.4. First part is called measuring agent and second part is called measuring controller.

However, the real structure of the application is a little more complex and each functional part consist of more modules. This is described in sections oriented for each functional part in details. The exception is core part used in both functional parts which is described apart in both functional parts depending on its utilisation context.

For example, measurement starting is used on the agent but the controller adds the code to start measurement to the code of measured method. Both agent and controller use measurement starting but it is interesting only from the agent point of view so it is described in the agent section.

3.1 Measuring agent

The part which runs on the same JVM like measured application is named measuring agent. Its task is to run measured application and service measuring infrastructure. During measurement it receives messages from server to retransform selected class and it sends measured values to the server.

The agent part itself does not use any third party libraries to minimize dependencies added to the measured application. The only exception is *SizeOf* class from package *net.sourceforge.sizeof* used for computing size of method arguments. This is a third party code but used in a form of source code and so it is contained in the compiled core part.

Agent is separated to three parts:

core
: This part is used to realize measurement as is described in section 3.1.1 and communication which is described in section 3.1.2. It is placed in *core.jar* and to run measured application it is needed to append this JAR to the bootstrap class path because its classes are used for measurement. Also it is needed to add this JAR as a Java agent to the JVM because *SizeOf* uses instrumentation API to compute size of an object. The same way must be added *disl-agent.jar* library of the DiSL engine to the bootstrap class path and as a Java agent.

launcher
: It serves for launching the measured application. During the launching it initialize configuration of the agent and starts thread for communication. After application terminates it interrupts communication thread, joint it and exits. It is packaged in *launcher.jar* and it is needed to add it to the class path of the measured application.

c-agent

This part contains native agent described in section 3.1.3. It serves for invocation of a class retransformation and is compiled to *libreloadagent.so* (or *libreloadagent.jnilib* for Darwin operating system). To run the measured application it is needed to add it as a native agent to the JVM. The library *libdislagent.so* (or *libdislagent.jnilib*) of the DiSL engine must be added as a native agent too.

3.1.1 Measurement

This part is packaged in *core.jar* and services measurement. Infrastructure for the measurement is in package *cz.cuni.mff.spl.instrumentation.runtime*.

Measurement start

Measurement is started by static method *startMeasurement* of *DataStore* class. As an input arguments it needs name for the measurement and optionally size of a method input. This class stores all started measurement in a hash table where the name of the measurement is used as a key. The key is unique for every measured method and contains full method name including is class and descriptor. The key allows access to the instance of class *DataHolder* which is dedicated for the method. Start of the measurement is then propagated to appropriate instance of *DataHolder* obtained from the hash table.

DataHolder stores starts of measurements for a single method. And as is described in section 1.2.1 it is also needed to distinguish which thread calls the measured method and the depth of recursion if the method calls itself. All this is detected at the moment of a measurement start by this class.

DataHolder uses identification of current thread as a key to another hash table where starts of measurements for every thread are stored in a stack. New start of a measurement for the same method called by the same thread is added and finished measurement is removed from the stack. Thus the size of the stack can be simple used as a depth of the method recursion, because it represents call stack of a single method by a single thread.

When the right place is picked for a new measurement start, the *Measurement* instance is created to store data of measurement and is added to the stack. Obtaining the current time and store it in the *Measurement* instance are the last instructions of the process of measurement start. That allows to use the most accurate time before the measured method starts its execution

Measurement stop

Measurement is finished in a similar manner like it is started. To stop measurement it is called static method *stopMeasurement* of *DataStore* class with measurement name as an argument. The first instruction is to obtain current time, so it is used time as close to the measured method execution end as it is possible.

Then it is used name of the measurement to get appropriate *DataHolder* which uses thread identification to get appropriate stack of started measurements. The top *Measurement* instance from this stack is removed and the time of measurement end is set to the it.

Finished measurement is then converted to its string representation and added to the queue of messages waiting to be send to the measuring controller. Since this time the *Measurement* instance is no longer present in the measuring infrastructure and can be collected by the garbage collector so it no longer takes any memory of the application.

Since *Measurement* instances are sent as a string representation, it is needed to be able to create new instance when the message is received by the controller. To do this the *Measurement* class has constructor which parses the string representation and instantiates new measurement with appropriate values.

3.1.2 Client listener

Communication services of the measuring agent provides packages placed in *core.jar*:

cz.cuni.mff.spl.instrumentation.communication

It contains the common part of communication used by both measuring agent and measuring controller. Class *DataProcessor* is used to read string message from *ByteBuffer* instance. The message must have its length in bytes as integer in first 4 bytes following by text of the message. It is prepared to receive incomplete message. In this case the message is stored in another buffer and completed after more data are received.

This is used by class *DataListener* which provides basic communication through channels in a new thread. It is specialized for both measuring parts apart. New message to send is added by *sendMessage* method. The messages are stored in a queue and are waiting until the communication thread removes them from the queue and sends them. This way allows other threads to add messages to be sent without waiting until the sending routine end.

Waiting messages are also stored in a hash set to prevent sending the same message twice in a single communication cycle. When new message is added it is checked, that the same message is not already present. If it is present, then new one is not added and the message is sent only once.

cz.cuni.mff.spl.reloading

This package contains *ClientDataListener* class which is a specialization of *DataListener*. The difference is that during initialization the channel is placed in a non-blocking mode. It means that the thread will not block when waiting for message from controller and can immediately return to be able to send measured values to the controller.

Another specialization is when new message is received. It means that controller sends command to retransform the class which name is contained in the message. In this case the listener calls static method *retransform* of *NativeMethod* class from this package.

This method does nothing in the Java code because it is just a Java API for native method implemented by the native agent part.

3.1.3 Native agent

Native agent provides just a single but very important service of a class retransformation. Implementation of this service was the most troublesome part of the application development.

At first try there was intended to use Java agent instead of native agent, because native agent is more complicated and it is not platform independent. However using Java agent to call retransformation method on the instance of *Instrumentation* from package *java.lang.instrument* does not lead to rerun the instrumentation and old version of the class was still used.

It was needed to examine how DiSL uses its native agent to obtain class for instrumentation. It appears that it registers callback for *ClassFileLoadHook* event which is called whenever the JVM obtains class file data so it works during class loading, but calling class retransformation from a Java agent does not trigger it.

After examination of JVM TI manual there was founded part: "This event is also sent when the class is being modified by the RetransformClasses function or the RedefineClasses function, called in any JVM TI environment." [10]

To be able to call class retransformation in a JVM TI environment it was needed to use the native agent which is able to get the environment during its start-up function as is described in section 1.3.2.

So the native agent was implemented and class retransformation was called in a JVM TI environment but the instrumentation still does not rerun. The last catch was the native DiSL agent. It was able to transform class when *ClassFileLoadHook* event occurs, but when it is triggered by *RetransformClasses* method this event is sent only to retransformation capable environments. To make the DiSL agent retransformation capable all what was needed to do is to add *can_retransform_classes* capability. With this modification the class retransformation has been finally achieved.

When the class retransformation is called from Java environment it takes the class name as an argument and the native function of the native agent is called. The agent uses the JVM TI environment to get array with all loaded classes, goes through it and checks names of classes. When class with specified name is found it calls retransformation of this class. If the desired class is not found, then it is not already loaded in the JVM. The DiSL agent automatically transforms the class at the time it will be loaded. That means it will be loaded in the same state like if it was found when the retransformation was called.

3.2 Measuring controller

The part running on the same machine where the instrumentation is done is called measuring controller. Its tasks are to send commands for class retransformation and receive measured data from measuring agent and to control measuring process. The measuring process is controlled by monitoring if some method measurement should be added or suspended and selecting the right method for it. Also if method measurement is completed, then the controller removes it and evaluates results.

Controller is separated into two parts:

core

Controller use also services provided by core part like measuring agent. If the services are not described in the section 3.1 or are used in a different context, then they are described in this section. This part is packaged in *core.jar* and has to be added to the class path of the instrumentation application. The difference from measuring agent is that it is not needed to by appended to the bootstrap class path, the regular class path is enough.

disl

This part provides services used exclusively by the measuring controller and does not affect the measured application in other way than is adding or removing measuring code. Is it packaged in *disl-instr.jar* which should be added to the class path of the instrumentation application. Also it uses third party libraries which should be added to the class path too. These libraries are *log4j-api-2.1.jar* and *log4j-core-2.1.jar* for logging, *ini4j-0.5.2.jar* for configuration, *commons-math3-3.3.jar* for evaluation and *disl-server.jar* for instrumentation.

3.2.1 Identification and storage

Measuring controller use helping classes for unique identification and storage of values which are described in this section.

Package *cz.cuni.mff.spl.description* placed in *core.jar* provides classes to store measurement identification. Method name and descriptor is stored in a single class *MethodDescription* which is used in *ClassDescription* to store class name together with method description.

Both classes are able to convert their instance to string and create instance back from string representation. Their string representations are created by *DataNameCreator* which is also used during the instrumentation to create unique name for measurement of a method. This assures that every part of the application use single format of unique identifier of a measured method.

Class description is used as an identifier of a method including its class. Method description identifies method in some particular class.

Package *cz.cuni.mff.spl.server.storage* from *disl-instr.jar* provides classes for storage objects in a hash map and a queue. At first all values are added to the map which then becomes unmodifiable. Next the values are queued to wait for processing. When new value should be processed the queue remove value with highest priority and returns it. When some value is processed it is stored in a list of processed values.

