UNIVERSITY OF TARTU
Institute of Computer Science
Software Engineering Curriculum

Ostap Maliuvanchuk

Performance optimization of a Java instrumentation agent for calling context encoding

Master's Thesis (30 ECTS)

Supervisor(s): Vesal Vojdani
Nikita Salnikov-Tarnovski

Tartu 2016

Performance optimization of a Java instrumentation agent for calling context encoding

Abstract:

The idea behind calling context encoding algorithms is to efficiently build a call graph of an application in order to be able to give developers a call stack trace of any event at any point of the program execution. Having the information that calling context provides enables developers to better interpret results of monitoring and profiling tools. In this paper, we discuss in greater detail the benefits of calling context encoding and the problems with current algorithms that are trying to construct calling context. We take an algorithm implemented as Java instrumentation agent - Lucce - and explain its promising possibilities, benefits over other similar algorithms, as well as its main performance problem. This thesis contributes to this field firstly by presenting an analysis of different methods of performance optimization and their applications to a Java agent, and secondly by applying these methods to the performance optimization of the Lucce algorithm and its Java implementation.

Keywords:
Performance optimization, calling context encoding**,** Java agent, profiling**,** instrumentation
CERCS: P170 Computer science, numerical analysis, systems, control

Funktsioonikutsete ajalugu kodeeriva Java agendi jõudluse tõstmine

Lühikokkuvõte:

Funktsioonikutsete ajalugu, mida kasutajale trükitakse pinujäljena, on suureks abiks programmis toimuva vea täpse asukoha leidmiseks lähtekoodis. Sügavamate probleemide puhul on vaja programmi täitmist pikemalt jälgida ja oluliste sündmuste toimumisel nende funktsioonikutsete ajalugu salvestada. Kuna terve ajalugu on väga pikk, siis on mõistlik seda kodeerida. Antud magistritöös uuritakse ühte konkreetset kodeerimise algoritmi Lucce. Välja on toodud selle eeliseid teiste algoritmidega võrreldes ning on näidatud probleeme jõudlusega. Eesmärgiks on selle algoritmi jõudlust tõsta ja selle näite varal tutvustada üldisi Java agentidega seotud jõudluse tõstmise võtteid.

Võtmesõnad:
Jõudluse optimeerimine, kutsekonteksti kodeering, Java agent, profileerimine, instrumenteerimine
CERCS: P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine

Table of Contents

1 Introduction 4

2 Related work 6

2.1 Efficient path profiling 6

2.2 PCCE 8

2.3 DeltaPath 10

3 Background 12

3.1 Algorithm 12

3.2 Instrumentation 14

4 Validation methodology 16

4.1 Testing 16

4.2 Benchmarking 16

4.2.1 Black box benchmarking 16

4.2.2 Micro benchmarking 18

4.3 Continuous integration 19

5 Preliminary benchmarking 21

6 Java performance tuning 23

6.1 Finding a bottleneck 23

6.2 Performance optimization techniques 25

7 Instrumentation improvements 31

7.1 Instrumentation with ASM 31

7.2 Attempted optimizations 36

8 Results 50

9 Conclusions 52

10 References 54

11 Appendix 56

11.1 Bash script to run DaCapo benchmarks 56

11.2 Glossary 57

11.3 License 58

1 Introduction

Java is a language with automatic garbage collection, which makes it easier for regular developers to work with objects and memory. Nevertheless, it does not fully prevent memory leaks, other nested bugs or human mistakes. If developers receive an error during the program execution, they can easily get a call stack trace that shows how, when and where the error occurred. However, for most complicated cases just knowing place of the error in the code is not enough as the error might have nothing to do with the actual problem. Therefore, we need a tool that would allow us to see the call stack trace at any time or point during the program execution. This would allow us to have a deeper understanding of the program flow and help us to find the existing hidden problems described above. The implementation, in the simplest form, would require storing the full stack alongside each event. However, in terms of memory taken to store all the stacks, this is not an acceptable solution. Therefore, ideally, it would be good to store just one number with every call, from which it would be possible to recover the full call stack trace.

