Institute of Software Technology

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelor's Thesis Nr. 140

Dynamic Instrumentation in Kieker Using Runtime Bytecode Modification

Albert Flaig

Course of Study: Softwaretechnik

Examiner: Prof. Dr. Lars Grunske

Supervisor: Dr.-Ing. André van Hoorn

Commenced: 2014/05/05

Completed: 2014/11/04

CR-Classification: D.4.8

Abstract

Software systems need constant quality assurance — this holds true in the development phase as well as the production phase. An aspect of quality is the performance of specific software modules. Kieker provides a framework to measure and diagnose runtime information of instrumented software methods. In its current state, Kieker only allows inserting probes before application start.

This thesis proposes an alternative concept to extend the functionality of Kieker regarding instrumentation. The alternative approach allows inserting probes during runtime. This is done using a technology known under the term Bytecode Instrumentation (BCI) which enables to change the binary code of classes during execution. Thus the software is "reprogrammed" during runtime to provide the measurement logic. The approach is carried over of another monitoring framework called AIM (Adaptable Instrumentation and Monitoring), which already features an established implementation of this technology. Hence, this thesis aims to connect the benefits of both frameworks.

This alternative concept is compared against Kieker's traditional way of performance measurement by the means of an experimental evaluation. The evaluation aims to investigate the impact on, (1) the overhead, (2) the turnaround time and (3) the reliability in terms of lost transactions. The results show a reduction of overhead, unfortunately at the cost of turnaround time. The reliability also drops due to an increase of lost transactions.

Zusammenfassung

Software-Systeme benötigen eine ständige Qualitätskontrolle — sowohl in der Entwicklungsphase als auch in der Produktionsphase. Ein Aspekt das die Qualität von Software ausmacht, ist die Performanz bestimmter Software-Module. Kieker bietet an dieser Stelle ein Framework an, um Laufzeitdaten von instrumentierten Methoden messen und auswerten zu können. Im aktuellen Stand können Messsonden (Monitoring Probes) in Kieker lediglich vor Programmstart eingesetzt werden.

Diese Thesis zeigt ein neues Konzept das die Funktionalität von Kieker in Bezug auf die Instrumentierung erweitert. Dieser alternative Ansatz erlaubt es Messsonden während der Laufzeit einzusetzen. Dies wird mit einer Technologie umgesetzt die bekannt ist unter dem Begriff Bytecode Instrumentation (BCI). Sie ermöglicht den Binärcode von Klassen während der Ausführung zu verändern. Somit wird die Software mit der Logik der Laufzeit-Messung versehen, indem sie zur Laufzeit „neu programmiert" wird. Der Ansatz wurde aus AIM (Adaptable Instrumentation and Monitoring), einem weiteren Monitoring Framework, übernommen. Dieses weißt eine bereits bestehende Implementierung dieser Technologie auf. Daher zielt diese Thesis darauf ab die Vorteile beider Frameworks zu verbinden.

Der Overhead von diesem alternativen Konzept wird verglichen mit der herkömmlichen Art die Kieker nutzt um Performanz zu messen. Der Vergleich wird mithilfe einer experimentellen Evaluation durchgeführt. Die Evaluation untersucht die Auswirkungen auf (1) den Overhead, (2) die Turnaround-Zeit und (3) die Zuverlässigkeit in Hinsicht auf Lost Transactions. Die Ergebnisse zeigen eine Verringerung des Overheads, leider auf Kosten der Turnaround-Zeit. Die Zuverlässigkeit sinkt ebenfalls aufgrund einer erhöhten Anzahl an verloren gegangenen Transaktionen.

Contents

Introduction

This chapter will give a brief overview on this thesis by giving a motivation (Section 1.1), listing the achieved goals (Section 1.2), and outlining the document structure (Section 1.3).

1.1 Motivation

Modern software needs a means to be monitored continuously in order to assure a certain standard of quality, e.g., in terms of stability and error detection rate. Approaches within the development phase are quite overhead-heavy on the tested software and can influence the test result. However, this overhead can be tolerated as any revealed malfunctions can usually still be fixed within development phase. This does not apply to the production phase of the product. In this phase, software needs to be as stable as possible. Any overhead can be fatal in this phase. Therefore, approaches in the production phase must be a lot more lightweight in terms of overhead. The monitoring software Kieker [Kieker Project, 2014; Rohr et al., 2008; van Hoorn et al., 2009, 2012] is one such approach which is open source and emphasizes on low overhead. It provides an adaptive monitoring feature which can turn on and off monitoring capabilities on certain components within the software depending on the needs. Thus the overhead can be determined on a per-component-basis and regulated to the very minimum.

However, this feature needs the components to be pre-instrumented. No new methods can be added dynamically during runtime. Also the pre-instrumented methods still have a check whether to execute monitoring or not, which also consumes a certain amount of computing power. This thesis aims to investigate an alternative approach for dynamic instrumentation and conducts a quantitative comparison between both approaches to resolve these drawbacks. The proposed approach is to use dynamic bytecode modification of the classes to inject the monitoring code during execution time of the software. Thus monitoring capabilities can be turned on and off on-demand. Normal execution will not be altered in any way as long as the monitoring capabilities are turned off.

1.2 Goals

This thesis consists of two parts: (1) the development of dynamic instrumentation support using AIM in Kieker and (2) a quantitative comparison between the traditional approach

in Kieker and the proposed alternative approach.

Development of Dynamic Instrumentation Support for Kieker

The first goal is to develop the dynamic instrumentation support by integrating Kieker into AIM . AIM is a similar monitoring framework like Kieker with emphasis on different aspects, which is covered in detail in Chapter 2 and Chapter 4. However, it possesses the capability to dynamically instrument Java applications during runtime through BCI. Its approach is transfered to Kieker in the development phase. Changes to the existing API in Kieker should be kept at a bare minimum. Optimally, the API would not change at all. The research questions to be investigated are: How can Kieker be integrated into AIM? How efficiently can it be integrated without increasing complexity or changing the architecture too much? How will the dynamic instrumentation behave on certain special cases?

Experimental Evaluation

The second goal is to assess the usefulness of the developed methods in comparison to Kieker's existing approach. For this purpose, an approach is worked out to conduct the evaluation. The following questions should be answered in the evaluation: Is there going to be a significant improvement in terms of overhead? Is the approach reliable? How does it behave in terms of scalability?

1.3   Document Structure

The remainder of this document is structured as follows.

Chapter 2 (Foundations and Technologies) presents vital background information on foundations and technologies which are essential for the comprehension of this thesis. Chapter 3 (Related Work) shows similarities with related work and differences. Chapter 4 (Development of Dynamic Instrumentation Support for Kieker) describes the established architecture of the development phase as well as challenges, limitations, and pitfalls. Chapter 5 (Experimental Evaluation) describes the evaluation process and its results. Chapter 6 (Conclusion) discusses the result of this thesis and future work.

Foundations and Technologies

This section gives a brief overview of concepts and technologies that will be used in this thesis. Section 2.1 gives an overview of Kieker's architecture its capabilities and strengths. Section 2.2 explains the principle behind dynamic instrumentation in Java and presents the techniques needed to achieve that. Section 2.3 gives brief information about the origin of AIM as well as its abilities and differences to Kieker.

2.1   Kieker Monitoring Framework

Kieker, is a Java-based monitoring framework [Kieker Project, 2014; Rohr et al., 2008; van Hoorn et al., 2009, 2012]. It is extensible and developed in regard to low overhead. Kieker consists of two main components: Monitoring and Analysis. Figure 2.1 shows their relationship. This thesis concentrates on the Monitoring component of Kieker that is responsible for instrumentation and data gathering. The Monitoring Probes collect data, e.g., execution time and control flow of instrumented methods. Figure 2.2 shows an example of control flow data which has been visualized by the Kieker Analysis component.
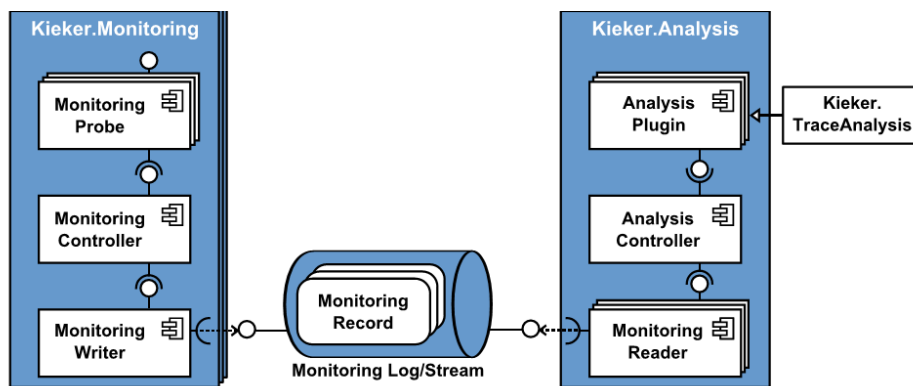

Figure 2.1. Overview of the Kieker framework components [Kieker Project, 2014]

The Monitoring Controller employs the JMX interface, controls the monitoring probes and passes the collected data to the Monitoring Writer. The monitoring writer writes the
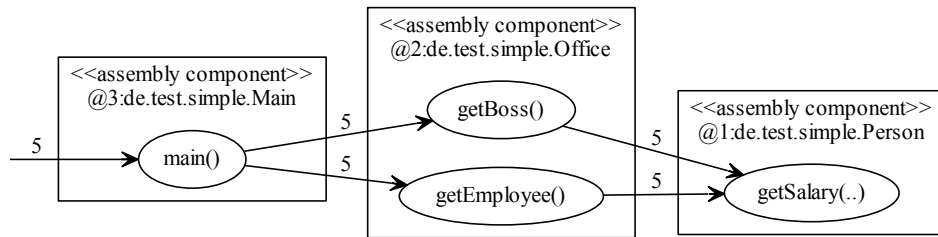
Figure 2.2. An operation dependency graph as generated by the Kieker Trace Analysis component showing various runtime information like execution count and call hierarchy. A similar illustration is depicted in the Kieker userguide [Kieker Project, 2014]

collected data to the stream which can be a variety of different outputs like file system, JMX, or a database.

Kieker does also provide samplers. Sampling works inherently different from instrumentation. Basically the stack trace is fetched in periodic time intervals, usually within range of several dozen nanoseconds. Analyzing the data, one can make out which methods the thread is executing most frequently. Thus a metric is presented showing the performance of methods. This approach needs no tinkering with the classes and can be used to find rough performance problems. However depending on the scan interval it may or may not miss methods [Heiss, 2005].

The next subsections will go more in detail to the inner workings of monitoring probes and controlling them via Kieker's adaptive monitoring API.

2.1.1 Monitoring Probes

Monitoring Probes can be deployed through various techniques. The most simple form is manual insertion. However the more advanced (and useful) techniques employ interception techniques provided by AspectJ [Asp], Java EE Servlet [Oracle, 2011], Spring [SpringSource, 2011] and Apache CXF [The Apache Foundation, 2011]. Listing 2.1 shows the most simple usage of AspectJ by the use of annotations.

The detailed procedure of a general monitoring probe can be seen in Figure 2.3. Each probe has a before part and an after part. Both parts register the current time which get passed on into a new monitoring record which in turn is then send to the monitoring controller. Having the time of the start and the end of the method body, one can calculate the execution time the method demands. However, the method `collectData()` not only collects execution data but also other data like the signature of the monitored method, the trace id, tree data, etc. The actual data collected depends on the type of probe that has been deployed.
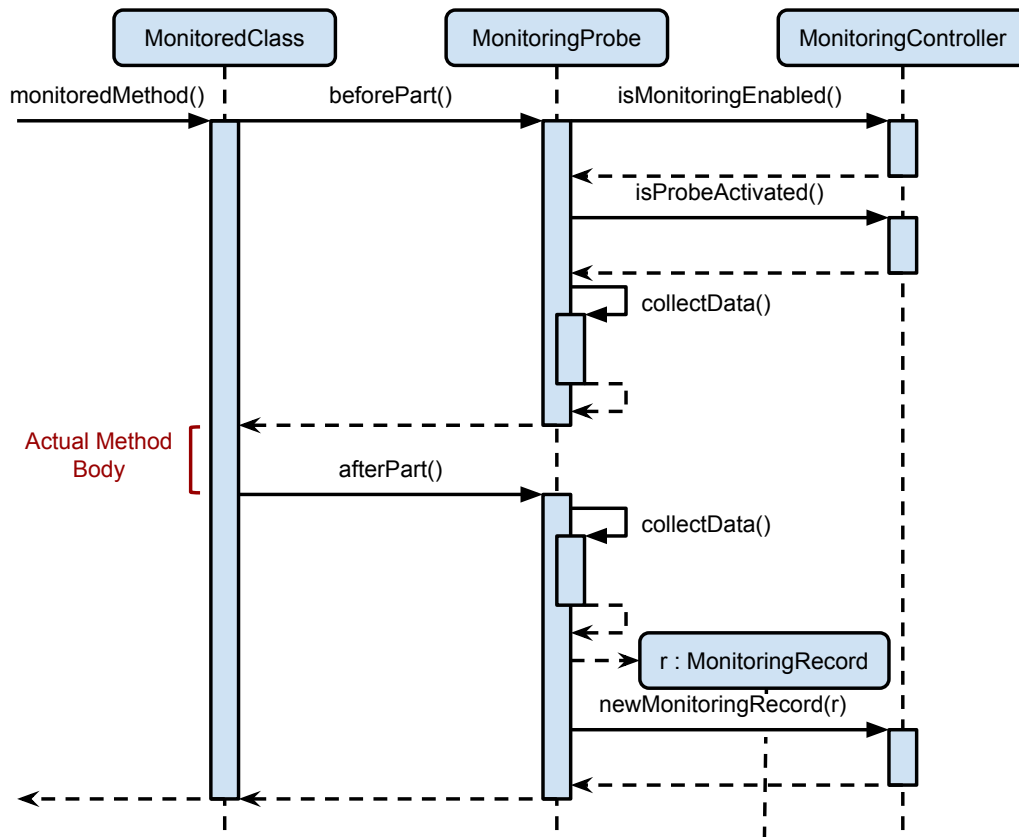
Figure 2.3. Sequence of a deployed Monitoring Probe Waller and Hasselbring [2012]

Kieker provides two concepts to deploying monitoring probes: Aspect Oriented Programming (AOP) and manual instrumentation of classes. The term Aspect Oriented Programming is the technique to describe a certain behavior which is common to a set of specific classes or components. In this case the behavior is to collect execution data. Once the monitoring probes have been deployed they can be activated and deactivated dynamically during runtime using Kieker's Adaptive Monitoring API. In the following listing one can see how to deploy a probe using the concept of AOP. In this case we use the implementation of AspectJ in Kieker.