Class *QueueMap* is common base for *OrderedMap* which stores values in a predefined order and *UnorderedMap* which stores values unordered. Both specializations functionality is the same but they differs in their implementation. Ordered map returns value with highest priority fast, but the priority of queued items must not change. Unordered map chooses value with highest priority by going through the queued items and selecting the value with highest priority, but the advantage is that priority of queued items can change.

3.2.2 Instrumentation

The main service of instrumentation, which is used by the DiSL engine, is to choose whether measuring code should be added to the method or not. It is provided by package *cz.cuni.mff.spl.disl*. The class *DiSLMeasure* contains methods with measuring code marked with DiSL annotations to be added to the measured methods. Whether or not the measuring code is added depends on a guard. Guards are described in section 2.4.

There are three types how measurement can be started and one type of measurement stop. Measurement stop type is the same for every start type so the choice if it should be added to the method end depends only if the method is present in the list of currently measured methods.

Measurement stop is inlined by *stop* method and uses *AllModeGuard* class as a guard.

The type of measurement start depends on the mode which is set for computing size of method workload. There are three possible modes:

nocheck

Using this mode no size of method workload is computed. It is the fastest mode of measurement but the measured methods cannot be compared using only similar size of their workload.

This type of measurement start is inlined by *startNoCheck* method and uses *NoCheckGuard* class as a guard.

sizeof

This mode uses simple and fast computing of method workload. For every method's argument it calls *sizeOf* method of *SizeOf* class from *net-.sourceforge.sizeof* package. This class is used as a Java agent so it has access to the *java.lang.instrument.Instrument* instance and calls *getObject-Size* on it. This returns an implementation-specific approximation of the amount of storage consumed by the specified object. The size of method workload is the sum of values returned by the method.

This type of measurement start is inlined by *startSizeOf* method and uses *SizeOfGuard* class as a guard.

deep

Usage of this mode adds the higher overhead caused by computing size of method workload, but it also produces the most accurate results. The computing depends if the method was specified with size processor which can compute its workload size and if the argument implements specific interface. These cases are described in detail in section 3.2.6. If none of these cases which depends on additional user code occurs, then default computing is used. That calls *deepSizeOf* method of class *SizeOf* which is again based on the method *getObjectSize* from instrumentation, but this time it computes not just approximation of the object's size itself but also size of all objects reachable from it. The size of method workload is again the sum of values returned by the method.

This type of measurement start is inlined by *startDeepSizeOf* method and uses *DeepSizeOfGuard* class as a guard.

GuardHelper is a class used by all guards to determine whether the method should be instrumented by its guarded snippet or not. It also contains instance of *ServerDataListener* which is used for communication and is initialized together with configuration during the first access. This access is done by the DiSL engine when it asks some guard if some method should be instrumented.

All guards mentioned above are simple classes with a single method which calls *measuredMethod* of the class *GuardHelper*. This method takes instrumented method's full name and descriptor and mode of size checking of the guard which calls it. It propagates the call to the server listener and returns whether the method should be instrumented.

Method should be instrumented by a snippet if and only if the type of its guard corresponds with the mode of size checking that is set in configuration and the method is present in the list of currently measured methods. The only exception is *AllModeGuard* used to add measurement stop which use only method's name and descriptor to check if it should be instrumented, because it is used for every mode of argument size checking.

Class *MeasurementContext* is a kind of static context used in the snippets to get static information for the snippet. It is important to bear in mind that snippet code is executed on another JVM so it can not simple access information from measuring controller. Static context is used for this case. Its methods are executed during the instrumentation and only their output is used as a constant in the snippet code inlined to the measured method.

The context creates unique name of the measured method using *DataName-Creator*, enables access to the scale which is set in configuration of measuring controller and returns name of size processor of the method. The call to get size processor propagates through listener and scheduler to the controller which stores classes by their name and is able to get correct measured method which returns its processor defined in the task file.

All informations the context provides after the server listener is initialized are constant, because the configuration and set of all measured methods after initialization is constant and the name of method is created in the same way for every call so for the same method it returns the same name.

3.2.3 Server listener

The package *cz.cuni.mff.spl.server* contains all services provided by the measuring controller which are not directly used by DiSL engine. The engine accesses only the list of currently measured methods, the configuration and set of all measured methods through guards and measurement static context. Only the list of currently measured method can change after the listener initialization. Other services are used internally to take control of measuring process.

This package itself contains only *Configuration* class which provides configuration for all parts of measuring controller and is described in section 3.3 and class *ServerDataListener* which specializes common *DataListener* for the function of server listener. Other services are in appropriate sub packages.

The server listener runs in a new thread different from the instrumentation thread and is responsible for communication with measuring agent. It uses scheduler to check if some retransformation command should be send. If some class

should be retransformed more than once in a single communication cycle, then only single message about retransformation is sent due to storing messages without duplicities. For example, this case may appear if one method finishes its measurement and other method of the same class should starts its measurement. There is not need to retransform the class twice because all modifications will be done at once.

The server listener is specialized in these ways:

- The difference of connection initialization is that it uses server kind of channel and the channel is placed in a non-blocking mode, because the listener needs to be able to return from reading data if no data arrives for long period, so it can add new method measurement when the delay between measurements is over the limit.
- During initialization it creates new scheduler instance and stores it.
- It provides access to the configuration and scheduler instance for guards and measurement static context.
- When new message is received it converts it to the *Measurement* instance and passes it to the scheduler. Then it asks the scheduler whether some method measurement should be removed or added. The listener output does not distinguish between measurement adding or removing. In both cases it just adds the name of method's class to the queue of waiting messages to be retransformed. The choice if the method should be instrumented is made by the guard during the retransformation.

  However, the listener needs to keep the same order of reactions to the received measurement. First it adds new measurement to the scheduler. Then it checks if some method measurement should be removed because it is finished by the received measurement. After that it checks if some measurements should be added or suspended. The last two actions can be called in any order because only one of them can happen as a reaction for receiving a single measurement.
- When no measurement is received it informs the scheduler about it and checks whether new measurement should be added.

3.2.4 Scheduling

Scheduling service means to observe how often new measurement is done and adjust the number of currently measured methods. Also it selects which method measurement should be added, suspended or removed. This service is provided by *Scheduler* class from *cz.cuni.mff.spl.server.scheduler* package.

This class is a specialization of *TimeStorage* class which is the base for objects storing measurements and the difference between time of last two measurements.

The scheduler computes new value of difference between last two measurement ends when it receives new measurement from server listener. New value is computed as a weighted mean of old and new difference:

$difference = (oldDifference * weight + newDifference)/(weight + 1)$

The *weight* is configurable and the higher it is the more time to change the value is needed. This helps to compensate sudden but not frequent changes between measurement end times.

The scheduler changes the optimal number of measurements which should run simultaneously in the case the time difference is out of its limits. If the difference is shorter than minimum, then it is decreased and if the difference is longer than maximum, then it is increased.

The optimal number of measurements is checked by server listener which adds or suspends measurement until the real number of measured methods is the same as the optimal. The optimal number cannot be lower than one so at least one method can be measured. Upper bound of the optimal number is limited by the number of methods which measurement has not yet been completed.

The scheduler contains a hash table of currently measured methods. The key to the table is created from method's name and descriptor. It is used by instrumentation guards to check whether a method should be instrumented. If the method is present in the table then, it should be measured at this time.

When the server listener asks scheduler for a new method to be measured, the scheduler gets the correct method from *Controller* class, adds it to the table of currently measured methods and return it to the listener.

Selection of a method that should be suspended is done by the scheduler which goes through the table of currently measured methods and select a method with minimal time between its last two measurement ends. This method is removed from the table, its placed to suspended status and returned to the server listener.

If new measurement completes measurement of some methods, these methods inform the scheduler about it. The scheduler removes them from the table of active methods and add them to the list of methods which should be deactivated. The server listener picks every method from the list at the end of a communication cycle and add a command to retransform its class to the queue of messages.

When the scheduler is informed by the listener that no new measurement is received in a communication cycle, it compares current time with the end time of a last measurement. If the difference is higher than its limit, then it increases the optimal number of measured methods and the listener should check it and add a new method to be measured.

3.2.5 Controlling

The controlling part of the measuring controller is responsible for loading measuring tasks, storing their representation and selecting the right method which measurement should be added. These services are provided by classes from *cz.cuni.mff.spl.server.controller* package.

The main class *Controller* is called whenever the scheduler needs selection of a new method to be measured. Its instance is created by the scheduler and during its initialization it loads measuring tasks from a file. The measuring task can be a single method to be measured or a comparison of two methods.

Task type depends on the mode of running which can be:

measurement
: In this mode only method measuring without any comparison and evaluation is done. In the task file should be just single methods for measurement.

comparison
: This mode runs method measurement and evaluates the comparison results. Comparisons of two methods are expected in the task file.

The controller creates *MeasuredClass* instances for every class of a measured method and stores them in an *OrderedMap* during the task loading. The key to the hash table of the storage is the class name, so every measured method of a single class is stored in the same class.

MeasuredClass stores instances of *MeasuredMethod* for every measured method of the class in an *UnorderedMap*. The key to the hash table of the storage is an instance of *MethodDescription* which contains class name, method name and method descriptor, so if the task file contains the same method more than once, then just single instance is used for every occurrence.

The difference between ordered and unordered map is described in section 3.2.1.

For the comparison mode of running the controller also creates *Comparison* instances which stores *MeasuredClass* instances in the same controller's storage as before, but the comparison itself is not stored in the controller. Instead, every measured method has a list of comparisons where it is used.

When a new method for measurement should be selected, the controller removes a class with the highest priority from the storage queue and asks the class to select its method for measurement. The class removes a method with the highest priority from the queue of methods waiting for measurement and returns it.