There are several algorithms that attempting to deal with this problem by building a calling context graph. One of them is "Lazy and Unintrusive Calling Context Encoding" or Lucce [1]. It is an algorithm that builds a calling context graph during the program execution in such a way, that at any time you can trace back the event or method that occurred. It has many benefits over other similar algorithms and has demonstrated good performance on small projects. However, the current version is unfortunately not suitable for production applications [1]. On a big project the overhead it produces is too high, with the result that the execution time of the instrumented program increases dramatically. Compared to other algorithms, Lucce has a lot of potential since it works in runtime, does not need to analyze code beforehand and can show the exact stack trace of an occurred event. Decreasing the overhead related to the execution of Lucce alongside with an application would make it possible to use the algorithm as a tool in production environments for many IT projects that use Java. This would be helpful for many developers in both computer software and academic industries in their day-to-day work and will lead to faster development process and lower debugging time. For example, it can be used as a memory leak detection tool.

The author of Lucce, Salnikov-Tarnovski, have tried the most obvious optimizations such as caching values and offloading work into parallel threads. Nevertheless, there is still a room for improvement [3]. Thus, this thesis will analyze existing algorithms, compare them with Lucce and investigate the problem with current implementation in order to find

out how we can improve the algorithms and decrease the overhead. After finding a method to optimize it, the solution will be implemented. To validate the results we will use the benchmarks suits on a new version and compare them to the results of the original version of the algorithm.

Performance optimization of Java agent has many stages that has to be done in order to see any kind of the improvement. On an example of Lucce we gathered and discussed practices, methodologies and techniques that help to save time during the optimization process and ensure the efficiency and correctness of the changes being made. The process of optimization used in this theses is applicable to any regular a Java agent as well as to performance optimization in general.

The structure of the thesis is as follows. Chapter 2 explains the state of the art algorithms that were trying to solve the same problems of path or calling context encoding. The benchmarks, as well as advantages and disadvantages of every algorithm, are discussed as well. In chapter 3, the workings of the Lucce algorithm are explained in detail, along with the implementation of instrumentation in Java. Chapter 4 presents the validation methodology and the preparation done in order to make the optimization process easier. In Chapter 5, the preliminary benchmarks techniques and their results are described. This is followed by the actual optimization techniques and methods used to make the algorithm faster in Chapters 6 and 7. In Chapter 8 we validate our achievements by doing benchmarks of the original and improved algorithm implementations on different versions of Java. Finally, in the Conclusion we summarized received results and discussed the further work.

2 Related work

The following section presents a review of the algorithms that are based on the similar main idea or principle as Lucce. That is to mark the calling context with an integer number and to have a possibility to decode this number back to a stack trace. The algorithms are discussed in the chronological order of the time they were published. In this way, it is possible to see the evolution and development of the algorithms. The review and description of the algorithms is based on the criteria listed below:

- Idea and possibilities of an algorithm
- Implementation
- Execution overhead
- Benchmarks
- Limitations

2.1 Efficient path profiling

The idea of representing a path as a single number with the possibility to decode it back was originally introduced by Ball and Larus [2]. They created an algorithm which is usually referred to as BL encoding, based on the first letter of authors' names. The algorithm is an alternative to the previously used methods of block and edge encoding. Edge encoding is based on finding the most frequent paths using a heuristic approach, thus it cannot actually identify all the most frequent paths. Although the algorithm was not perfect, such inaccuracies were omitted in favor of the low overhead of the algorithm. Ball and Larus decided to improve the accuracy of the algorithm while still keeping the overhead relatively small. The BL algorithm provides users with a way to get results that are more accurate and it still works with a comparable overhead. For example, on SPEC95 benchmarks the BL overhead is 31%, while the edge profiling is 16% [2]. There are a few possible usages of the BL algorithm, for example, as a tool for optimization and performance improvement or as a test coverage tool that needs path profiling to discover the program's test coverage. The algorithms works with intra-procedural calls and identifies paths from the START to END node. Nodes in between do not count.