The usage is simple. The monitored method is simply annotated with the specific annotation. Aspectj weaves the corresponding probe. More on this can be looked up in the user guide [Kieker Project, 2014]. The next listing shows an equivalent form of deploying a probe by manual insertion.

```
3      private String name;
4
5      @OperationExecutionMonitoringProbe
6      public String updateProduct(int productId) {
7         String product = Products.get(productId);
8         name = product;
9         return name;
10     }
```

Listing 2.1. Monitored Class and its Monitored Method instrumented with AspectJ

```
3      private static final MC = MonitoringController.getInstance();
4
5      public String updateProduct(int productId) {
6         // Before Part
7         final boolean instrumented = MC.isMonitoringEnabled() && MC.isProbeActivated();
8         if(instrumented) {
9            data = collectDataBefore();
10        }
11        // Actual Method Body
12        String product = Products.get(productId);
13        // After Part
14        if(instrumented) {
15           data = collectDataAfter(data);
16           MC.newMonitoringRecord(new MonitoringRecord(data));
17        }
18        return product;
19     }
```

Listing 2.2. Pseudo Java Code of a manually instrumented method. Semantically equivalent to Listing 2.1.

2.1.2 Kieker's Adaptive Monitoring API

Adaptive monitoring in Kieker works by providing the ability to select which components of the software have to be monitored. This can be selected very fine granularly. However these components (currently) still have to be pre-instrumented. When using AOP one can easily deactivate certain probes using the Monitoring Controller. Providing a pattern to the deactivateProbe(...) or activateProbe(...) methods will activate or deactivate the probes matching that specific pattern. Listing 2.2 shows a call to the isProbeActivated() which reflects whether the specific probe is activated or not and thus whether to collect monitoring data or not. How such a pattern should look like is described in the kieker.monitoring.adaptiveMonitoring.conf which is included in the Kieker binary release. As the Monitoring Controller provides an interface to JMX, one can activate and

deactivate specific monitoring probes locally or remotely with an external tool. The concept will be described in-depth in Chapter 4.

Additionally to the Monitoring Controller Interface there is a second method to activate and deactivate certain probes. That goes by modifying the `kieker.monitoring.adaptiveMonitoring.conf` file. Kieker reads this file in regular intervals and synchronizes its internal patterns with the content. The interval and the used file for adaptive monitoring can be configured. More information is available in the Kieker User Guide [Kieker Project, 2014].

2.2 Dynamic Instrumentation

Dynamic instrumentation describes the ability to instrument classes and their method bodies while the application is running by the means of bytecode modification. However to achieve that a few techniques are needed. For example, Java provides means to introspect data structures within classes like methods, fields, or class members. This is called the reflection mechanism. However it lacks the ability to modify the behavior of these classes. Therefore, many different libraries exist which allow direct modification of the bytecode of Java classes. Examples are SERP [ser], ASM [asm] and Javassist [jav]. ASM and SERP can instrument methods when provided with the direct Java bytecode snippet. On the other hand javassist supports instrumentation provided by source code snippets. Various libraries support one or both of these methods. As AIM uses Javassist for bytecode modification, this library will be focused on right now.

2.2.1 Javassist

Javassist [jav] is a powerful library which has many uses. Its primary function is however to provide means for runtime BCI. It has support for both methods of bytecode modification: Injecting direct Java bytecode and providing Java source files which are compiled on the fly and subsequently injected. Listing 2.3 shows the usage of the library.

```
 9  public class HelloWorld {
10      public static void main(String[] args) throws Exception {
11          CtClass cc = ClassPool.getDefault().get("de.test.HelloWorld$SayHello");
12          CtMethod cm = cc.getDeclaredMethod("printHelloWorld");
13          cm.insertAfter("{ System.out.println(\"Hello World!\"); }");
14          cc.toClass();
15          SayHello inst = new SayHello();
16          inst.printHelloWorld();
17      }
18      static class SayHello {
19          public void printHelloWorld() { }
20      }
21  }
```

Listing 2.3. Most simple application to demonstrate the javassist BCI.

Notice that there is no class reloading used in the example. It relies on the fact that the class `SayHello` is not loaded until `cc.toClass()` is executed, which completes the BCI.

2.2.2 Hotswap

Even though we can modify classes during runtime, we cannot simply exchange classes which reside in working memory. The Hotswap technique is needed for that. It is part of the Java Instrumentation API which has been introduced in Java JDK 5.0 [Oracle, a]. Java Hotswap works by swapping classes which reside in the JVM with a new class. The class definition in itself has to be provided as bytecode. The current Hotswap implementation in the JVM supports only swapping of method bodies. After a Hotswap occurred, only the latest class version will be used. The old class version ceases to exist within the JVM working memory. At this point it will be important to evaluate what exactly will happen when a class needs to be swapped which is currently executing. Listing 2.4 shows the previous example modified with a Hotswap execution. Notice however, the instrumentation attribute has to be set by deploying a javaagent. See also [jav, 2006].

```java
14  public class HelloWorld {
15      public static Instrumentation instrumentation;
16
17      public static void main(String[] args) throws Exception {
18          CtClass cc = ClassPool.getDefault().get("de.test.HelloWorld$SayHello");
19          CtMethod cm = cc.getDeclaredMethod("printHelloWorld");
20          cm.insertAfter("{ System.out.println(\"Hello World!\"); }");
21          Class<?> c = cc.toClass();
22          ClassDefinition cd = new ClassDefinition(c, cc.toBytecode());
23          instrumentation.redefineClasses(cd);
24          SayHello inst = new SayHello();
25          inst.printHelloWorld();
26      }
27      static class SayHello {
28          public void printHelloWorld() { }
29      }
30  }
```

Listing 2.4. Most simple application to demonstrate Hotswap with javassist.

2.3 AIM Framework for Adaptive Instrumentation and Monitoring

AIM is an abbreviation of Adaptable Instrumentation and Monitoring. It has been developed by researchers at the Karlsruhe Institute of Technology and has recently been released as open source software [Wert and Heger, 2014]. Like Kieker it is also a monitoring

framework. However the internal architecture and external interface differ from Kieker fundamentally. Most importantly, AIM has capabilities for dynamic bytecode instrumentation while Kieker has not. Furthermore, AIM is equipped with its comprehensive instrumentation description. However Kieker shines with its extensive adaptive monitoring regular expression matching which is particularly based on the use wildcards. Further, Kieker is a monitoring framework which has been developed to be highly extensible. Kieker also possesses strong analysis capabilities, however that is out of scope of this thesis.

In contrast to Kieker, AIM strictly isolates the application under test from the main monitoring and instrumentation component by having a server-client model. Thus the usage is inherently different from Kieker. The application under test is started with a specific command line argument which fires up the AIM Service Interface. This interface acts as a server whilst the main monitoring and instrumentation component acts as a client. The client can issue instrumentation commands to the server and receive monitoring records. Section 4.1 also describes more similarities and differences especially in terms of internal architecture.

Related Work

This Chapter discusses the work that has been done in the field of performance measurement. Section 3.1 discusses state of art tools which are currently used most commonly whilst Section 3.2 focuses on research done in this field and distinguishes it from this thesis.

3.1 State of the Art

In the field of performance measurement, dynamic instrumentation is not new. VisualVM for instance, a profiling tool which comes with the JDK straight off the bat, provides a sampler and a profiler capable of dynamically instrumenting classes. Its dynamic instrumentation capabilities are based on the NetBeans profiler [Sedlacek, 2008]. The NetBeans profiler is in turn based on the work of JFluid [Dmitriev, 2003; Heiss, 2005]. VisualVM however has a trimmed down version of the NetBeans profiler. It is not very flexible as it cannot select to instrument specific methods of certain classes. A class can either be instrumented wholly or not at all. Kieker is more flexible in this regard because of its adaptive monitoring API.

BTrace provides a plugin for VisualVM which enables more flexible and extensive dynamic instrumentation capabilities. BTrace, like AIM, possesses the ability to instrument during runtime using BCI.

3.2 Distinction

Little research has been done in comparing compile-time instrumentation and dynamic instrumentation in Java. Although there has already been a hybrid approach [Iskhodzhanov et al., 2013], it is not focused on Java.

Kieker is a monitoring framework which has been focused on in research papers since its creation. An evaluation of overhead of the Kieker monitoring framework has already been conducted several times [Ehlers et al., 2011; Ehlers and Hasselbring, 2011; van Hoorn et al., 2009; Waller and Hasselbring, 2013, 2012]. However none of these have evaluated Kieker in accordance with dynamic BCI.

A dynamic BCI support for Kieker has already been developed by [Wert, 2012; Heger, 2012]. An experimental evaluation has also been done by [Wert, 2012]. However, Kieker

has since evolved and has gotten the adaptive monitoring API with release 1.6 [Kieker Project, 2014].

This thesis therefore aims to investigate the differences between compile-time instrumentation and dynamic instrumentation. Kieker does compile-time instrumentation with the help of AspectJ. However, AspectJ has little to no overhead compared to manual instrumentation [Dufour et al., 2004; Hilsdale and Hugunin, 2004]. Hence, the investigation on the compile-time instrumentation is rather done on manually instrumented methods. The dynamic instrumentation part is something Kieker lacks and needs to be developed. Afterwards the evaluation will take place.

Development of Dynamic Instrumentation Support for Kieker

This chapter will describe the developed dynamic instrumentation support for Kieker. The goal is to incorporate the dynamic instrumentation functionality of AIM into Kieker. Therefore an overview is created to get a general idea of the differences and similarities between AIM and Kieker. The first step is to identify the part of AIM which employs the wanted functionality. The second step is to compare the architecture of both frameworks and identify the point at which existing components are to be modified and new components are to be created. An overview of these steps is shown in Section 4.1 and described in detail in Section 4.2. Section 4.3 describes challenges and limitations of the developed dynamic instrumentation support.

4.1   Similarities and Differences between AIM and Kieker

The AIM Client sends an Instrumentation Description to the AIM Service Interface. The Instrumentation Description contains various information about the experiment in question. It is like a request to the Application under Test to query specific data about it. Included is information about to be deployed probes, its scopes, certain samplers, specific restrictions, and some JVM events to be used in the experiment. Samplers fetch periodically in certain time intervals data about the underlying system like disk and memory usage, CPU utilization, etc. The locations of probes are described within so-called Scopes. Kieker's corresponding feature is the use of patterns within the Adaptive Monitoring API. AIM has certainly a lot of potential in gathering data.

However most of this functionality is not needed to integrate the Dynamic Instrumentation support in Kieker. Figure 4.1 shows the components which are relevant to this thesis. It depicts the related components for dynamically deploying monitoring probes in AIM. The AIM Client can roughly be translated to the JMX interface in Kieker. Both can be used to determine which methods are to be instrumented. AIM uses Scopes for this, while Kieker uses patterns. The AIM Client sends the Instrumentation Description to the Instrumentation Agent. The Instrumentation Agent acts as a Server within the Application under Test. Unlike Kieker, AIM uses two separate processes on the operation system level. Receiving Instrumentation Descriptions the Instrumentation Agent transfers the command

to the Method Instrumentor. The Method Instrumentor uses the Java Instrumentation API to get a list of all currently loaded classes in the JVM. Having this list, it evaluates which classes are to be instrumented. The resulting sublist is forwarded to the BCInjector. The BCInjector uses the Probe Snippets which are built from the Probe Builder and instruments the given classes using Javassist. The resulting classes are then forwarded to the Java Instrumentation to be hotswapped in the JVM. AIM is, however, unable to instrument classes which are not yet loaded in the JVM.
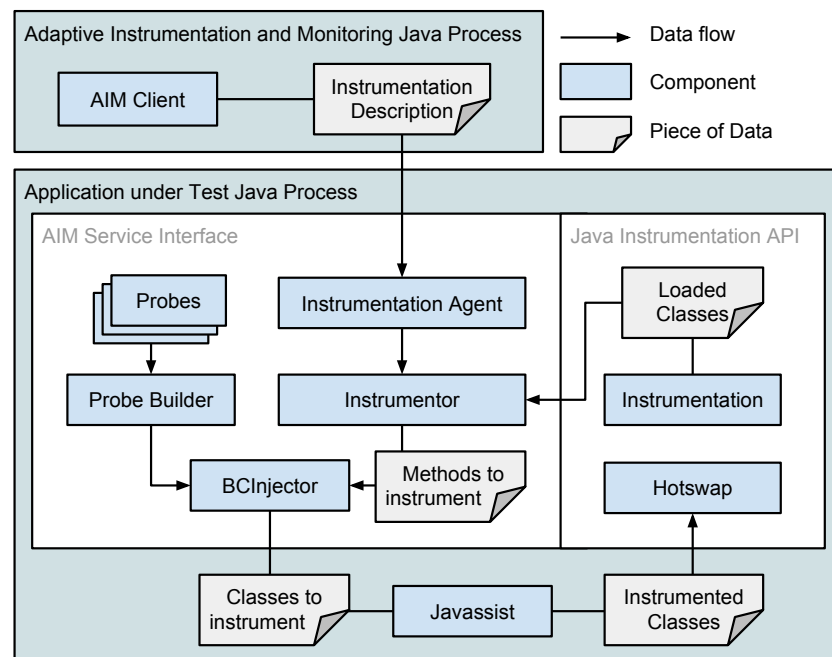
Figure 4.1. Overview of the components involved for BCI in AIM.