The priority of the class which is described in section 1.5.2 changes, because the class loses one of its methods waiting for measurement and gains one method which measurement has been started. If the class has any other method waiting for measurement, then it is returned back to the controller's storage queue with this modified priority. If class does not have any waiting method left, then it is not returned back, so it cannot be selected again when new measurement is required.

When some method is suspended by the scheduler, the method is placed to the waiting status like it was at the beginning and it is stored back to the queue of waiting methods of its class. This changes priority of a particular class and this class should be now present in the controller's queue of classes with methods waiting for measurement. If the class is not present in the queue, then it is added with the new priority. But if the class is present in the queue, then it is needed to be removed first and after that added again to preserve correct priority order.

When a new measurement is added, the controller selects the right class and method based on the identifier of the measurement. The measurement is then added to the correct measured method, which checks the number of finished measurements and if there are enough, it completes measurement of the method.

Adding measurement to the method can change priority of the method, because the priority is based on the number of comparisons where this method occurs and number of remaining measurements. If two methods have the same number of comparisons, then the method with a lower number of remaining measurements has the higher priority as is described in section 1.5.2.

The number of comparisons can be lowered if method measurement is completed and number of remaining measurements decreases with every added mea-

surement. However, unlike the class which has to be removed and added again in the queue when its priority changes, the method does not need to do so. That is because methods are stored in an unordered map which allows changes of priority of stored values.

Completing of measurement means to place the method to processed status, store measured values to the file, run evaluation and inform method's class about the completion. If the class has all measured method completed, then the class itself is completed and informs the controller about it. When all classes are completed, then all measurements are done.

3.2.6 Size of method workload

Adding measurement to the measured method includes to add it to the list of measurements waiting to be written into an output file and if comparison mode of running is used, then the measurement is also added to the bucket of measurements with similar size of method workload.

Different ways of method workload size computing are supported by package *cz.cuni.mff.spl.instrumentation.size* and its sub packages placed in *core.jar*.

The existing classes support default computing as is described in section 3.2.2 which computes size of method workload as a single value. The size can be scaled using linear or logarithmic scale with configurable base.

For example, linear scale with 10 used as a base aggregates measurements which workload size is the same multiple of 10. Logarithm scale used with the same base aggregates measurements which workload size is the same power of 10.

These default workload size computing can be used just by setting the proper configuration and does not need any user code to be written. However, for deep mode of method workload size computing there is possibility for the user to use own specific computing.

One way is to implement interface *Computable*. If the method argument is an object which implements this interface, then its *computeSize* method is used to compute its size instead of default computing by *deepSizeOf* method of class *SizeOf*.

When specific computing without adding implementation of the interface to existing classes is needed or there should be used different scales, multiple values of workload size or different aggregation, there can be created implementation of *SizeProcessor*.

Different size processor can be specified for every method in the task file and it is used to compute workload size for method's measurement. It has method *computeSize* which takes instance of a class of the measured method and array of method's arguments and produce specialization of *MethodSize* class. This class stores any values created by computing workload size and is added to the measurement.

When measurement is sent to the measuring controller, the *MethodSize* instance is converted to its string representation with other measured values. As a part of the aggregation of measurements by the workload size the *MethodSize* instance is created, initialized from its string representation and its method *aggregate* is called with specific scale base to get instance of *SizeAggregator*.

The aggregator is used to store measurements with similar workload size so the size which are considered similar should return equal aggregator.

The scale base used for aggregation should implement interface *ScaleBase* and is obtained by *getBase* method of size processor.

Thanks to this hierarchy only the size processor is needed to be explicitly specified, because method size and scale base are created by the processor and aggregator is created by the method size.

Any user created classes used for workload size computing should be added to the same class path as the *core.jar*, because they are used by the measuring agent to compute size of method workload and then by the measuring controller to aggregate measurements by their workload size.

3.2.7 Evaluation

If the comparison run mode is used, then completing of a method measurement also means to run evaluation of the results. For this case the scheduler creates instance of *Evaluator* class from *cz.cuni.mff.spl.server.evaluator* package during its initialization.

When a measurement is added to a bucket of measurements with similar workload size chosen by the aggregator, the method checks whether the bucket size is equal to the configured minimal number of measurements. If it is equal, then the bucket is full and may be evaluated, so the method goes through its comparisons and checks if there is a comparison that can be evaluated.

The comparison can be evaluated if the other method involved in the comparison has full bucket of measurements with equal aggregator. If the matching bucket of the other method is not yet full, the evaluation is postponed until some measurement of the other method is added and it fills the bucket.

After the comparison is evaluated it is removed from the list of comparisons of both its methods. If the method has no comparison to be evaluated left, then its measurement is finished.

It may happen that the number of measurement reaches the maximal limit without evaluation of all its comparisons. In this case the method checks all its comparisons when the method measurement is stopped and searches for comparisons where the other method has processed status. That means both methods of the comparison finished their measurement and no other measurement will be added to them, so the evaluation may be done and it try to produce at least some result with lower accuracy. If there is a lesser number of samples of the method than 2, then the evaluation cannot be done at all.

3.3 Configuration

Configuration of the measuring process includes configuration of both measuring agent and measuring controller. Without any configuration the default values are used.

3.3.1 Configuration of the agent

The measuring agent can be configured by setting system property for the JVM using *-D* option. These values can be configured:

spl.client.port
: This property sets the integer value of the port used by agent for communication with measuring controller. The default port is *38692*.

spl.client.sleep
: It sets the time between two communication cycles of agent thread as an integer value in milliseconds. The default sleep time is *100* milliseconds.

  If it is greater than zero, then the agent's thread calls sleep for the specified time at the end of each communication cycle. The lesser is the time the faster are measurements sent to the controller so it can faster react if the delay between two measurements is out of its bound, however, lesser sleep time causes higher communication overhead.

spl.client.measurements
: It sets the integer value of maximal number of measurements allowed for a single method. The default limit is *10000*.

  This should generally be greater than maximal number of measurements configured for measuring controller, because it does not remove the method's measuring code, the measuring code just decides to not start measurement and so there is some overhead even when measurement does not start.

  It is a safety limit for fast and frequently used methods which can generate a lot of measurements in a short period of time. The measurements can be generated faster than it is possible to send them to the measuring controller which cannot react in time and remove the method measurement. This causes application's termination by running out of memory.

  When number of measurements for a single method is bounded by this limit, no more measurements are started and measurements that has been already done are sent to the controller over time. After these measurements are received by the controller, the method exceeds the number of maximal measurements and the controller finishes its measurement.

  If the agent's limit is lower than the controller's limit, then method measurement may be never ended, measuring code would not be ever removed and the method's comparisons may not be evaluated if it does not reach the minimal number of measurements for some bucket of aggregated measurements.

3.3.2 Configuration of the controller

INI file is used for configuration of the measuring controller, because there can be specified higher number of parameters for it. The default path to the file is *splserver.ini*, which can be changed by system property **spl.server.config** for the JVM. It is the only parameter of the controller which is specified by system property.

Other values are loaded from the INI file during controller's initialization and are stored in class *Configuration* from package *cz.cuni.mff.spl.server*.

Values are separated into four sections to highlight their sense.

communication

This section contains parameters used for communication with measuring agent.

initial-sleep
: This is an integer number of milliseconds for which the controller will sleep after its initialization. The default sleep time is *3000* milliseconds. It can be used to delay the start of measurement until the measured application starts-up.

cycle-sleep
: It sets the time between two communication cycles of the controller thread as an integer value in milliseconds. The default sleep time is *100* milliseconds.

  If it is greater than zero, then the controller's thread calls sleep for the specified time at the end of each communication cycle. The lesser is the time the faster controller receives measurements and can react if the delay between two measurements is out of its bound. It may be similar to the sleep time set for the agent.

port
: It is the integer value of the port used for communication with measuring agent. The default port is *38692*.

scheduler

This section contains parameters which affects the number of measurements which run simultaneously and type of method workload size computing.

initial-optimal-measurements
: This is an integer number of measured methods which will be added for measurement after the measurement starts. The default number is *3*.

  If it is set to low number, then the measurement will start slowly and if the delay between measurements is longer than limit, then new measurement is added. Measurement can take longer but the initial overhead is lesser.

  If it is set to number higher or equal to the number of all measured methods, then all methods will be added for measurement when the measuring controller starts. Measurements can be suspended if the delay between them is lower than limit. This will cause higher initial overhead but measurement will be completed faster.

min-measuring-diff
: This is a long value of the lower bound for delay between two measurement ends in nanoseconds. The initial delay is *1000000* nanoseconds which is 0.001 second. If the delay is lower than this bound and there is more than one currently measured method, then some measurement will be suspended.

max-measuring-diff
This is a long value of the upper bound for delay between two measurement ends in nanoseconds. The initial delay is *100000000* nanoseconds which is 0.1 second. If the delay is higher than this bound and there is any method waiting for measurement, then some measurement will be added.

argument-size
This sets mode of workload size computing. The initial value is *deep*. Possible values are:

nocheck disables workload size computing.

sizeof sets simple and fast, but inaccurate mode of workload size computing.

deep enables accurate and user specified workload size computing.