Looking at the architecture of Kieker, one can see a few similarities to AIM. Especially when considering the case of using Kieker through the JMX Interface (Figure 4.2). The Monitoring Controller consists of several components, each having a distinct function. The JMX Controller initializes the MBeanServer for accessing the Monitoring Controller through the JMX Interface. When sending a Pattern to the JMX Controller the call is delegated to the Probe Controller. The Pattern may be either an activating or a deactivating one. Listing 4.1 shows examples of activating and deactivating patterns. The Probe Controller adds the new pattern to its managed list of patterns.

Unlike AIM, Kieker needs the classes of the Application under Test to be preinstrumented. This can be done upon startup of the application using Aspect Oriented Programming (AOP) or the instrumentation code can be inserted manually to the source code before
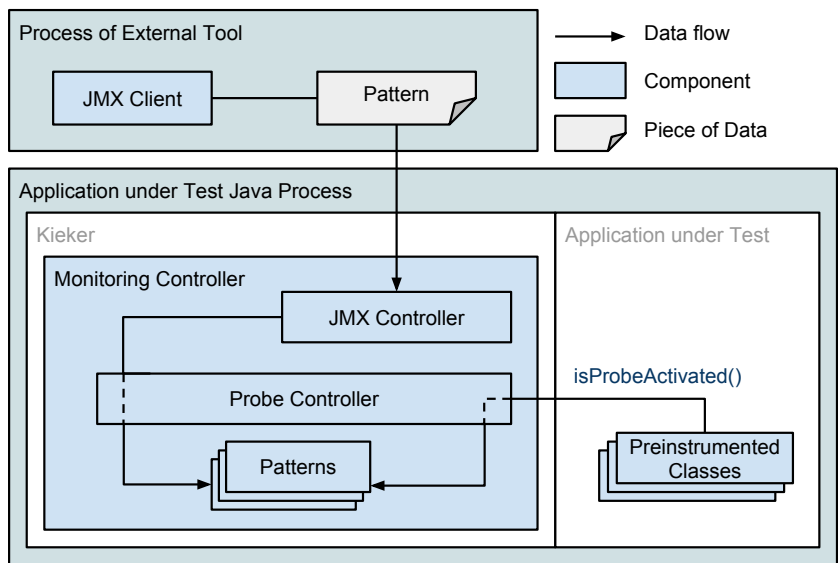
Figure 4.2. Overview of the components involved for enabling a Probe using Adaptive Monitoring through the JMX Interface in Kieker.

```
+ org.apache.commons.io.CopyUtils.*(..)
− org.apache.commons.io.DirectoryWalker.*(..) throws Exception
+ org.apache.commons.io.EndianUtils.*(..)
+ org.apache.commons.io.FileCleaner.*(..)
+ org.apache.commons.io.IOUtils.*(..)
− public static byte[] org.apache.commons.io.IOUtils.toByteArray(java.io.InputStream)
```

Listing 4.1. Examples of Instrumentation Patterns to feed Kieker's Adaptive Monitoring API.

compilation. Each probe checks whether it is activated using the Monitoring Controller. This sequence has already been depicted in Figure 2.3. The call is delegated to the Probe Controller. The Probe Controller sequentially checks the signature of the probe with all patterns in the pattern list from the newest to the oldest. The first match will determine whether the probe is activated or not, depending whether the pattern was an activating or deactivating one.

All in all one can highlight these differences in both frameworks:

▷ AIM does not have as an extensive pattern matching system like Kieker. The instrumentation description allows for simple regular expression matching though.

▷ The instrumentation in AIM is on demand. The monitoring code is injected when the

probe is activated and removed when deactivated.

▷ Kieker has the monitoring code in the application since startup and checks whether to execute it or not depending whether the probe is activated or not.

▷ AIM does not manage a history of instrumentation descriptions, while Kieker does manage a history of patterns in the pattern list.

▷ AIM cannot instrument classes which are not yet loaded in the JVM. Neither statically (at startup) nor dynamically (during runtime). The reason is that the Java Instrumentation API does not provide a list of classes residing in the classpath but rather only the classes loaded in the JVM.

4.2 Approach

Comparing Kieker's architecture against the architecture of AIM the differences can be highlighted. Subsequently the point at which to start the development can be deducted. The approach is segmented in two parts: The First Implementation and the Second Implementation. The First Implementation is a proof of concept prototype, whilst the Second Implementation mitigates the pitfalls and limitations introduced in the First Implementation.

4.2.1 First Implementation

In AIM the crucial component is the *BCInjector*. It takes two inputs: An AIM instrumentation description (describes which methods of which classes are to be instrumented) and a corresponding AIM probe.

For the first input a new monitoring probe needs to be developed which mimics the interface to the *BCInjector*, however still reports its collected data to the Kieker *Monitoring Controller*. On the other hand Kieker's probe instrumentation patterns need to be translated to AIM instrumentation descriptions and be used as the second input to the *BCInjector*. To achieve that the *Instrumentor* needs to be rewritten to be compatible with Kieker's adaptive monitoring API. In other words, the rewritten component needs a direct interface to the *Probe Controller*.

So in the first attempt we have two new components: (1) one component equivalent to AIMs *Instrumentor*, onward called *KiekerInstrumentor*, and (2) another to be compatible to the interface to the *BCInjector*, onward called *OperationExecutionProbe*. The pattern list which is used by the Probe Controller is discarded. Figure 4.3 illustrates the first attempt.

The process is now altered that the instrumentation pattern is directly delegated to the *KiekerInstrumentor* without saving it anywhere. *KiekerInstrumentor* receives all loaded classes from the Java Instrumentation API and checks each class against the pattern. The resulting list represents the classes which are to be instrumented. This list is forwarded
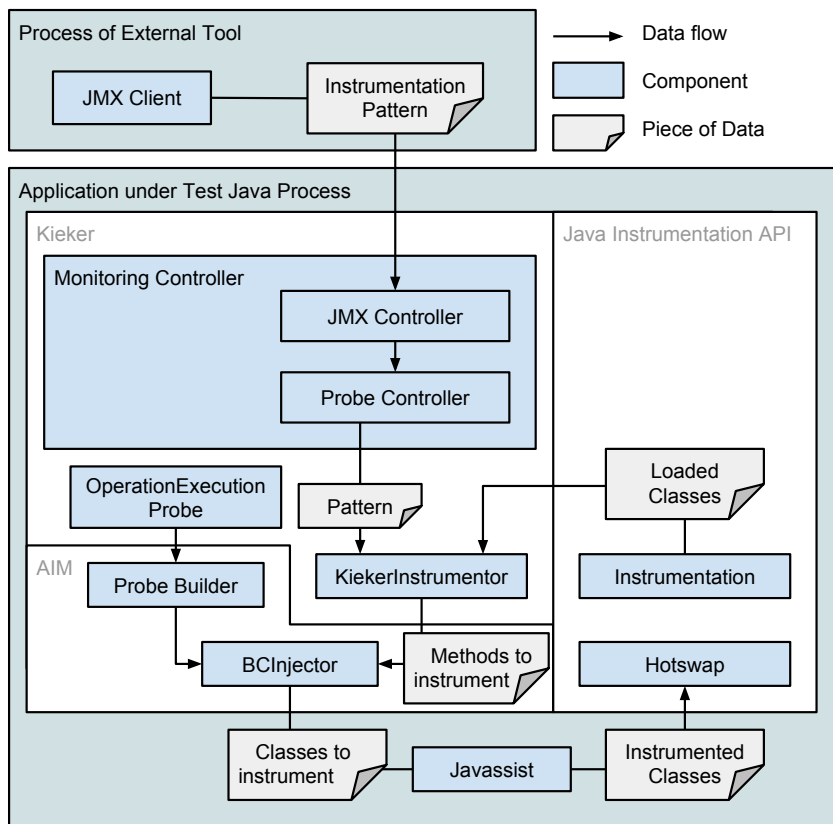
Figure 4.3. First attempt integrating Kieker into AIM. Two new components are introduced.

to the *BCInjector*, Along with the code snippet of the *OperationExecutionProbe* that is built through the *Probe Builder*. Having both inputs, the *BCInjector* calls Javassist to instrument the classes and consequently hotswap them in the JVM.

In the end this attempt proved to be incorrect. Take for example Listing 4.1 which was shown above. If the user inputs these lines sequentially from top to bottom there will be an error. The next to the last line tells *KiekerInstrumentor* to instrument the class `org.apache.commons.io.IOUtils` and all of its methods. The last line tells *KiekerInstrumentor* to not instrument one specific method. The expected result would be that all methods in `IOUtils` are instrumented except for the specific one. However the actual result is that no classes are instrumented at all. Firstly Kieker does indeed instrument all methods correctly. However having no memory of previous instrumentation patterns, all instrumented methods of the class are discarded as the original class definition is used for dynamic BCI.

The next problem in this attempt is the lack of the ability to instrument classes which

are not yet loaded in the JVM. This is due to a limitation in the Java Instrumentation API. It does neither provide an interface to get all class definitions within the classpath, nor does it allow to hotswap classes which are not yet loaded. If it did provide that functionality it would be a contradiction to the Java Virtual Machine Specification though [Lindholm and Yellin, 2013]. The Java Virtual Machine loads classes lazily, only as needed. So to eliminate this drawback a component needs to be established which listens on every class load the JVM conducts. This is introduced in Section 4.2.2.

4.2.2 Second Implementation

The second attempt tends to eliminate the two drawbacks from the previous attempt: (1) To correctly instrument classes depending on the instrumentation patterns and (2) to allow for instrumentation of classes which are not yet loaded. The resulting process is depicted in Figure 4.4.

Correct Instrumentation

To deal with the first objective, the Kieker patterns list has to be reintroduced. Whenever a new pattern (activating or deactivating) the *Probe Controller* sends a request to the *KiekerInstrumentor* to reinstrument all loaded classes. No matter the pattern incoming, the *Probe Controller* always sends the whole pattern list to the *KiekerInstrumentor*. The *KiekerInstrumentor* in turn evaluates all loaded classes against the pattern list to determine the classes which are to be instrumented. This is done by checking each method signature of each class against the whole pattern list.

The chronology of the instrumentation pattern list is crucial at this point. Each method signature is checked against every pattern in the pattern list in historical order, from the pattern added most recently to the oldest one. The first match determines whether to instrument that specific method or not, depending on whether the matched pattern was an activating or deactivating one. For example, having one entry in the pattern list:

```
+ org.apache.commons.io.IOUtils.*(..)
```

This clearly instruments all methods in the class *IOUtils*. Now adding another entry (which deactivates all methods in this class starting with *to*) makes it obvious that the newest pattern is the most relevant one and has higher priority than older ones.

```
+ org.apache.commons.io.IOUtils.*(..)
− org.apache.commons.io.IOUtils.to*(..)
```

A class which does not have any methods activated is not going to be instrumented. The resulting list of methods is as usually propagated to the BCInjector and then instrumented, etc.

Instrumentation of classes not yet loaded

The second objective involves developing a new component to listen for whenever the JVM loads a new class. This call is to be intercepted and propagated to the *KiekerInstrumentor*. However, Java does not provide such an interface. Conventionally there is no way to do that.

So a custom class loader is developed. The custom class loader, onward called *Kieker-ClassLoader*, features a listening pattern to enable the missed functionality. The application under test is started with a specific command line argument to JVM to specify to use the said class loader. The *KiekerClassLoader* listens on class loads and propagates the call to the parent class loader (Figure 4.4).
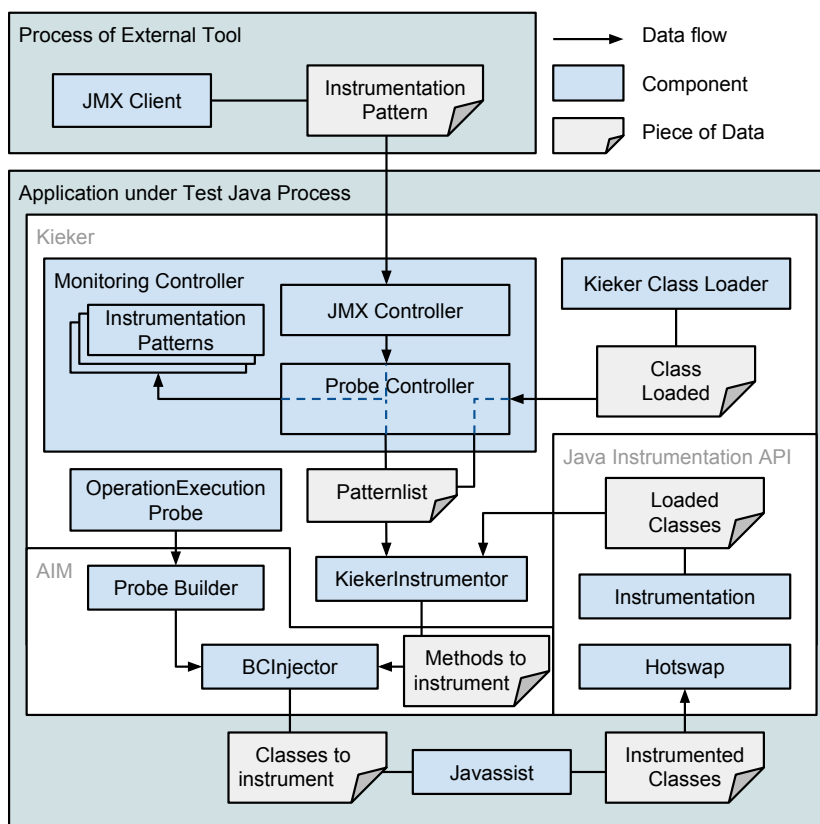
Figure 4.4. Second attempt integrating Kieker into AIM. Instrumentation pattern list is restored and *KiekerClassLoader* introduced.

However using propagation to the parent class loader will not work as hoped. The reason lies within the way Java class loaders work. Chapter 5.3 of the Java Virtual Machine

Specification says: [Lindholm and Yellin, 2013]

> Class or interface creation is triggered by another class or interface D, which references C through its run-time constant pool. [...] If D was defined by the bootstrap class loader, then the bootstrap class loader initiates loading of C.

In other words if the *KiekerClassLoader* propagates the event to the parent class loader it will not get word of any classes referenced by the loaded class. E.g., if a small application consists of three classes: *Main*, *Office* and *Person*. The *Main* class starts and calls *Office* which in turn calls *Person*. In this case the only class the *KiekerClassLoader* would get word off is the *Main* class. The *Main* class would have been defined by the parent class loader and its references would not even know about *KiekerClassLoader*.

So the solution is to write a custom class loading code. The class definition is read from the file system and then defined in *KiekerClassLoader*. The path on the file system is determined by the package of the to be loaded class. This method has also its drawbacks: Having a custom class loading code which overwrites the native class loading code is not stable.

As of now the *KiekerClassLoader* does not load classes which reside in jar files. In order to achieve that the classpath has to be parsed and every jar file in the classpath needs to be searched. Such an operation has a heavy impact on performance. So the paths of the various jar files need to be cached to reduce the performance impact which, however, increases complexity. Thus the additional functionality has not been implemented. A further information is discussed in Future Work (Section 6.3).

The *KiekerClassLoader* successfully delegates class loading events to the *Probe Controller* now. Each time a new class is loaded by the JVM, the *Probe Controller* sends another request to the *KiekerInstrumentor*. However, in this case it is known that only a single class needs to be reinstrumented. Thus there is no need to fetch all loaded classes from the Java Instrumentation API. Nevertheless, the pattern list still needs to be forwarded to the *KiekerInstrumentor* to ensure whether the class actually needs to be instrumented or not.

Using this attempt both pitfalls have been overcome — yet with deficiencies. Still this attempt clearly shows a proof of concept.

4.3   Challenges and Limitations

Kieker's adaptive monitoring architecture was clearly not designed for dynamic instrumentation support. Having to check all loaded classes and each of their methods against all instrumentation patterns in the pattern list of the *Probe Controller*, has a drastic impact on the turnaround time. For this reason a simple heuristic has also been developed which splits the instrumentation patterns into two parts: A class instrumentation pattern containing the fully qualified class name, and the rest of the pattern. This allows to check for the fully qualified class name and not further check each and every method of this class if the class name does not even match. A better heuristic is discussed in Section 6.3.

As already explained the class loader is a custom implementation that is not stable. Normally a class loader should ask its parent class loader whenever a class is to be loaded. However this custom implementation does not follow this convention and instead first checks if it can load the class itself and only if it fails it will delegate the call to the parent class loader. Applications may put a precondition on a clean class loader implementation to work.

Interesting to note, is that running methods are not instrumented and not altered in any way. New executions will be executed on instrumented classes, however the classes which are currently executing will stay alive until no method of it is executing anymore. This behavior is also described in the javadoc of the Java Instrumentation API [Oracle, b].

Experimental Evaluation

This chapter will describe the experiment that is conducted as well as any results which have come forth. The goal is to evaluate the usefulness of the developed method. The Goal Question Metric (GQM) approach is used to determine the needed metrics in Section 5.1. In Section 5.1.2 MooBench, an overhead measuring tool, is introduced. In Section 5.1.1 a suitable strategy is selected on how to conduct the experiment. Section 5.2 conducts the experiment.

5.1 Approach

The developed method is assessed in terms of usefulness. The GQM approach is used to achieve that goal [Ludewig and Lichter, 2007]. The established research questions are:

▷ **RQ-1**   How does it fare in terms of performance?

▷ **RQ-2**   Does it scale well?

▷ **RQ-3**   Is it reliable?

Figure 5.1 shows the resulting metrics which are raised in order to answer the research questions. How well the approach performs and how well it scales is measured by the overhead and the turnaround time. Its reliability is measured with the amount of lost transactions.

The data for the metrics is gathered by performing experiments. Each experiment is performed for both approaches, BCI and manual instrumentation. Finally the metrics are established by the use of the data and compared against each other to see whether an improvement can be observed.

5.1.1 Strategy

This section uses the guide on choosing the kind of test depicted by Field and Hole [2014]. In the experiment, data is gathered which resembles a score on how well the approach behaves. The data is a unit of time which is a ratio type of data and also parametric. The experiment for determining the overhead has only one independent variable which is also the condition: Either conducting the experiment trough BCI or trough manual
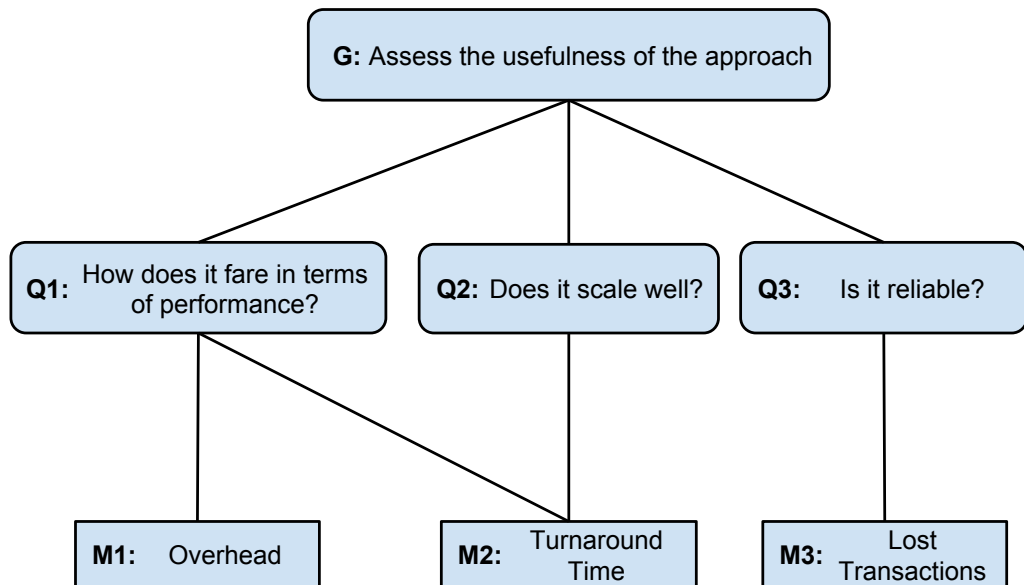
Figure 5.1. GQM plan used to achieve the evaluation goal [Ludewig and Lichter, 2007]

instrumentation. So according to Field and Hole [2014] a repeated measures t-test is conducted.

The t-test is conducted with the help of a calculator [Andreß, 2003].

5.1.2 MooBench - Overhead Measurement

There is a need for a tool which conducts the experiment and gathers the needed data. MooBench has a lot of potential in this field. It specializes in the field of measuring overhead of monitoring tools [Waller and Hasselbring, 2013; Waller, 2013]. Not only does it collect data, but it also analyses and illustrates the data with the help of R [R Project]. It has also out of the box support for Kieker.

MooBench accounts for Java Just in Time Compilation (JIT) by repeating the experiment very often until no more optimizations are deployed. It divides the recorded data in two halves and rates only the second one, as the first one has heavier fluctuations than the second one. Waller and Hasselbring [2013] recommend having $2,000,000$ iterations of the experiment. In some cases JIT may even recognize a method as dead code and eliminate it completely. MooBench also accounts for these cases by calculating dummy values. It also mitigates the performance fluctuations of garbage collections by doing a warm up phase in which memory is being reserved for the experiment.

The performance and overhead evaluation will be conducted with the help of this library, as it fits most.

5.2 Conduction

The experiment is segmented into four parts: Overhead Analysis, Turnaround Analysis, Scalability Analysis and Reliability Analysis. Each with its own experiment procedure and results. Overhead Analysis answers **RQ-1**, Scalability Analysis **RQ-2** and Reliability Analysis **RQ-3**.

5.2.1 Overhead Analysis

Figure 5.2 shows an extended version of Figure 2.3. It shows the sequence of a manual instrumentation of the monitored method which is used in this experiment. It is annotated similarly like in the paper by Waller and Hasselbring [2013]. The sequence diagram is extended by the turnaround time which will be important in the next section.

Note that T_2 is executed exactly once for every monitored method. At this point, the *Probe Controller* caches whether the signature of the monitored method is to be monitored or not. This is done to avoid executing this expensive query multiple times. As the experiment only contains one class and one method, T_2 will not be covered in I.

- M is the actual method time.
- $C_1 + C_2 = C$ is the time spent collecting monitoring data.
- W is the time spent writing the recorded data.
- I is the instrumentation time (does not include T_2).
- $T_{MI_1} + T_{MI_2} = T_{MI}$ is the turnaround time of the manual instrumentation approach.
- T_{BCI} is the turnaround time of the BCI approach.

To measure the actual method time M, the experiment is also conducted without any monitoring at all. C is measured by conducting the experiment with a no-operation record writer and subtracting M. W can be measured by conducting the experiment with an active record writer and subtracting $C + M$. I is the time it takes to determine whether to collect monitoring data or not. It can be measured by conducting a BCI experiment and subtracting $C + M + W$ (see Figure 5.3).

Figure 5.3 depicts the process of the BCI of the monitored method. In contrast to the manual instrumentation process it does not contain I. Thus I is the actual overhead which is to be assessed. The performance of the monitoring writer can heavily fluctuate due to hard drive slowdowns etc. Also both approaches contain W and it does not influence the result in any way, so it will be neglected in the analysis part.

The experiment is conducted with the following settings in MooBench:

▷ Sleep time of 5
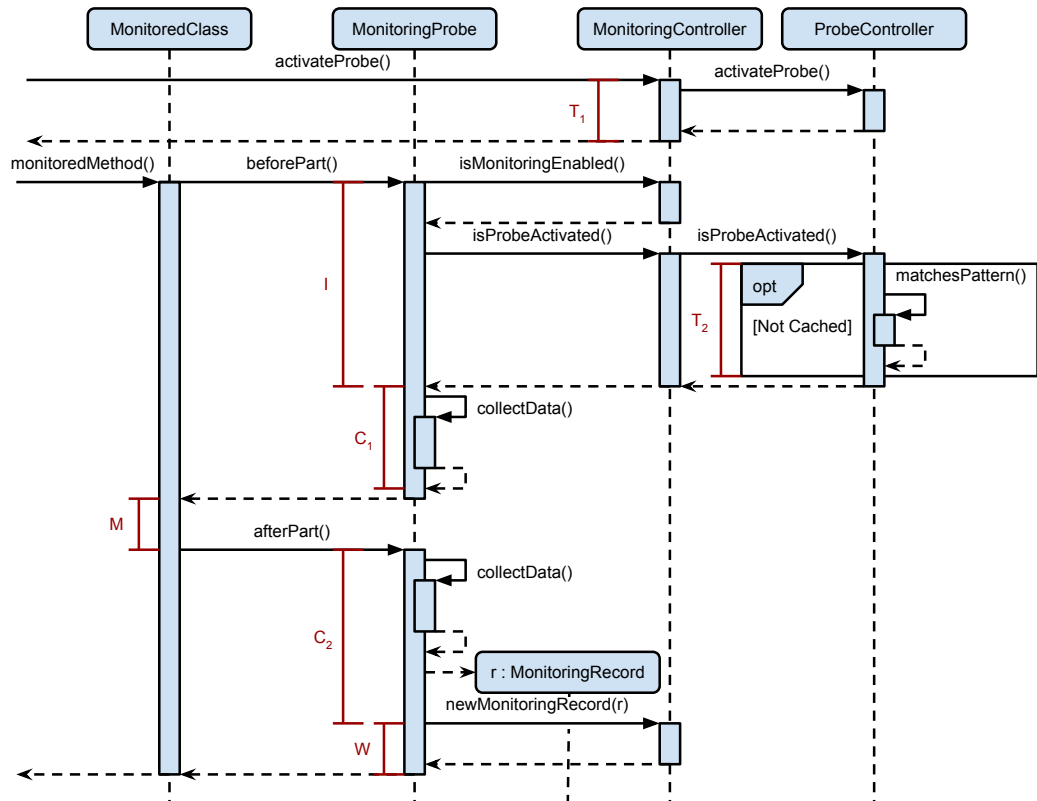
5. Experimental Evaluation

Figure 5.2. The process of manual instrumentation for evaluation [Waller and Hasselbring, 2013]

▷ Single loop

▷ Single thread

▷ Recursion depth of 10

▷ 2 000 000 method calls

▷ Having a method time of 0 ms

▷ No quickstart

▷ Java HotSpot Server VM

▷ Maximum of 4 GB allocated RAM

The experiment is conducted on the following machines:

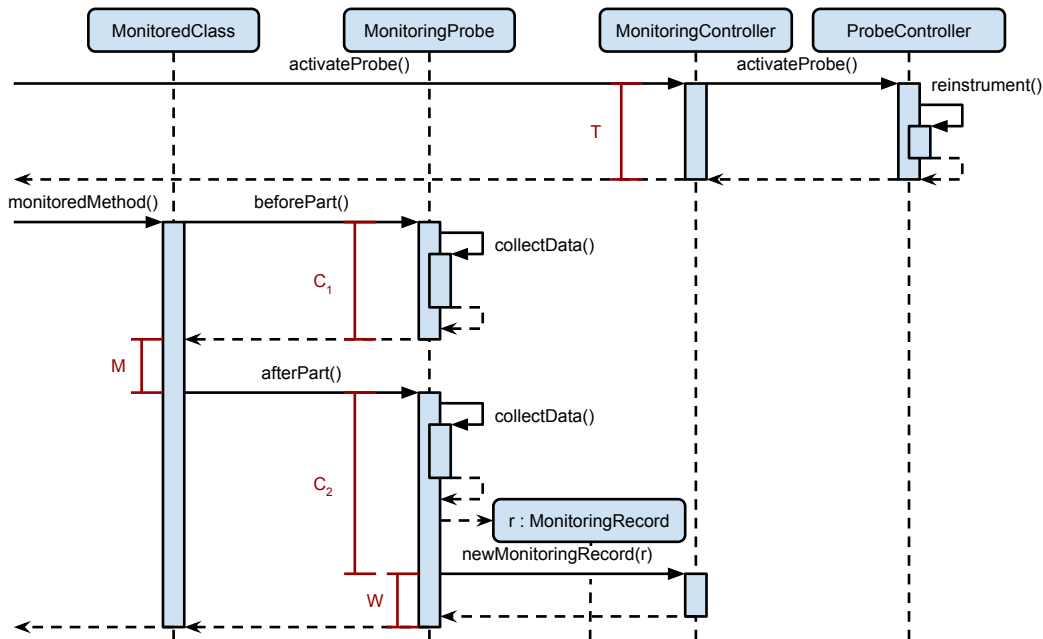**Figure 5.3.** The process of BCI for evaluation

Emulab Machine	Personal Machine	University Machine
Ubuntu 12.04 LTS	Windows 7	Gentoo
OpenJRE/Oracle JRE 1.7	Oracle JRE 1.7	OpenJRE 1.7

Figure 5.4 shows the result of the experiment. The left side shows the the BCI approach, the right side shows the manual instrumentation approach.