More detailed description of workload size computing modes is in section 3.2.2 and user specific computing is described in section 3.2.6

time-diff-weight
It sets the integer weight used to compute new value of the difference between last two measurement ends. The default weight is *3*. Section 3.2.4 describes how new value is computed.

controller

This section contains values used to specify measurement input and output, limits for number of measurement of a method and mode of running.

tasks-file
It specifies path to the file containing tasks for measurement. The default path is *instr.tasks*. Format of this file is described later in section 3.3.3.

output-dir
It specifies path to the directory where measurements are written into files. The default path is *out*. The directory is created if it does not exists. Every method use own file for its measurements which name is the identifier of the method consisting of class name, method name and descriptor and file extension *.msr* where chars /, < and > are replaced by dot.

overwrite-measurements
It is a boolean value determining whether files with measurements should be overwritten if they exists. The default value is *1* for which the files are overwritten. If it is set to *0*, then new measurements are added to the existing files.

human-readable-measurements
It is a boolean value determining whether measurements should be written into files in a human readable format. The default value is *1* for which human readable format is used. When it is set to *0*, then the values of measurement are written in this order separated by comma: start time, stop

time, recursion depth, thread identifier, measurement identifier, workload size string representation.

max-method-measurements
It is an integer value of the maximum number of measurements which can be measured for a single method regardless of the workload size. The default value is *1000*. The higher value is used, the higher is the chance to measure enough samples of the same similar workload size for evaluation.

min-method-measurements
It is an integer value of the minimal number of measurements aggregated by its workload size which can be used for evaluation. The default value is *50*. Higher value can produce more accurate results but also extends the period for which method is measured.

measurements-to-flush
It is an integer number of measurements before they are written to the output file of appropriate method. The default value is *50*. All measurements are written to the file when method measurement is completed. This periodical recording is for the case when the measuring process is unexpectedly terminated.

run-mode
It specifies run mode of the measurement. The default value is *comparison*. Possible values are:

measurement mode runs only measurement without evaluation of comparisons. In the task file should be just methods for measurement.

comparison mode runs method measurement and evaluates the comparison results. Comparisons of two methods are expected in the task file.

evaluator

This section contains values used for evaluation of comparisons. These values are used only for *comparison* run mode.

result-file-path
It specifies the path to the file where comparison results are written. The file and all parent directories are created if they do not exist. The default path is *out/result.eval*.

limit-p-value
It is a double value of the limit p-value used to check whether t-test result satisfies hypothesis that the two means are equal. If the result p-value is greater or equal to the limit p-value, then the hypothesis is considered to be satisfied. The default value is *0.05*.

overwrite-results
This boolean value determines whether file where comparison results are written should be overwritten (value *1*) or new results should be appended to the file if it exists (value *0*). The default value is *1*.

human-readable-results

It is a boolean value determining whether measurement comparison results should be written into file in a human readable format. The default value is *1* for which human readable format is used. When it is set to *0*, then the values of measurement are written in this order separated by comma: left method's full name, comparison sing, left multiplication constant, right multiplication constant, right method's full name, result of comparison, result p-value, lesser number of measurements to compare in the same interval of workload size values, interval of values of workload size used for aggregation.

arguments-size-scale-base

It is a long value of a scale base. The default value is *100*.

arguments-size-scale

It specifies the scale used to store measurements with similar size of method arguments. Possible values are *linear* or *lin* for linear scale and *logarithmic* or *log* for logarithmic scale. The default value is *linear*. More about the predefined scales can be found in section 3.2.6.

Both scale and scale base configuration affect comparison only when *sizeof* mode of argument size computing is used. For *deep* mode they have effect only when no size processor is used or it should be used but its initialisation fails, then the default scale is used instead.

3.3.3 Measuring tasks

These tasks are specified in a separate file which default path is *instr.tasks*. Its content depends on the run mode of the measuring:

measurement

For *measurement* mode the file should contain only methods to be measured separated by a new line. The method is specified by its class name including packages separated by slash (inner class is separated by dollar sign), followed by dot and method name and at the end its descriptor is added. For example method *int compareTo(Integer anotherInteger)* of the class *java.lang.Integer* is specified as *java/lang/Integer.compareTo(Ljava/lang/Integer;)I*

It is also possible to measure constructor of a class. This can be done when *<init>* is used as a name of the measured method, the rest of method specification is the same as for ordinary method.

Argument size processor can be optionally specified after the method specification. It can be done by adding comma and processor's class name including its package separated by dots after the method specification. The processor has to implement *cz.cuni.mff.spl.instrumentation.size.computing-.SizeProcessor* interface and has to implement constructor with an empty argument list. Its instance is created by reflection during measurement to compute sizes of method arguments when *deep* mode of argument size computing is configured, but these values are used only when *comparison* mode is used together.

The method specification can be figured like this *pkg/MethodClass.method descriptor, pkg.ProcessorClass.*

comparison
Comparison of two methods should be specified on each line when this mode is used. Comparison is described by left method specification optionally with argument size processor followed by comparison sign, then optional multiplication expression can be used and right method specification optionally with argument size processor is placed at the end.

Comparison sign can be one of <, <=, =, >=, >.

Multiplication expression is specified as two double values in a parenthesis separated by comma. Left side of the expression is used to multiply values of time measured for left method and right side of the expression is used to multiply values of time measured for the right method before comparison is evaluated by t-test. If no expression is specified, then default *(1, 1)* value is used which means that no value will be changed.

Comparison can be figured like this *leftMethod sign (leftExpression, rightExpression) rightMethod.*

3.4 Usage

This section describes what system requirements are needed and how to compile and run the application. It also explains examples which are part of the distribution to demonstrate the usage.

3.4.1 Compilation

The application is distributed as a source code that needs to be compiled. To do so it is needed to use Unix operation system with following programs:

- GCC (v 4.7.1)
- make (v 3.82)
- Java JDK (v 1.7)
- Apache Ant (v 1.8.2)

On some platforms there may be need to set *JAVA_HOME* variable to the right path of the JDK directory manually before compilation. Compilation is done by executing command

```
ant
```

from the application root directory. Shared native library is created into *build/c-agent* directory and Java archives are placed in *build/jars* directory.

To get all files needed to run measuring with custom application use command

```
ant dist
```

It will compile and copy all necessary libraries into *dist/lib* folder and run scripts into *dist* folder. In this folder it also creates new empty *instr.task* file where measuring tasks may be written and copies default server configuration to the *splserver.ini* file. There is also *runExample.sh* script which sets paths to the needed libraries to *lib* folder and then calls *runApp.sh* script described later.

3.4.2 Running

To run the application use command

```
./runApp.sh <classpath> <main-class> [params]
```

which takes class path of the measured application as first parameter and main class as second parameter followed by optional number of parameters for the measured application. This script runs the controller server and then the measured application.

To launch controller server and application separately use

```
./runServer.sh
```

first to launch controlling server and then call

```
./runClient.sh <classpath> <main-class> [params]
```

to run the measured application.

Scripts *runClientOnly.sh* and *runServerOnly.sh* are shared by both ways of running and should not be called directly.

When custom argument size processor is used it is needed to set path to its JAR file in *PROCESSOR_JAR* environment variable. Colon must be added at the beginning of the path.

3.4.3 Examples

Project contains six examples illustrating application usage. They are located in *examples* directory each in its own folder. Before their compilation the main application should be already compiled. Every example is compiled with

```
ant
```

command called from the concrete example folder and except **retransformation** each example is launched by

```
./runExample.sh
```

command without parameters. If the example produces any files they are located in *out* folder.

Retransformation

This example is a little different because it does not use the controlling server. It uses just stub of it to show the ability to retransform classes at run time on demand. To run it it is needed to call

```
./runServer.sh
```

first and then

```
./runExample.sh
```

script, because there is needed to use two consoles to interact with server. This example does not use any configuration files.

Simple application which runs loop where it prints number of iteration and calls *Integer.compareTo* method is used as the measured application. The server prints information whenever it is asked to retransform a class and also every measurement it receives from the client application. After every console input in the server console confirmed by Enter it send a message to retransform class java.lang.Integer to the client application. For the first run the modification is active and it adds measuring code and printing start and stop of the transformed method's run. Second input deactivates the modification so it runs retransformation of the class without changes. Activation of modification will alternate after every console input for the server.

Measuring

There can be seen how to run the application with measurement and how to specify concrete method to measure. Custom user classes including inner class are measured and their methods use different number and type of parameters. Also *java.lang.Integer* class is measured to show the ability to measure Java system classes.

Comparing

Comparison of two methods is shown in this example. Also it uses deep argument size checking mode and custom implementation of *Computable* interface for objects that can compute its own size when they are used as an arguments of the measured method.

Processor

It shows comparison of two methods like in the previous example, but this time it uses custom argument size processor for deep argument size checking mode. This is useful to measure third party code which should not be changed. Because the JAR containing processor has to be on the class path of both client and server JVM, it is needed to set environment variable *PROCESSOR_JAR* to the path to that JAR and this path must be preceded by colon as is done in its *runExample.sh* script.

Sorting

Simple sorting application is used for measurement. It prints how long single cycle lasts so it shows measurement overhead for application which performance depends mainly on the processor. This example is described at full length later in section 4.1.

ROME

Measurement of ROME library is shown in the example. It is used to demonstrate measuring a real-world application and is described in detail later in section 4.2.

4. Evaluation

The described prototype was tested on two kinds of applications to show features of the approach and the results are shown in this chapter. First example is a simple sorting application and second one uses ROME as an example of usage with a real-world application.

4.1 Sorting

Simple application which contains cycle that sorts large number of integers was created to show interference of the measurement on the time of execution of an application which performance depends mainly on the processor. Only measurement mode without any comparisons was used in this example.

4.1.1 Measurement of sorting

Twenty-two of the most frequently called methods was chosen for measurement. All of them was from *java.lang* or *java.util* packages so it also demonstrates the ability to measure Java system classes. Example of measured times for method *java.lang.Integer.compareTo(Integer)* is on figure 4.1.