The BCI approach shows a slight increase in execution time as soon as the method is instrumented. When activating the monitoring record writer the execution time increases hugely and also fluctuates hugely as expected. In comparison the manual instrumentation results however show a massive increase in execution when the method is instrumented. It seems very strange that even deactivating the Monitoring completely won't reduce the execution time significantly. Thus the experiment has been repeated with the Oracle JRE. However the bizarre result remains (Figure 5.5).

Only after deactivating JIT the results started to seem plausible (Figure 5.6). The experiment was then repeated with two other machines, a Windows 7 machine and a Linux Gentoo machine, which both gave more plausible results (Figure 5.7). Unable to find a valid explanation, the weird result is discarded.

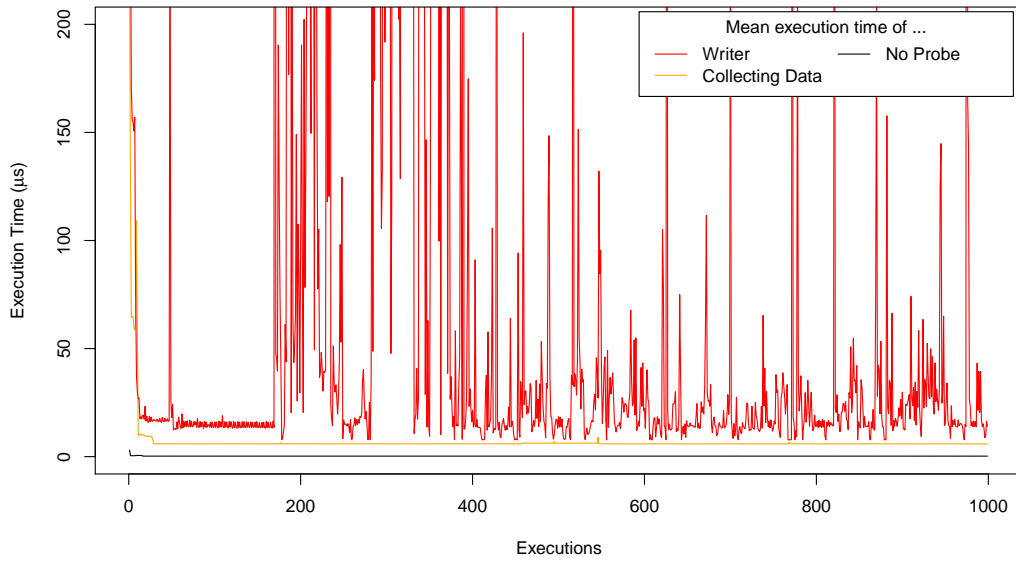Using the data of the experiment (ignoring the emulab experiment) the values determined are shown in Figure 5.8
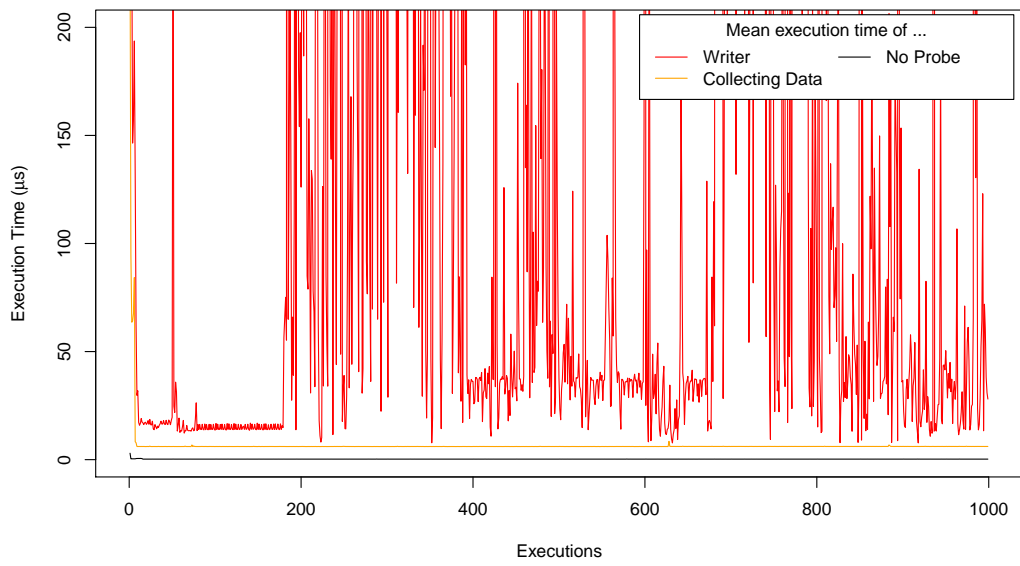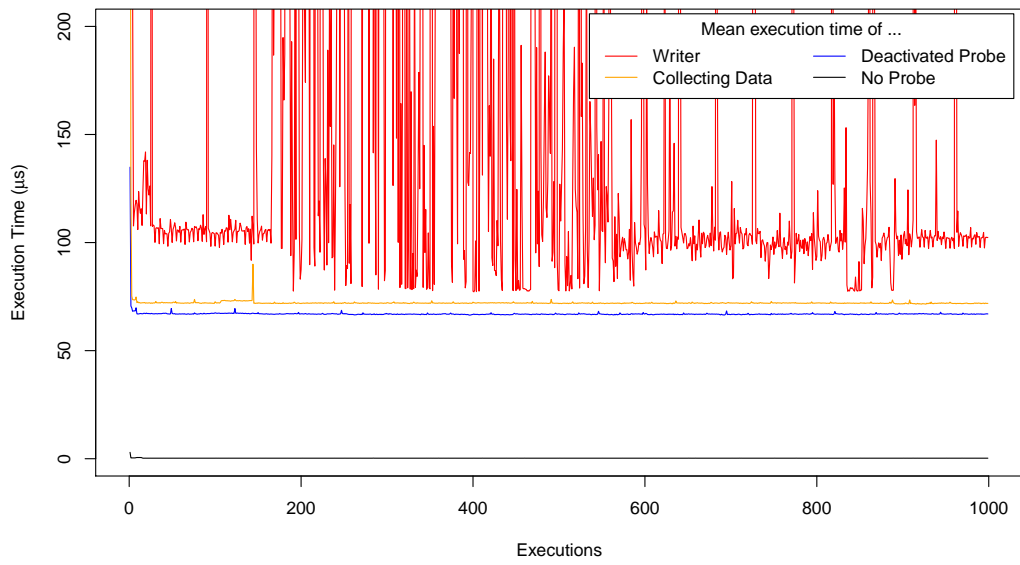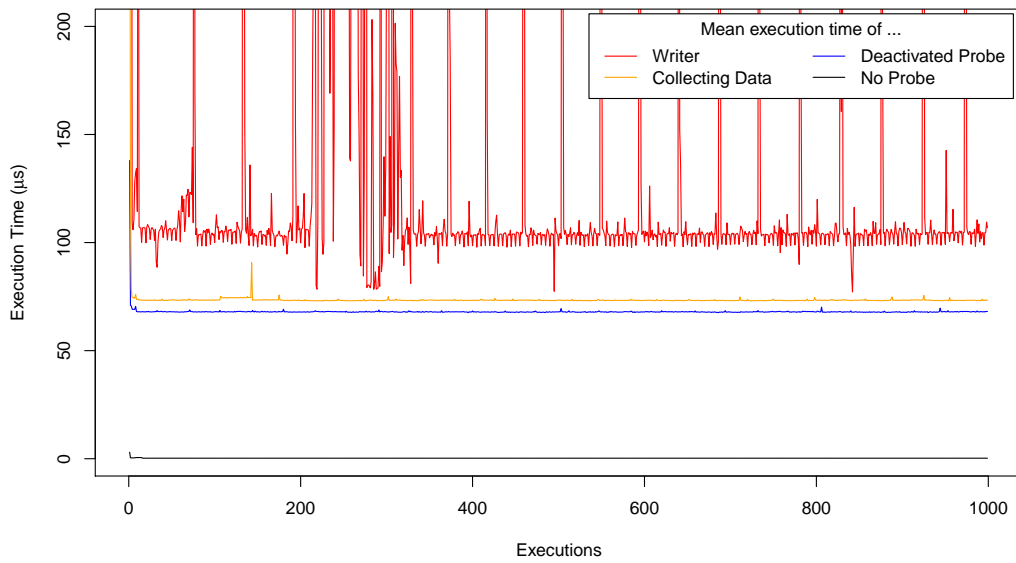
(a) BCI results

(b) Manual instrumentation results

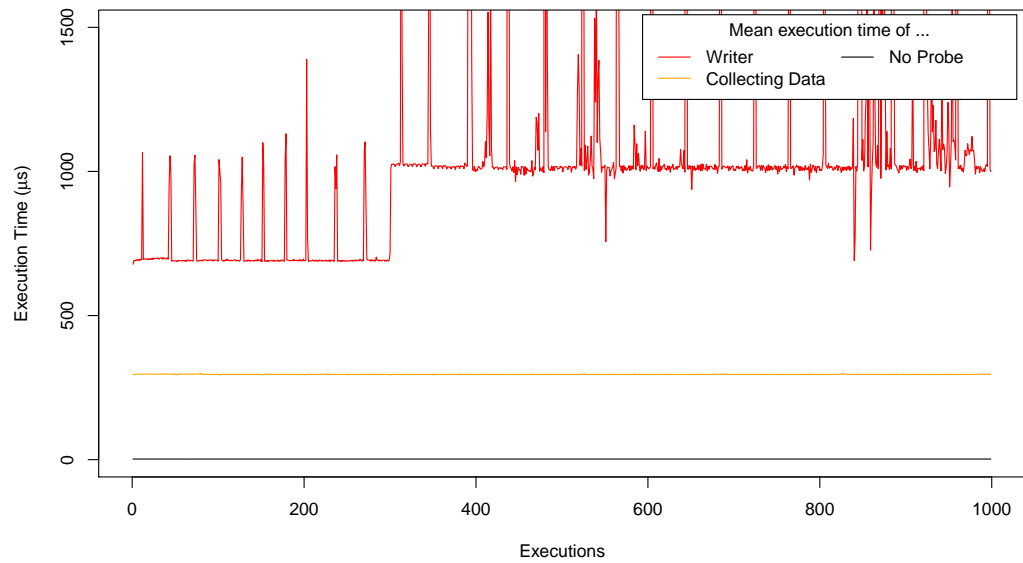**Figure 5.4.** Overhead experiment results on Ubuntu 12.04 LTS with OpenJRE 1.7

(a) BCI results

(b) Manual instrumentation results

Figure 5.5. Manual instrumentation result on Ubuntu 12.04 LTS with OracleJRE 1.7

(a) BCI results

(b) Manual instrumentation results

Figure 5.6. Overhead experiment results on Ubuntu 12.04 LTS with OpenJRE 1.7 without JIT
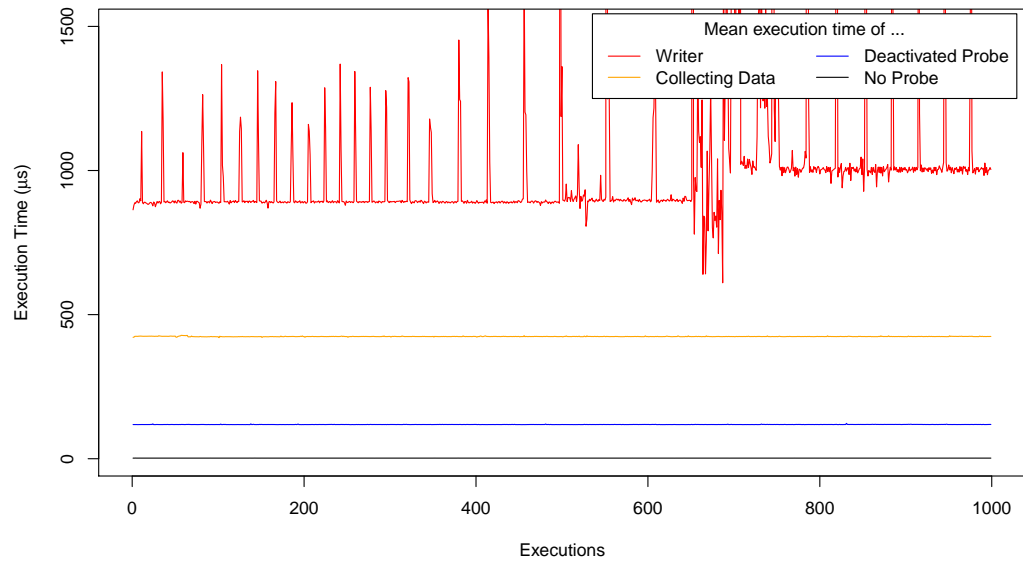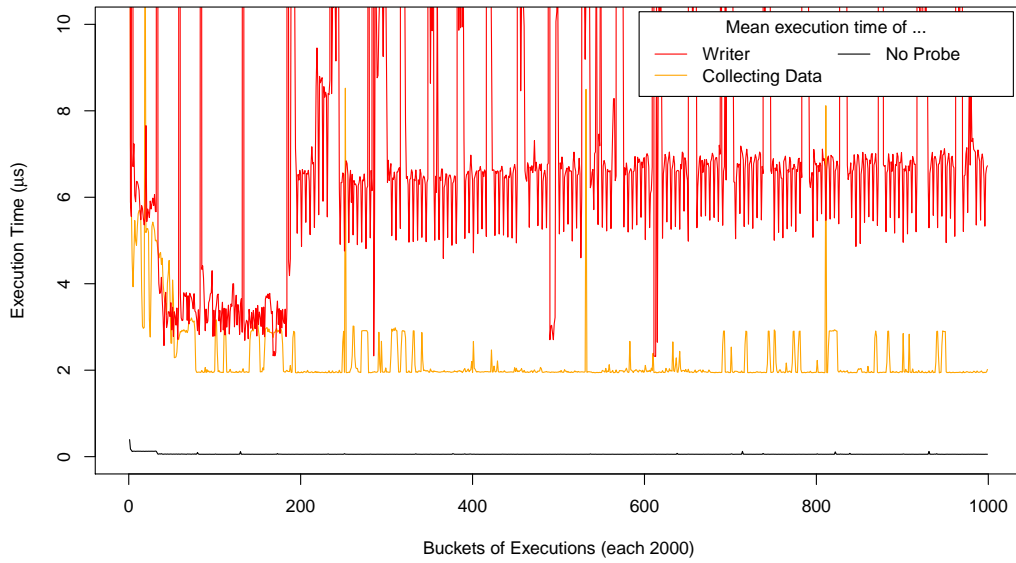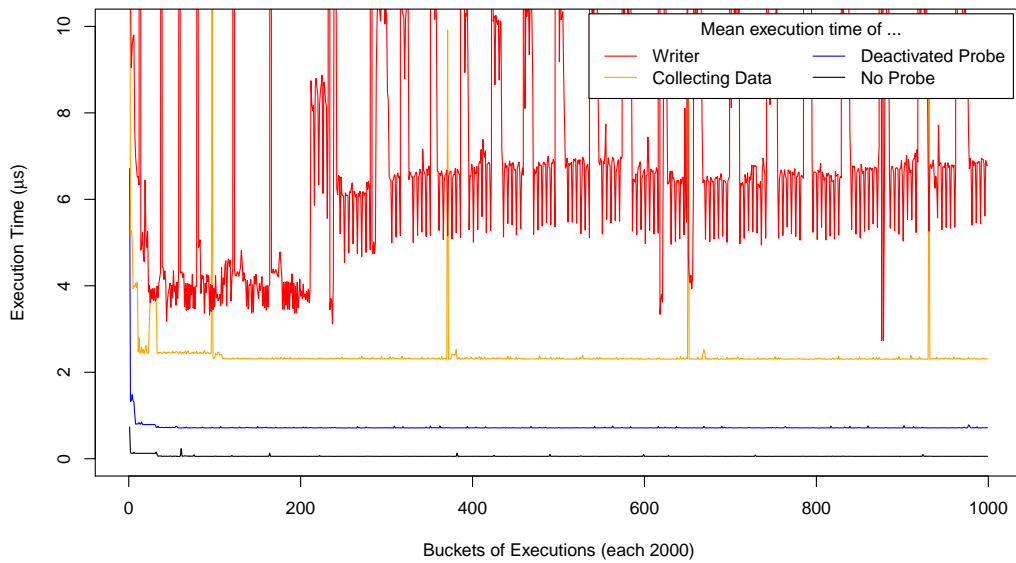
(a) BCI results