4.1.2 Measurement influence

Measurement has some unavoidable overhead to the measured application. However it is possible to control the current overhead by measuring different number of methods at once. Three possible configurations of the measuring application are shown in this chapter with their characteristics.

Measure all at once

First option is to measure all methods at once. It has the advantage that measurement is completed as fast as is possible, but it has also the highest maximal overhead.

This can be done by configuring high initial number of measured methods and low number of minimal delay between measurements so no method will be suspended. In the INI configuration these values may be changed (run mode was changed because there was not needed to use any comparisons):

```
[scheduler]
initial-optimal-measurements = 100
min-measuring-diff = 1

[controller]
run-mode = measurement
```

Figure 4.2 shows high peak of sorting cycle time when the measurement starts. The time rises from about 25 ms to over 1 s and then quickly falls back when the measurement is done in about 10 cycles.

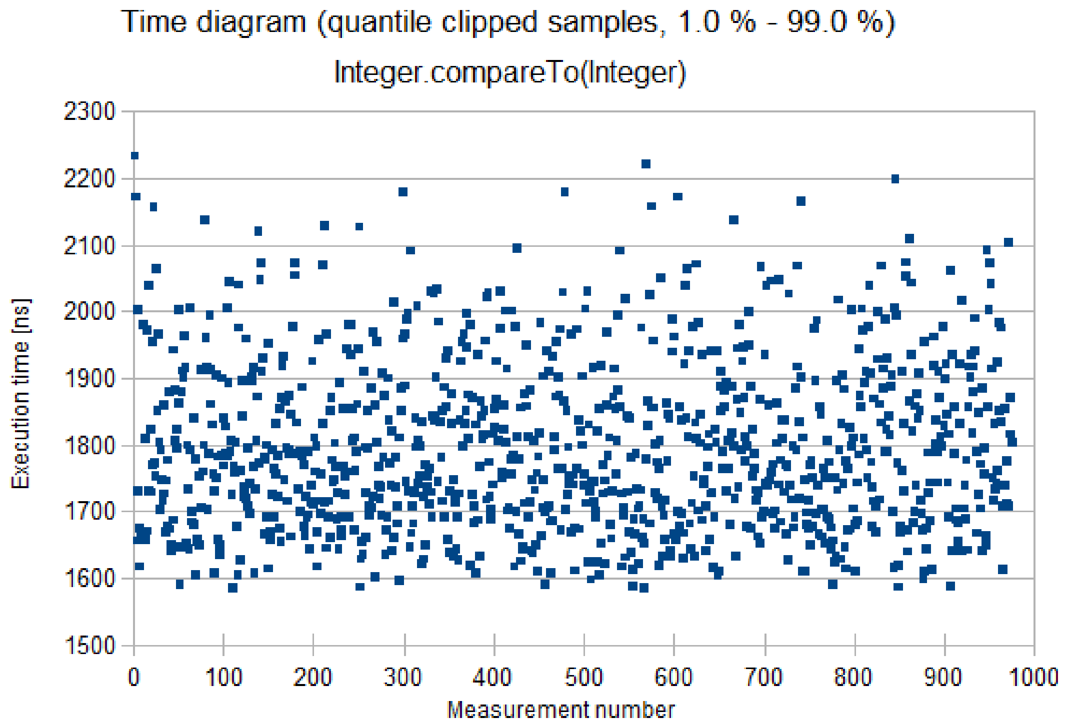


Figure 4.1: Integer.compareTo measurement time diagram.

Measure one by one

Contrary to measure all methods at once there is possibility to measure one method after another. There may be never measured more than one method at once so the overhead is as low as is possible. The disadvantage is that measurement of all methods lasts longer.

To use this kind of measurement it is possible to set 1 as the initial number of methods to measure and high number of maximal delay between measurements which can be done by setting the INI configuration like this:

```
[scheduler]
initial-optimal-measurements = 1
max-measuring-diff = 100000000000

[controller]
run-mode = measurement
```

On figure 4.3 can be seen time behaviour of the sorting cycle. The maximal peak is around 400 ms which is much lower than previous overhead, but the application needs almost 80 cycles to finish measurement.

Measure with controlling

The third option is to measure multiple methods at once but with the possibility to dynamically change number of active methods. This allows to add method measurement when delay is long or suspend it if the delay is short. This way may have lesser overhead than measure all methods at once and take less time than measure one by one.

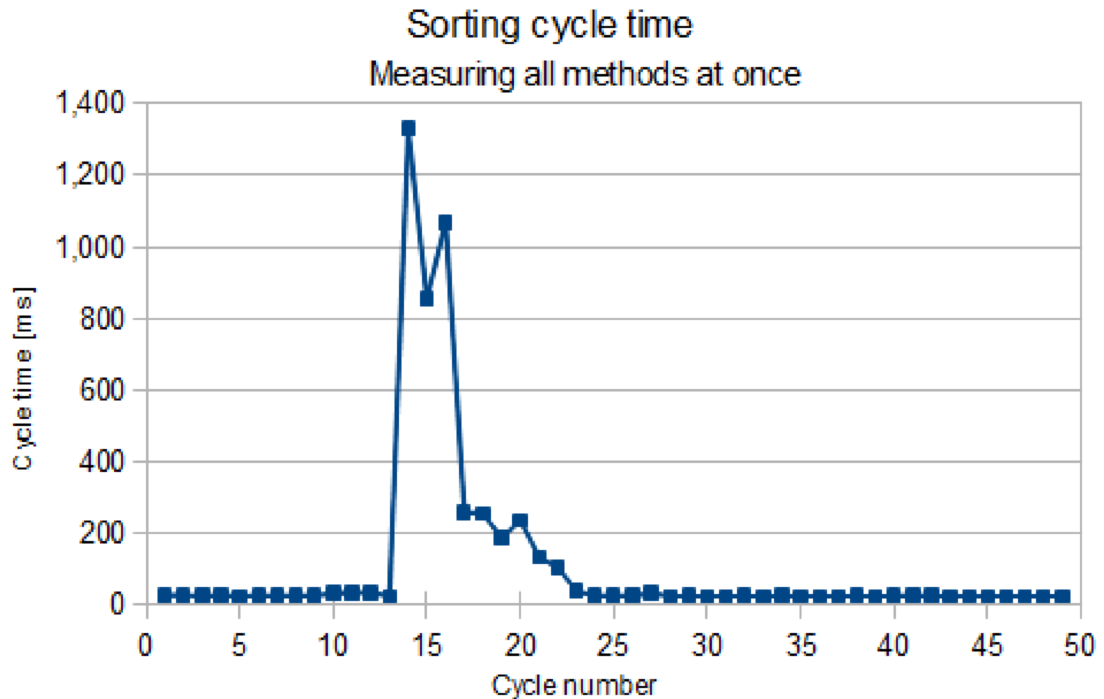


Figure 4.2: Sorting cycle time when all methods are measured at once.

The configuration may vary depending on measured application and what is considered as the range of acceptable delay between measurements. This time also time difference weight was set to higher value to prevent frequent oscillation of adding and suspending method measurement. This configuration was used for the example:

```
[scheduler]
initial-optimal-measurements = 1
min-measuring-diff = 1000
max-measuring-diff = 100000000
time-diff-weight = 5

[controller]
run-mode = measurement
```

Figure 4.4 shows time of sorting cycles during measurement. The highest overhead is about 600 ms which is higher than measuring methods one by one but it is about half of the highest overhead of measuring all methods at once. The number of sorting cycles needed to finish measurement is about 40 so it is finished in a half time than the measuring methods one by one.

Figure 4.5 presents number of concurrently measured methods during the measurement on the Y-axis and actions of controller when it decides that number of active methods should change are on the X-axis.

When method measurement is finished there is a single action decreasing the number of measured methods by one followed by increasing this number by one. It means that the controller removes completed method and activates measurement of a new method.

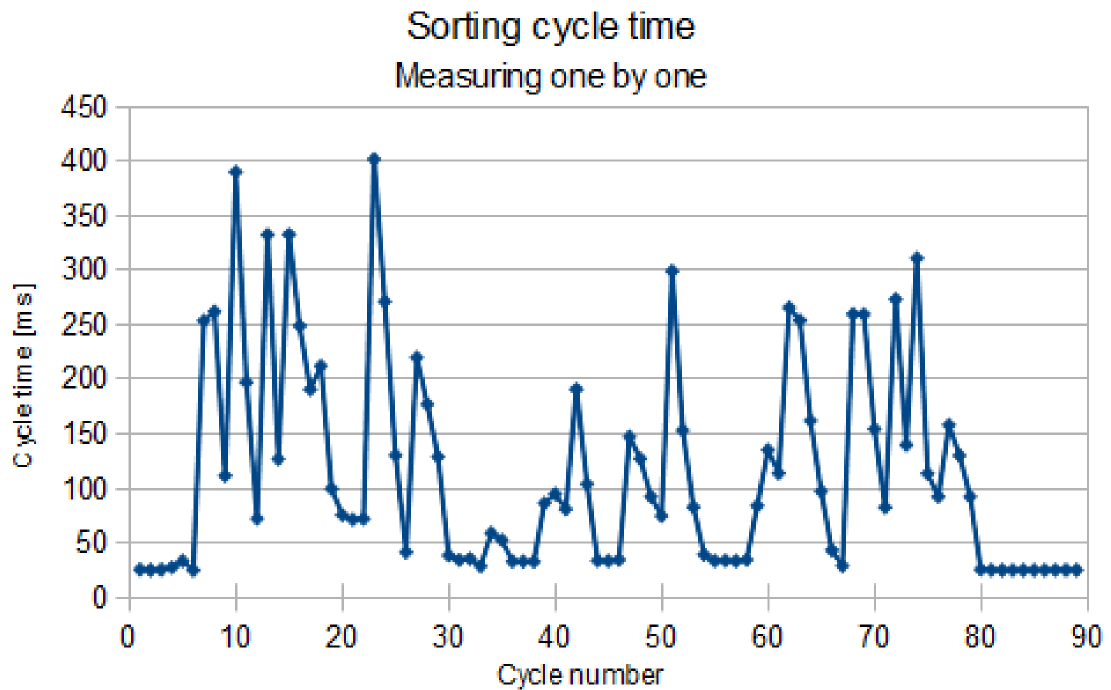


Figure 4.3: Sorting cycle time when methods are measured one by one.

When there are two same values of two following actions, the controller found that it needs to change the number of active methods for other reason than finishing measurement of a method. If the following action increases number of active methods, then the delay between measurements was higher than is allowed and the controller activates measurement of a new method. If the action decreases this number, then the delay was lesser than is allowed and controller suspends measurement of a method.

The measurement starts with single measured method followed by activation of another method because the delay between measurements was high. Then it alternates the number of active methods between 1 and 2 which means that two methods are measured concurrently.

After this it lowers the number of concurrently measured methods because the delay was low and measures only single method at once. The last peak means that the delay became high again and the controller activates measurement of new methods until all methods that left for measurement were activated and at the end it subsequently deactivates measurement of completed methods.

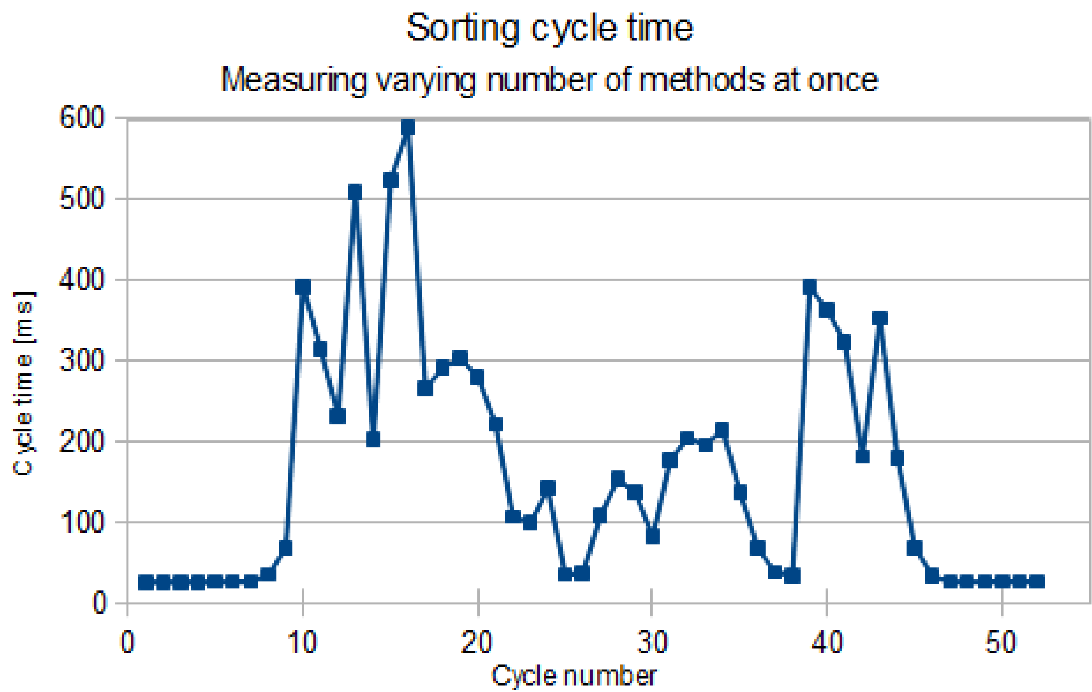


Figure 4.4: Sorting cycle time when varying number of methods is measured at once.

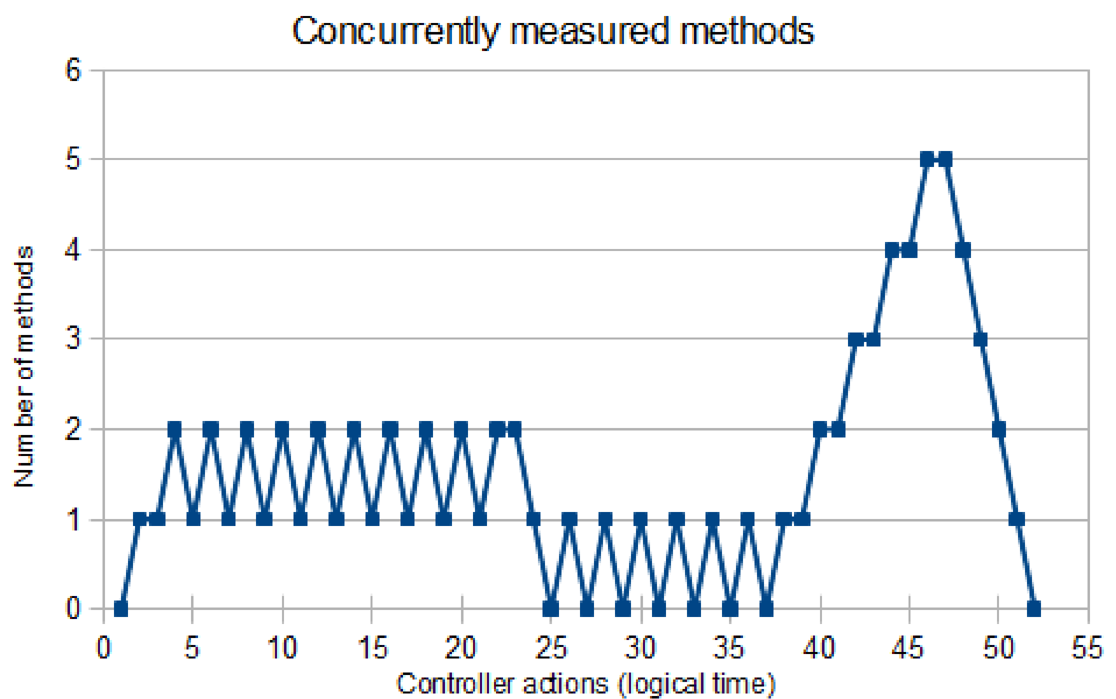


Figure 4.5: Number of concurrently measured methods during sorting.

4.2 ROME

Previous work on SPL tools included case study that successfully used JDOM [6] library, which purpose is to provide a Java-based solution for accessing, manipulating, and outputting XML data from Java code, as a real-world application. This time it was intended to use it too but now with real data instead of some testing inputs.

ROME [11] is a set of RSS and Atom utilities for Java. It is open source and includes a set of parsers, converters and generators for the various formats of syndication feeds.

ROME uses JDOM for manipulating with XML data, so it was chosen to be used and measured in this work. Simple application which reads a syndication feed using ROME was created. It reads feed in a cycle from a real web and prints time how long lasts one cycle.

4.2.1 Measured methods

Fast methods which are frequently called and complex long-running methods was chosen for measurement.

Verifier

"A utility class to handle well-formedness checks on names, data, and other verification tasks for JDOM" (JavaDoc documentation).

It contains fast and during document parsing frequently called methods. These methods was measured:

checkAttributeName(String)

It checks if the supplied name is a legal attribute name.

checkElementName(String)

It checks if the supplied name is a legal element name.

checkXMLName(String)

It is a utility function for sharing the base process of checking any XML name. It is used by both *checkAttributeName* and *checkElementName* methods so its run time is compared with them and it is assumed that the outer method runs at most two times longer than the inner method.

SAXBuilder

"Builds a JDOM Document using a SAX parser" (JavaDoc documentation).

Its measured method *build(java.io.Reader)* builds a document from the supplied Reader.

SyndFeedInput

Only this class is from ROME utility and its measured method is *build(java-.io.Reader)*. It builds instance of interface *SyndFeed* which is interface for all types of feeds. *SAXBuilder* is used to create the document from which the feed representation is created so both build method times are compared and the *SyndFeedInput* is assumed to run at most two times longer than the *SAXBuilder*.

```
[communication]
cycle-sleep = 0

[scheduler]
initial-optimal-measurements = 1
min-measuring-diff = 100000
max-measuring-diff = 100000000
argument-size = nocheck
time-diff-weight = 4

[controller]
tasks-file = cmp.tasks
output-dir = out/measurements
max-method-measurements = 500
min-method-measurements = 200
```

Listing 18: INI configuration for ROME measurement.

4.2.2 Configuration

Listing 18 shows changes in INI configuration used for controlling server and listing 19 presents the file with comparing tasks (original file has one comparison on a single line, syntax is described in section3.3.3).

4.2.3 Overhead

Because this application downloads feeds over the Internet, the time of feed object creation depends significantly on the communication delay, so the measurement has only minor influence. The time of feed creation is illustrated on figure 4.6, where the moment of measurement start and stop is highlighted.

Figure 4.7 shows number of concurrently measured methods during the process. It started with single method, then the controller activated measurement of a new method. This cause that the delay between measurements was too low, so one method was suspended. After its measurement was finished new method was activated. Then it continues to activate new methods until all remaining methods was measured at once. At the end one method should be suspended but it completed its measurement before the suspended method was deactivated, so there was not needed to activate it again and all remaining method finished their measurement.