(b) Manual instrumentation results. Similarities can be observed with [Ehlers et al., 2011]

Figure 5.7. Overhead experiment results on Windows 7 with Oracle JRE 1.7

	$M(\mu s)$	$I + M(\mu s)$	$C + M$	$I + C + M(\mu s)$
Mean	0.0561	0.7160	2.0718	2.3389
Confidence Interval (CI) 95%	0.0005	0.0006	0.0353	0.0382
Median	0.0000	0.7890	1.9740	2.3680
Max	131.8410	71.0530	13086.6300	13955.8340
Min	0.0000	0.3940	1.5780	1.9730
Standard Deviation	0.2871	249.53	18.0090	19.4660
Sample Size	1,000,000	1,000,000	1,000,000	1,000,000

Figure 5.8. Overhead time experiment results

To perform the overhead analysis two hypothesizes are established [Field and Hole, 2014]. We compare the overhead caused by $C + M$ and $I + C + M$ which are the results of BCI and manual instrumentation respectively.

H_0 There is no significant change in the overhead
H_A There is a significant reduction in overhead

Using the values from Figure 5.8 the t-test can be conducted:

$n_1 = n_2 = 1,000,000$
$\bar{y}_1 = 2.0718$
$\bar{y}_2 = 2.3389$
$s_1 = 18.0090$
$s_2 = 19.4660$
$H_0 = \mu_1 - \mu_2 = 0$
$H_A = \mu_1 - \mu_2 < 0$
$\alpha = 0.05$

The resulting t value is -10.0721. The rejecting range is $]-\infty, -1.6449]$. t does lie within this range. It is shown that indeed a significant improvement can be made out, thus eliminating H_0 and proving H_A.

5.2.2 Turnaround Analysis

When using the BCI approach there are two cases: (1) If a class has not been loaded yet and (2) if a class has already been loaded by the JVM. The sequence of events that take place are illustrated in Figure 5.9. In the case that a class has not been loaded yet, the JVM first loads it and then checks if it needs to be instrumented. Then it is instrumented and reloaded again. The other case does not have this step and thus fares a little bit better in terms of turnaround time. Also class loading takes place by reading the class definition from the hard drive which, as already stated in Section 5.2.1, being neglected in this evaluation. Thus only the second case is considered.
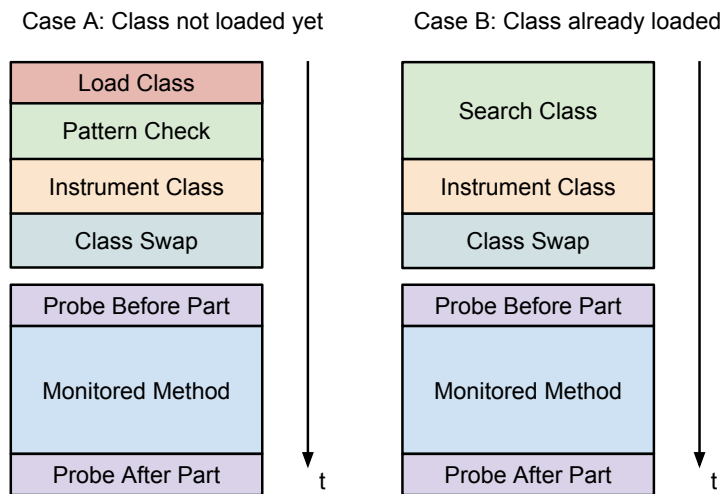
Figure 5.9. The two cases when instrumenting

The experiment is conducted on a set of dummy classes. By the help of a macro, 10,000 dummy uninstrumented and preinstrumented classes are generated, each with a single method. A sample dummy file is shown in Listing 5.1. This is necessary to bypass the cache of the *Probe Controller*. The experiment is conducted to measure the time T. The procedure in the case of the BCI approach is to instrument one class and afterwards execute the instrumented method exactly one time. In the case of the manually instrumentation approach the procedure is to activate the probe of one class and also execute the instrumented method exactly once. The procedure is depicted in Figure 5.2 and Figure 5.2 from Section 5.2.1.

The results of this experiment are shown in Figure 5.10.

Using the data of the experiment the values determined are shown in Figure 5.11

	$T_{BCI}(\mu s)$	$T_{MI}(\mu s)$
Mean	38984.74	129.47
CI 95%	646.28	21.87
Median	37846.74	111.31
Max	190712.76	5510.88
Min	36867.00	62.36
Standard Deviation	7373.26	249.53
Sample Size	500	500

Figure 5.11. Turnaround time experiment results
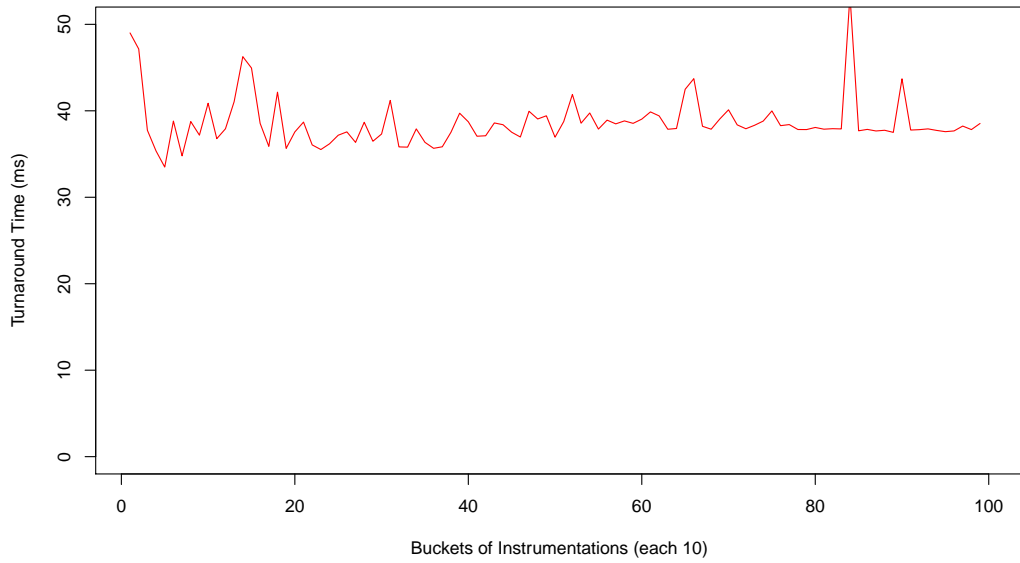
(a) BCI results
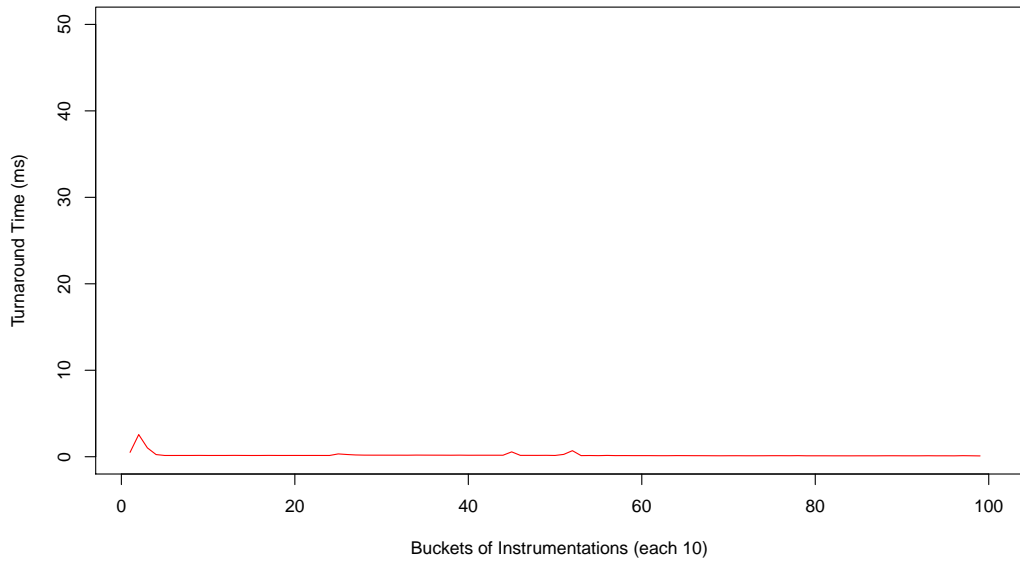
(b) Manual instrumentation results

Figure 5.10. Overhead experiment results on Windows 7 with Oracle JRE 1.7

```
 4
 5   public final class MonitoredClassBCITurnaround00005 implements MonitoredClass {
 6
 7       final ThreadMXBean threadMXBean = ManagementFactory.getThreadMXBean();
 8
 9       public final long monitoredMethod(final long methodTime, final int recDepth) {
10           if (recDepth > 1) {
11               return this.monitoredMethod(methodTime, recDepth - 1);
12           } else {
13               final long exitTime = System.nanoTime() + methodTime;
14               long currentTime;
15               do {
16                   currentTime = System.nanoTime();
17               } while (currentTime < exitTime);
18               return currentTime;
19           }
20       }
21   }
```

Listing 5.1. Monitored dummy class uninstrumented
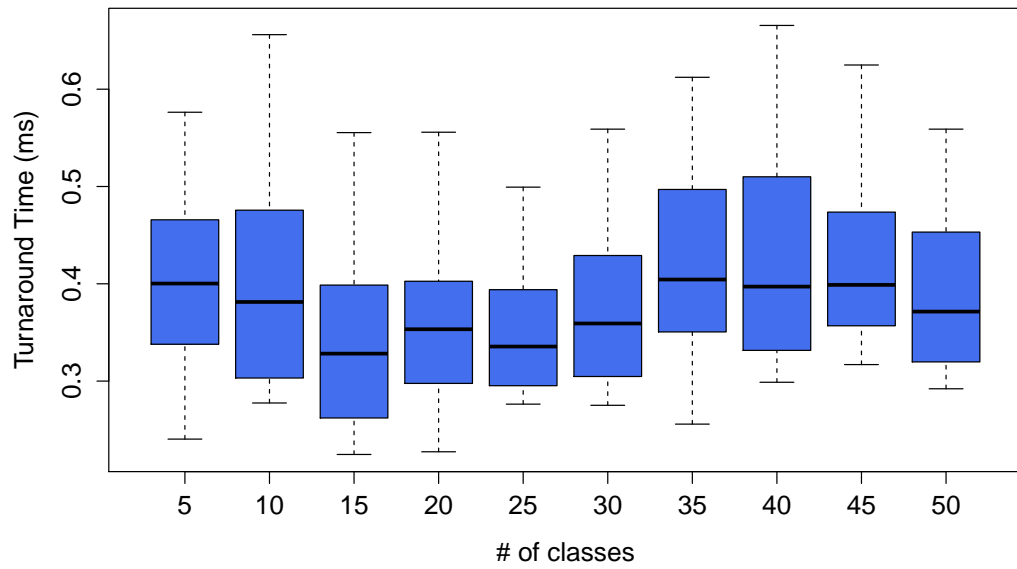
5.2.3 Scalability Analysis

After the turnaround analysis it is apparent that the turnaround time is tremendous in comparison to the manual instrumentation. So an analysis is conducted to see how far the BCI approach scales. The previous experiment is modified to instrument a certain amount of dummy classes simultaneously. Afterwards the dummy method of every instrumented dummy class is executed exactly once. On the manual instrumentation side, the probe is activated for the same amount of preinstrumented dummy classes. Afterwards, like with BCI, every dummy method is executed only once to induce the caching. This experiment is first conducted with 5 classes. Then the experiment is conducted a total of ten times, each with 5 more classes. Figure 5.12 show the results.