4.2.4 Results

The measurement shows that assumptions holds but the results was a little unstable. This is the reason that every outer method is assumed to last at most two times longer than the inner method. In reality the multiplication may be closer to one, but then the results may be more unstable.

Table 4.1 shows measured p-values for each comparison (limit p-value for equality was 0.05).

```
com/sun/syndication/io/SyndFeedInput.build(Ljava/io/Reader;)
Lcom/sun/syndication/feed/synd/SyndFeed;
<= (1,2)
org/jdom/input/SAXBuilder.build(Ljava/io/Reader;)
Lorg/jdom/Document;

org/jdom/Verifier.checkXMLName(Ljava/lang/String;)
Ljava/lang/String;
>= (2,1)
org/jdom/Verifier.checkElementName(Ljava/lang/String;)
Ljava/lang/String;

org/jdom/Verifier.checkXMLName(Ljava/lang/String;)
Ljava/lang/String;
>= (2,1)
org/jdom/Verifier.checkAttributeName(Ljava/lang/String;)
Ljava/lang/String;
```

Listing 19: Comparing tasks for ROME measurement.

Left measurement	Sign	Multiplication	Right measurement	p-value
SyndFeedInput	<=	(1,2)	SAXBuilder	0.983
checkXMLName	>=	(2,1)	checkElementName	< 0.00001
checkXMLName	>=	(2,1)	checkAttributeName	< 0.00001

Table 4.1: Comparisons and their p-value.

Comparison of probability densities of creation times of an XML document and the feed created from it is presented on figure 4.8.

Comparison for fast running methods are shown on figure 4.9 for methods *Verifier.checkAttributeName* and *Verifier.checkXMLName* and on figure 4.10 for *Verifier.checkElementName* and *Verifier.checkXMLName*.

The values for method *Verifier.checkXMLName* are the same on both figures because the result uses the same values for both comparisons where this method is used.

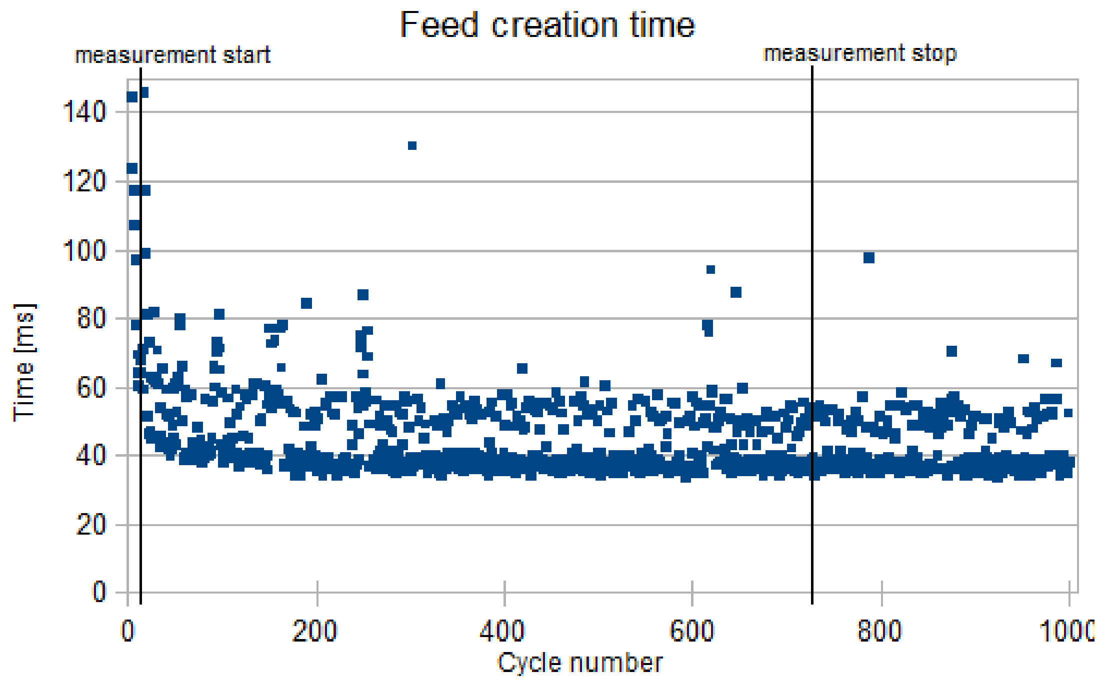


Figure 4.6: Feed creation time.

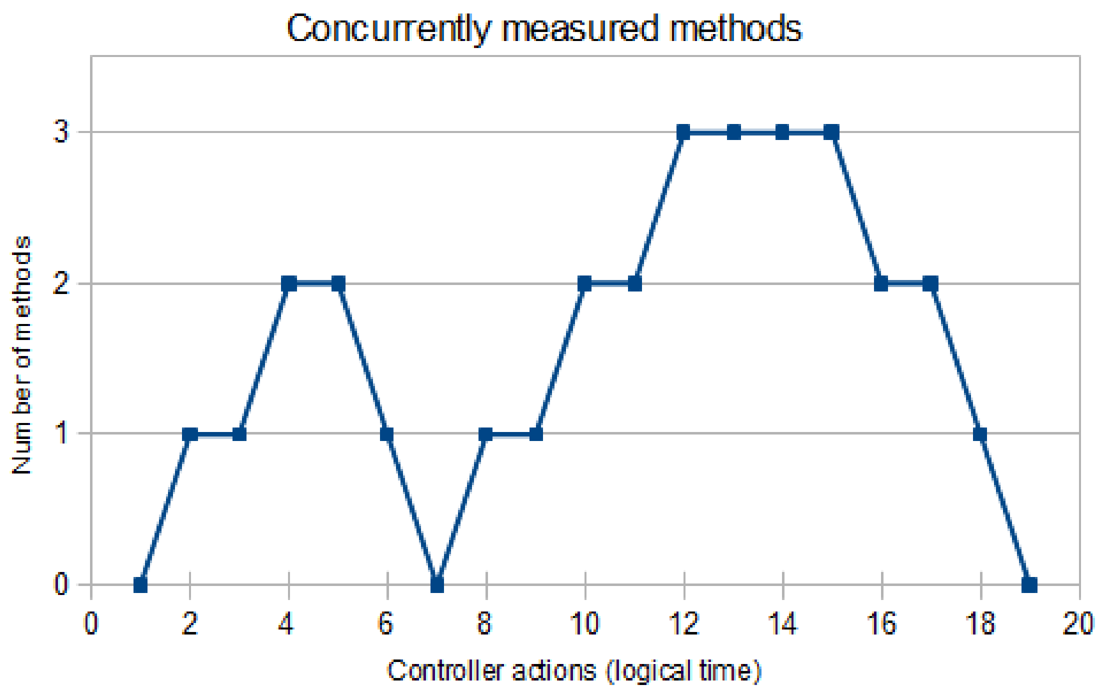


Figure 4.7: Number of concurrently measured methods during feed creation measurement.

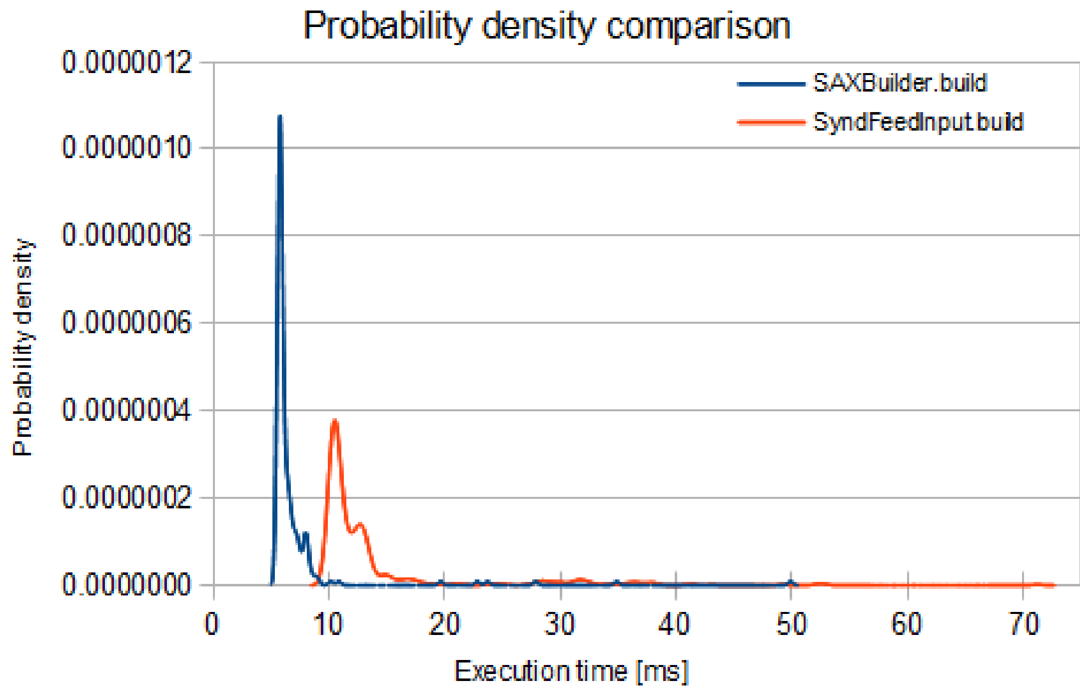


Figure 4.8: Feed and XML creation time comparison.