In case of the manual instrumentation the turnaround time is, as expected, almost constant no matter the amount of classes. The BCI approach shows a static linear increase in turnaround time. However this experiment is conducted with the best case scenario for the BCI approach. There are three variables that affect the turnaround time during BCI heavily: (1) the amount of loaded classes in the JVM, (2) the amount of methods in the instrumented classes and (3) the amount of instrumentation patterns in the pattern list. Because when starting the instrumentation, each loaded class is matched with each pattern in the pattern list. If a class matches, every method of it will be matched against the instrumentation pattern. This experiment was conducted with exactly one pattern in the pattern list by the use of wildcards. The classes were preloaded before starting the experiment. Also every instrumented dummy class had only a single method.
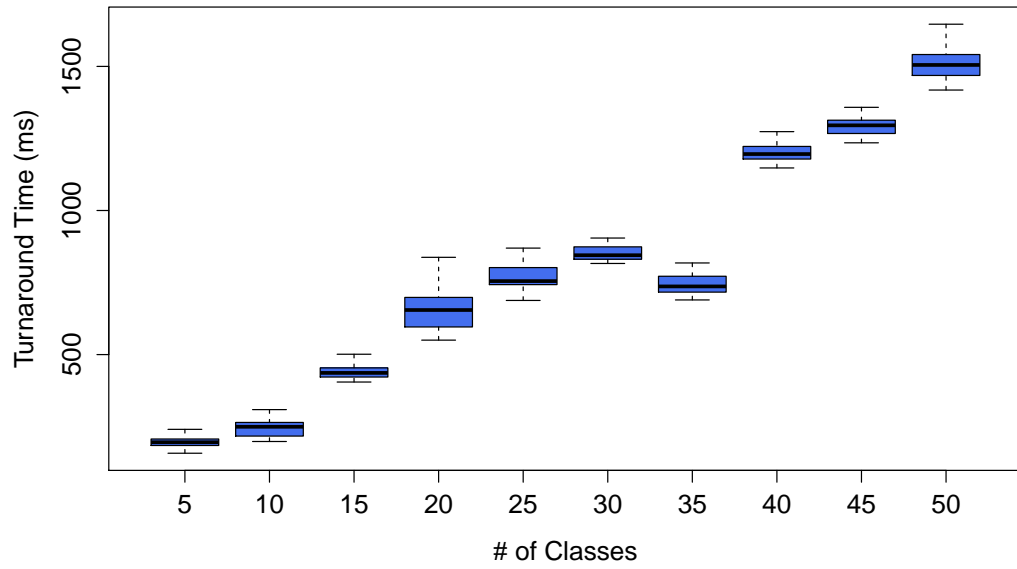
(a) BCI results

(b) Manual instrumentation results

Figure 5.12. Scalability experiment results on Windows 7 with Oracle JRE 1.7

5.2.4 Reliability Analysis

The reliability analysis will show how the turnaround time affects an application. For this purpose, a simple HTTP server is created. All the HTTP server does, is sending a reply with the content `This is the response`, when getting a GET request. Upon getting a POST request the server instruments the method responsible for replying. Listing 5.2 shows a simplified code snippet of it.

For workload Apache JMeter is used [Apache Software Foundation]. It constantly sends a flood of requests to the HTTP server to simulate users accessing a website. The load drive system is configured to send a POST request once every second. The POST request makes the server activate instrumentation. This is done to simulate a webserver being dynamically instrumented while under heavy work load.

JMeter is first calibrated on a no instrumentation server implementation. It is calibrated to have as many simultaneous threads until the server is just about to start missing requests. Figure 5.13 shows an experiment with no lost transactions. The determined value is 200 threads. These 200 threads are sequentially started throughout the first 30 seconds of the experiment. The experiment runs then for another 180 seconds. JMeter checks every request to be replied with `This is the response`. If a wrong reply arrives it will be count as a lost transaction. Figure 5.14 show results with lost transactions.

Now, several metrics will be defined for this experiment.

- L_{MI_i} amount of lost transactions for the manual instrumentation at experiment i.
- L_{BCI_i} amount of lost transactions for the BCI at experiment i.
- S_{MI_i} amount of samples for the manual instrumentation at experiment i.
- S_{BCI_i} amount of samples for the BCI at experiment i.
- $\sum_{i=1}^{10} S_{MI_i} = S_{MI}$ sum of all the samples of all experiments for the manual instrumentation.
- $\sum_{i=1}^{10} S_{BCI_i} = S_{BCI}$ sum of all the samples of all experiments for the BCI.
- $S_{MI} \dfrac{L_{MI_i}}{S_{MI_i}} = \widehat{L}_{MI_i}$ mean amount of lost transactions independent from amount of samples for the manual instrumentation at experiment i.
- $S_{BCI} \dfrac{L_{BCI_i}}{S_{BCI_i}} = \widehat{L}_{BCI_i}$ mean amount of lost transactions independent from amount of samples for the BCI at experiment i.
- $\sum_{i=1}^{10} \widehat{L}_{MI_i} = \widehat{L}_{MI}$ number of mean lost transactions for the manual instrumentation.
- $\sum_{i=1}^{10} \widehat{L}_{BCI_i} = \widehat{L}_{BCI}$ number of mean lost transactions for the BCI.

The experiment is repeated on each approach 10 times and afterwards compared against each other. As the amount of samples varies from experiment to experiment, the amount of lost transactions are divided by the amount of samples, to get a value independent from the number of samples. After all experiments have been conducted a mean value of all

samples is calculated which, afterwards, is multiplied with the amount of lost transactions. Figure 5.16 summarizes the result of the experiments. Figure 5.15 gives an illustration.

```java
18  public class HandlerSimple implements HttpHandler {
19      public static final IMonitoringController CTRLINST = MonitoringController
20              .getInstance();
21      public static boolean instrumented = false;
22      private static final String CLAZZ = "test.HandlerMI";
23      public static final String SIGNATURE = "public void " + CLAZZ
24              + ".handle(com.sun.net.httpserver.HttpExchange)";
25
26      public static final Runnable instrument = new Runnable() {
27          @Override
28          public void run() {
29              CTRLINST.activateProbe(SIGNATURE);
30          }
31      };
32      public static final Runnable uninstrument = new Runnable() {
33          @Override
34          public void run() {
35              CTRLINST.deactivateProbe(SIGNATURE);
36          }
37      };
38
39      public void handle(final HttpExchange t) throws IOException {
40          final String response = "This is the response";
41          t.sendResponseHeaders(200, response.length());
42          OutputStream os = t.getResponseBody();
43          os.write(response.getBytes());
44          os.close();
45
46          if (t.getRequestMethod().equals("POST")) {
47              if (instrumented)
48                  new Thread(uninstrument).start();
49              else
50                  new Thread(instrument).start();
51              instrumented = !instrumented;
52          }
53      }
54  }
```

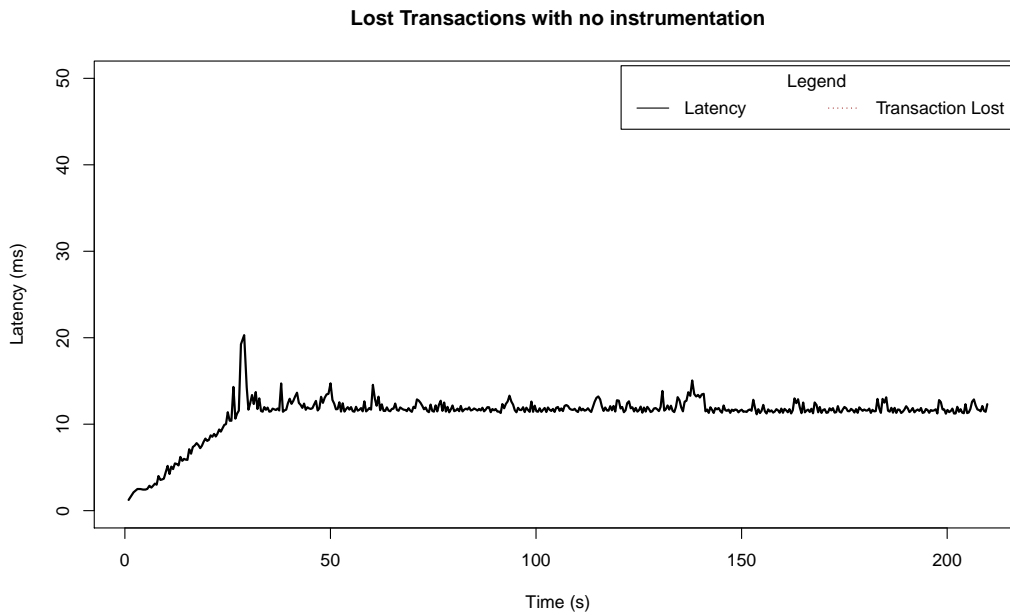**Listing 5.2.** Simple HTTP server handler.

Figure 5.13. Progression of latency over the course of the experiment with no instrumentation. No lost transactions were registered. Conducted on Windows 7 with Oracle JRE 1.7

	$\widehat{L}_{MI}$	$\widehat{L}_{BCI}$	Lat_{MI}	Lat_{BCI}
Mean	24.817	50.743	11.436	15.980
CI 95%	3.282	2.474	0.002	0.006
Standard Deviation	26.385	28.444	6.549	12.746
Sample Size	10	10	26,105,373	15,655,600

Figure 5.16. Lost transactions and latency experiment results
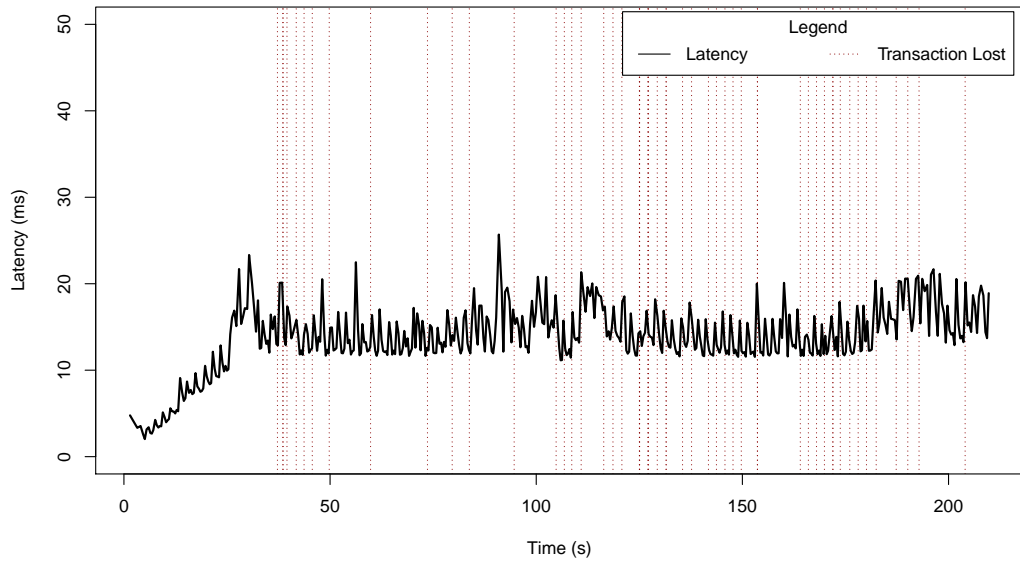
(a) BCI results
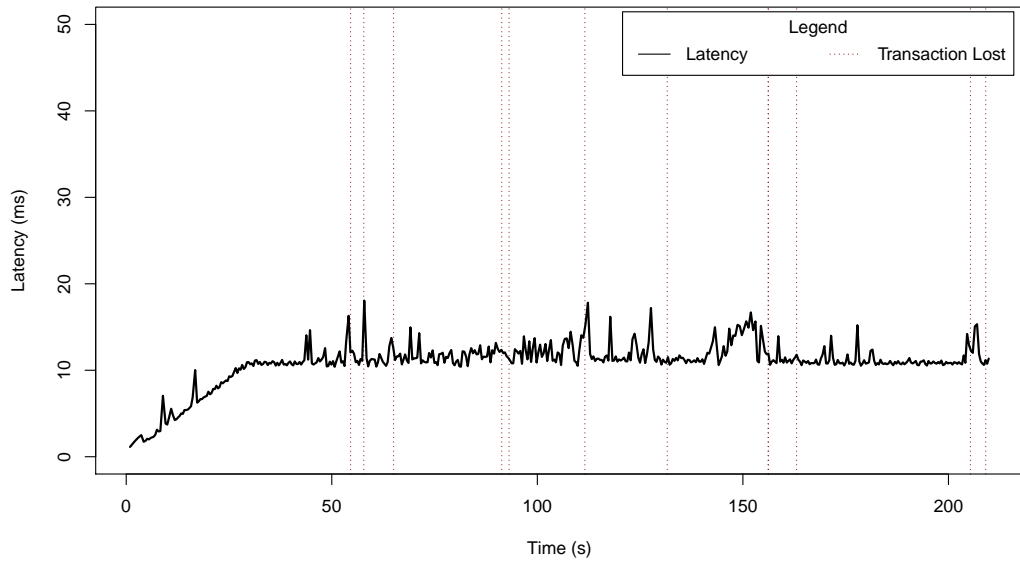
(b) Manual instrumentation results

Figure 5.14. Result of one reliability experiment on Windows 7 with Oracle JRE 1.7
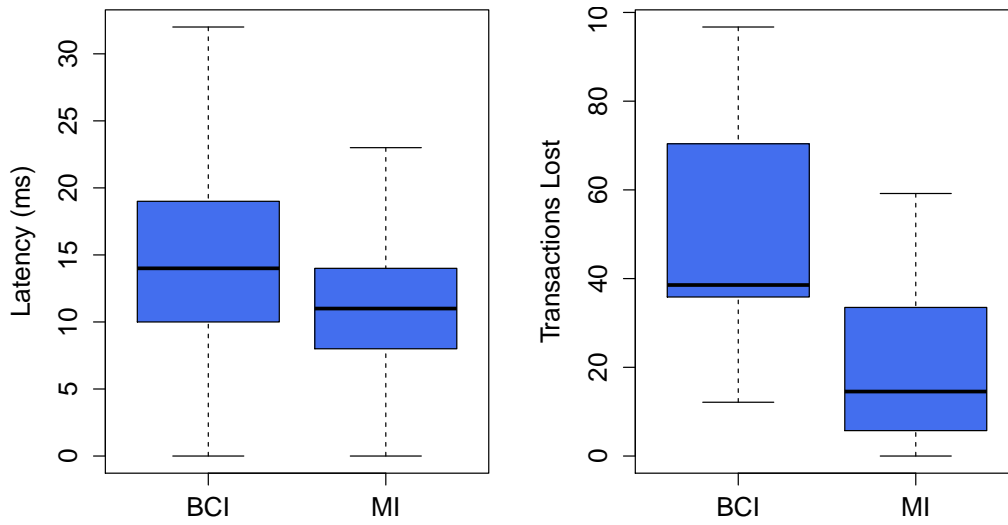
Figure 5.15. Amount of lost transactions and latency in comparison with both methods.

To perform the reliability analysis, again, two hypothesizes are established.

H_0 There is no significant change in lost transactions
H_A There is a significant increase in lost transactions

Using the values from Figure 5.16 the t-test can be conducted:

$n_1 = n_2 = 10$
$\bar{y}_1 = 24.817$
$\bar{y}_2 = 50.743$
$s_1 = 26.385$
$s_2 = 28.444$
$H_0 = \mu_1 - \mu_2 = 0$
$H_A = \mu_1 - \mu_2 < 0$
$\alpha = 0.05$

The resulting t value is -2.1132. The rejecting range is $]-\infty, -1.7341]$. t does lie within this range and thus H_0 is rejected. So there is a significant change in lost transactions.

Conclusion

This chapter is to put a lid on this thesis. Section 6.1 summarizes the result of the evaluation. Section 6.2 discusses the conduction of this thesis. Section 6.3 will give a showcase of future work which can be done.

6.1 Summary

Kieker has been successfully integrated into AIM. The experimental evaluation shows a reduction in overhead when using BCI. The turnaround time is increased heavily as a side effect though. The turnaround time scales linearly given an efficient enough implementation of the *Kieker Instrumentor* component. Improvements on the performance of this crucial component are given in Section 6.3. The reliability analysis showed that the dynamic BCI also has a negative impact on the amount of lost transactions, in contrast to manual instrumentation. The impact is borderline significant, though. All in all the approach has its advantages as well as its drawbacks.

6.2 Discussion

This thesis showed the feasibility of dynamic bytecode instrumentation using Kieker. Initially, this thesis consisted of three parts: (1) a survey of existing approaches for bytecode modification during execution time (2), the development of an alternative dynamic instrumentation approach using the approach which came out of the survey (3) a quantitative comparison between the approach using bytecode modification and Kieker's existing one. The goal of the survey of runtime bytecode manipulation techniques and technologies was to identify various technologies which provide the capability of modifying Java classes during runtime. Furthermore, the technologies should have been evaluated in terms of functionality and feasibility to determine the right choice. However this part was neglected after a properly working framework has been found [Wert and Heger, 2014]. So this thesis consists now only of the two latter parts.

Dynamic BCI gives great capabilities to Java applications, however it also has its drawbacks. A freshly instrumented class is in interpreter mode. It takes a few thousand executions, until JIT does heavy optimization on it. So in most cases the reduction of the overhead does not pay for itself. Rather its a better idea to use compile-time instrumentation

with AOP or manual instrumentation as much as possible. Dynamic BCI, however, comes in handy when instrumenting methods not specified by AOP. A hybrid approach would be most suitable for Kieker.

6.3 Future Work

This thesis showed a proof of concept to the usage of BCI in accordance with Kieker. However this concept is not a full-fledged all-around solution. It has its deficiencies which can be addressed in future work. One important point is to reduce the overhead during instrumentation pattern matching. A optimal solution would be to use the patterns in a hierarchical way. Top-level packages would be scanned first. This would improve the turnaround time of the BCI a lot and would help scalability as well.

Also the current class loader implementation does not support loading class files from within jars. An improvement could be done by using an established solution for class loading like [Kamranzafar].

Another point to improve the performance is to directly instrument classes upon loading. The current solution reads the class file from the file system, then loads the class into the JVM. After this the check is done whether to instrument the class or not and is only afterwards instrumented and reloaded yet again into the JVM. An improvement would be to make class instrumentation happen upon reading the class file, before defining the class within the JVM.

Also it would dramatically increase turnaround time to cache all classes loaded by the JVM. Newly loaded classes are registered by the *Kieker Class Loader* which can then be added to the cache and held in suitable data structure. This would eliminate the expensive fetch to get all loaded classes from the Java Instrumentation API.

Furthermore a hybrid approach can be developed for Kieker. As of now Kieker's adaptive monitoring API was replaced by the dynamic BCI support. The hybrid approach would decouple adaptive monitoring and dynamic BCI. This would allow for quick disabling and enabling of monitoring probes without the heavy turnaround of the BCI approach. And still it would allow to dynamically instrument methods, which were not preinstrumented during compile-time.

In regard to research, this thesis has done no experimental evaluation using macro benchmarking. Every experiment is a micro benchmark. Thus evaluating the proposed approach on a bigger piece of software would give more insight on the overhead of more usual case scenarios.

Glossary

AIM **Adaptable Instrumentation and Monitoring.** A Java monitoring framework which has the capabilities to dynamically instrument applications during runtime by the means of BCI. Developed by researchers of the Karlsruhe Institute of Technology and released as open source software licensed under the Apache License, Version 2.0. iii, 1–3, 7–9, 11, 13, 14, 16, 17, 19, 43

AOP **Aspect Oriented Programming.** A technique to modularize a piece of software by other means than by classes and components. Behavior which is common to a set of certain classes is centralized in aspects. Usually used in monitoring, profiling and logging frameworks.. 11, 14, 44, 45

BCI **Bytecode Instrumentation.** Describes the technique to modify the bytecode of classes of an application either before or during its runtime. Commonly used in AOP, especially to inject Monitoring Probes.. iii, 2, 7, 8, 11, 14, 17, 23, 25, 27–37, 40, 43–45

CI **Confidence Interval.** A percentage value expressing the amount of samples which are within the given interval. E.g., 100 numbers have a mean value of 50. In this example having a 95% confidence interval of 10 will tell that 95 number lie between 40 and 60.. 32, 33, 39

GQM **Goal Question Metric.** An approach to define metrics by establishing research questions which are answered with the help of metrics. Used to avoid gathering data for unnecessary metrics. [Ludewig and Lichter, 2007]. 23, 24

JIT **Just in Time Compilation.** A technique Java employs to increase the performance of frequently used components by compiling them during runtime into native bytecode. Usually during runtime more optimization can be deployed than during compilation time so it is very efficient. However it can falsify experiments if not taken in consideration carefully.. 24, 27, 30, 43

Bibliography

[Asp]   AspectJ language extension. URL `http://www.eclipse.org/aspectj/`.

[asm]   ASM - Website. URL `http://asm.ow2.org/`.

[jav]   Javassist - Website. URL `http://www.csg.ci.i.u-tokyo.ac.jp/~chiba/javassist/`.

[ser]   SERP - Website. URL `http://serp.sourceforge.net/`.

[jav 2006]   The -javaagent: Option, 2006.   URL `http://javahowto.blogspot.de/2006/07/javaagent-option.html`.

[Andreß 2003]   H.-J. Andreß. T-Test für Mittelwertunterschiede zwischen zwei unabhängigen Stichproben, 2003. URL `http://eswf.uni-koeln.de/lehre/stathome/statcalc/v2401.htm`.

[Apache Software Foundation]   Apache Software Foundation. Apache JMeter. URL `http://jmeter.apache.org/`.

[Dmitriev 2003]   M. Dmitriev. Design of jfluid: A profiling technology and tool based on dynamic bytecode instrumentation. Technical report, Mountain View, CA, USA, 2003.

[Dufour et al. 2004]   B. Dufour, C. Goard, L. Hendren, O. De Moor, G. Sittampalam, and C. Verbrugge. Measuring the dynamic behaviour of aspectj programs. In *ACM SIGPLAN Notices*, volume 39, pages 150–169. ACM, 2004.

[Ehlers and Hasselbring 2011]   J. Ehlers and W. Hasselbring. A self-adaptive monitoring framework for component-based software systems. In I. Crnkovic, V. Gruhn, and M. Book, editors, *5th European Conference on Software Architecture (ECSA '11)*, volume 6903 of *Lecture Notes in Computer Science*, pages 278–286. Springer-Verlag, 2011. URL `http://oceanrep.geomar.de/14429/`.

[Ehlers et al. 2011]   J. Ehlers, A. van Hoorn, J. Waller, and W. Hasselbring. Self-adaptive software system monitoring for performance anomaly localization. In *Proceedings of the 8th ACM International Conference on Autonomic Computing*, ICAC '11, pages 197–200, New York, NY, USA, 2011. ACM.

[Field and Hole 2014]   A. Field and G. Hole. *How to Design and Report Experiments*. SAGE Publications, Limited, 2014.

[Heger 2012]   C. Heger. *Automatische Problemdiagnose in Performance-Unit-Tests*. February, 2012.

Bibliography

[Heiss 2005] J. J. Heiss. Better profiling through code hotswapping: A conversation with jfluid project lead, misha dmitriev, 2005. URL http://www.oracle.com/technetwork/articles/java/dmitriev-qa-138654.html.

[Hilsdale and Hugunin 2004] E. Hilsdale and J. Hugunin. Advice weaving in aspectj. In *Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 26–35. ACM, 2004.

[Iskhodzhanov et al. 2013] T. Iskhodzhanov, R. Kleckner, and E. Stepanov. Combining compile-time and run-time instrumentation for testing tools. *Programmnye produkty i sistemy*, 3:224–231, 2013. URL http://swsys.ru/index.php?page=article&id=3593&lang=en.

[Kamranzafar] Kamranzafar. URL https://github.com/kamranzafar/JCL.

[Kieker Project 2014] Kieker Project. *Kieker 1.9 User Guide*. Software Engineering Group, Kiel University, Kiel, Germany, Oct. 2014. URL http://kieker-monitoring.net/documentation/.

[Lindholm and Yellin 2013] T. Lindholm and F. Yellin. *Java SE 7 Virtual Machine Specification*. Boston, MA, USA, 2013. URL http://docs.oracle.com/javase/specs/jvms/se7/html/index.html.

[Ludewig and Lichter 2007] J. Ludewig and H. Lichter. *Software Engineering - Grundlagen, Menschen, Prozesse, Techniken*. dpunkt.verlag, 2007.

[Oracle a] Oracle. *Java Instrumentation API*, a. URL http://docs.oracle.com/javase/6/docs/technotes/guides/instrumentation/index.html.

[Oracle b] Oracle. *Java SE 1.5.0 API Specification*, b. URL docs.oracle.com/javase/1.5.0/docs/api/java/lang/instrument/Instrumentation.html.

[Oracle 2011] Oracle. Java Servlet Technology. http://www.oracle.com/technetwork/java/index-jsp-135475.html, 2011.

[R Project] R Project. *R*. Free Software Foundation. URL http://www.r-project.org/.

[Rohr et al. 2008] M. Rohr, A. van Hoorn, J. Matevska, N. Sommer, L. Stoever, S. Giesecke, and W. Hasselbring. Kieker: Continuous monitoring and on demand visualization of Java software behavior. In *Proceedings of the IASTED International Conference on Software Engineering 2008 (SE'08)*, pages 80–85. ACTA Press, Feb. 2008.

[Sedlacek 2008] J. Sedlacek. Profiling with visualvm, 2008. URL https://blogs.oracle.com/nbprofiler/entry/profiling_with_visualvm_part_1.

[SpringSource 2011] SpringSource. Spring. http://www.springsource.org/, 2011.

[The Apache Foundation 2011] The Apache Foundation. Apache CXF. http://cxf.apache.org/, 2011.

[van Hoorn et al. 2009]   A. van Hoorn, M. Rohr, W. Hasselbring, J. Waller, J. Ehlers, S. Frey, and D. Kieselhorst. Continuous monitoring of software services: Design and application of the Kieker framework. Technical Report TR-0921, Department of Computer Science, Kiel University, Germany, Nov. 2009.

[van Hoorn et al. 2012]   A. van Hoorn, J. Waller, and W. Hasselbring. Kieker: A framework for application performance monitoring and dynamic software analysis. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering (ICPE 2012)*, pages 247–248. ACM, Apr. 2012.

[Waller 2013]   J. Waller. Benchmarking the performance of application monitoring systems. Forschungsbericht, November 2013. URL http://eprints.uni-kiel.de/22475/.

[Waller and Hasselbring 2012]   J. Waller and W. Hasselbring. A comparison of the influence of different multi-core processors on the runtime overhead for application-level monitoring. In V. Pankratius and M. Philippsen, editors, *Multicore Software Engineering, Performance, and Tools*, Lecture Notes in Computer Science, pages 42–53. Springer, Juni 2012.

[Waller and Hasselbring 2013]   J. Waller and W. Hasselbring. A benchmark engineering methodology to measure the overhead of application-level monitoring. In *Proceedings of the Symposium on Software Performance: Joint Kieker/Palladio Days 2013*, pages 59–68. CEUR Workshop Proceedings, November 2013.

[Wert 2012]   A. Wert. *Uncovering performance antipatterns by systematic experiments*. PhD thesis, Master's thesis, Karlsruhe Institute of Technology, 2012.

[Wert and Heger 2014]   A. Wert and C. Heger. Adaptable instrumentation and monitoring, 2014. URL http://sopeco.github.io/AIM/.

Decleration

I declare that this thesis is the solely effort of the author. I did not use any other sources and references than the listed ones. I have marked all contained direct or indirect statements from other sources as such. Neither this work nor significant parts of it were part of another review process. I did not publish this work partially or completely yet. The electronic copy is consistent with all submitted copies.

_____

place, date, signature