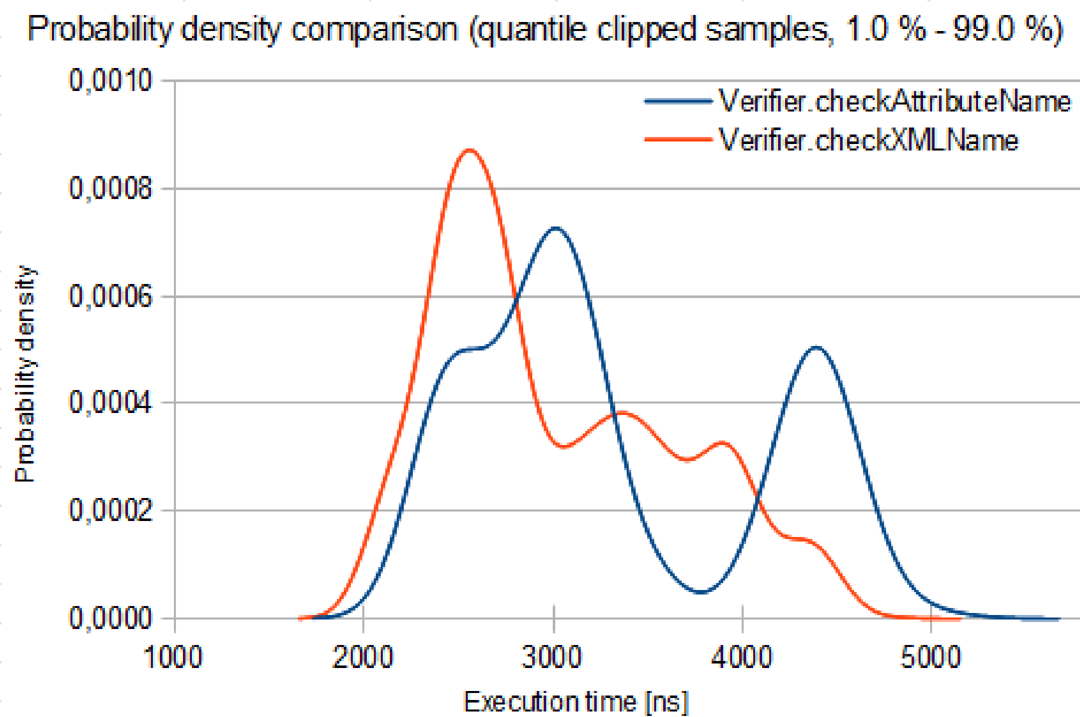


Figure 4.9: Time comparison of checkAttributeName and checkXMLName.

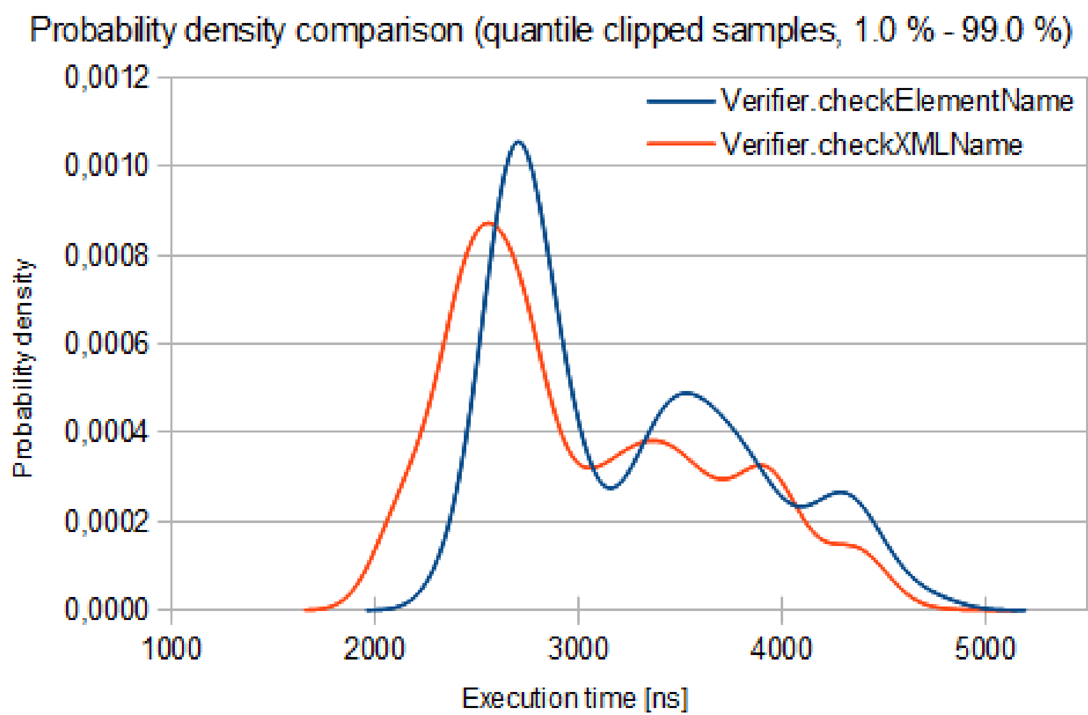


Figure 4.10: Time comparison checkElementName and checkXMLName.

Conclusion

This work describes the problem of capturing runtime performance assumptions of individual methods using SPL expressions. Contrary to existing SPL tools it aims to measure performance of a software which is actually used in a real environment.

This issue and its differences was closely analysed and possible solutions were proposed. That includes answers to the matter of automatic modification of the production application by adding and removing measuring code, distinguish size of the data which are used by measured methods to be able to compare methods with similar input data size, continuous processing of the measured data and controlling number of measured methods to avoid high overhead caused by measurement.

Number of technologies for code transformation was explored for the purpose to insert measuring code to the application and their main advantages and disadvantages were presented. DiSL is one of these technologies which was chosen and used to create a prototype in Java.

The prototype provides functionality of measurement and comparison of methods performance mentioned above. It allows different configuration to be used based on the application which should be measured, whether the measurement should be finished as soon as is possible or it may last longer but low overhead is required. It also makes possible for the user to create own processors of method input data so it is able to work with almost any data input and express its size as is required.

Usage of the prototype was demonstrated on two examples. First is a simple processor demanding sorting application and second uses a real-world tool ROME which works with real data downloaded over the Internet.

Because the prototype is just a command line tool, in the future it may be base for a work which adds graphic user interface and output that uses richer statistics. This will help for the user to better undertake this approach and add it into the process of developing and monitoring application software.

Bibliography

[1] Lubomír Bulej, Tomáš Bureš, Jaroslav Keznikl, Alena Koubková, Andrej Podzimek, and Petr Tůma. Capturing performance assumptions using stochastic performance logic. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, ICPE '12, pages 311–322, New York, NY, USA, 2012. ACM.

[2] Shigeru Chiba. *Javassist*, 2015. `http://www.csg.ci.i.u-tokyo.ac.jp/~chiba/javassist`.

[3] OW2 Consortium. *ASM - Home Page*, 2014. `http://asm.ow2.org`.

[4] The Eclipse Foundation. *The AspectJ Project*, 2015. `http://eclipse.org/aspectj`.

[5] Vojtěch Horký, František Haas, Jaroslav Kotrč, Martin Lacina, and Petr Tůma. Performance regression unit testing: A case study. In *Computer Performance Engineering*, volume 8168 of *Lecture Notes in Computer Science*, pages 149–163. Springer Berlin Heidelberg, 2013.

[6] Jason Hunter and Rolf Lear. *JDOM*, 2015. `http://jdom.org/`.

[7] IBM. *JPROF - Java Profiler*, 2010. `http://perfinsp.sourceforge.net/jprof.html`.

[8] Lukáš Marek, Yudi Zheng, Danilo Ansaloni, Lubomír Bulej, Aibek Sarimbekov, Walter Binder, and Petr Tůma. Introduction to dynamic program analysis with DiSL. *Science of Computer Programming*, 98, Part 1(0):100 – 115, 2015. Fifth issue of Experimental Software and Toolkits (EST): A special issue on Academics Modelling with Eclipse (ACME2012).

[9] Lukáš Marek, Yudi Zheng, Danilo Ansaloni, Aibek Sarimbekov, Walter Binder, Petr Tůma, and Zhengwei Qi. Java bytecode instrumentation made easy: The DiSL framework for dynamic program analysis. In *Programming Languages and Systems*, volume 7705 of *Lecture Notes in Computer Science*, pages 256–263. Springer Berlin Heidelberg, 2012.

[10] Oracle. *JVM Tool Interface*, 2014. `http://docs.oracle.com/javase/8/docs/platform/jvmti/jvmti.html`.

[11] *ROME - Home*, 2014. `http://rometools.github.io/rome`.

List of Tables

2.1 Technology overview. 33

4.1 Comparisons and their p-value. 62

List of Abbreviations

SPL Stochastic Performance Logic

JVM Java Virtual Machine

JAR Java Archive

JVM TI JVM Tool Interface

API application programming interface

XML Extensible Markup Language

SAX Simple API for XML

DOM Document Object Model

Attachment A

Attached CD contains text of this thesis and source code of the prototype described in it. Compiled prototype is not included, because it uses native code which has to be compiled for the target platform.

The content is:

README.TXT describes the CD content.

thesis.pdf is the text of this work.

code is folder with source codes of the prototype. Its content, compilation and running is described in *code/README.TXT*.

javadoc is folder with generated javadoc, root page is *javadoc/index.html*.

output is folder with example output generated by the prototype when running measurement of the ROME example. It contains these files:

- **measurements** is folder containing files with measured data.
- **result.eval** is the evaluation result.
- **rome.out** is output printed by the measured application.
- **spldisl.log** is the log of the measurement.