The BL algorithms performs the following functions:

- For every edge the algorithm assigns a value, so that the sum of them in any path from START to END is different. This sum also should be minimal, in order that the final encoding contains a number from 0 to *n*-1, where *n* is the number of paths. The algorithm for creating and assigning the value is displayed in Listing 1, and the result graph can be seen in Figure 1. As you can see every possible path from A to B has a different sum of edge values.

```
foreach vertex v in reverse topological order {
        if v is a leaf vertex {
                NumPaths(v) = 1;
        } else {
                NumPaths(v) = 0;
                for each edge e = v->w {
                        Val(e) = NumPaths(v);
                        NumPaths(v) = NumPaths(v) + NumPaths(w);
                }
        }
}
```

Listing 1. Algorithm for assigning values to edges [2].

- The algorithm finds the minimal cost set of chord edges, with respect to edge weighting
- Then the algorithms places the appropriate instrumentation on chords
- At the end the algorithm has to map the paths to a correspondent integer representation, in order to be able to decode the number back to the path

The algorithm was implemented as a proof of concept in a tool called PP using C++ and EEL library to handle the instrumentation. Nowadays, it is widely used by many tools to provide better debugging experience. The benchmarks were run on the SPEC95 [4] test suit and show that the BL algorithm had an average overhead of 30.9%, compared to the edge profiling algorithm which on the same machine had an average overhead of 16%. These are good results and proved that path profiling can be used instead of edge profiling, since it is more accurate and gives longer paths.

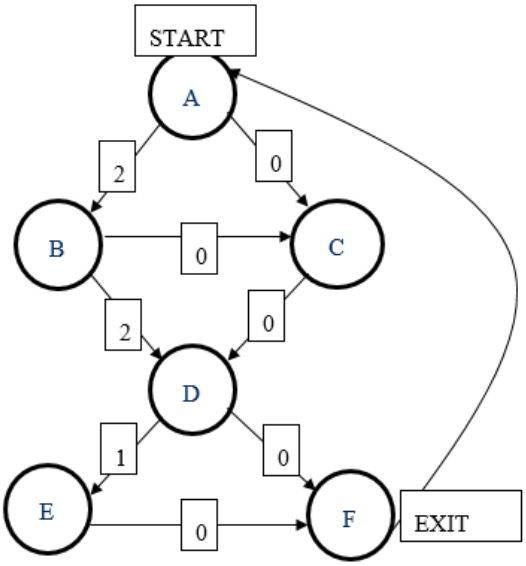

Figure 1. Result of assigning values to edges

Nevertheless, the algorithm has several limitations. The algorithm works just for intra-procedural acyclic paths which means that back edges have to be removed. All the encoding is done for paths from the START to END node, and intermediate paths are not encoded. Despite these limitations, the BL algorithm continues to be widely used in many profiling tools.

2.2 PCCE

As described above, one of the problems with the BL encoding is that it worked just for intra-procedural paths. Sumner et al., based on path encoding, created new algorithm that works not only with intra-procedural calls. He introduced a Precise Calling Context Encoding or PCCE algorithm [5], which forms the basis for Lucce algorithms. The main goal of the algorithm is to uniquely identify every calling context from the start of execution by integer number, so that it can be easily decoded back to the stack trace, as the usual back tracing stack of calls is too expensive. Sumner et al. have discovered that the context encoding scheme has to satisfy slightly different criteria than the BL algorithm: ‘context encoding has a different criterion, that is, all unique paths leading from the root to a specific node have unique encodings, because we only need to distinguish the different contexts with respect to that node’ [5]. This observation allowed their algorithm to handle recursive methods and function pointers.

The algorithm has two stages:

1) Annotation – for every node, the algorithm computes a total number of calling contexts.
2) Instrumentation – in runtime the algorithm calculates an identifier for each calling context for a specific node, see Listing 2.

```
for n ∈ N {
        s ← (n has a dummy edge in E) ? 1 : 0
}
for each edge e = <p, n, ℓ> in E {
        if e is not a back edge {
                insert id = id + s before ℓ
                insert id = id − s after ℓ
                s ← s+numCC(p)
        } else {
                insert push(<id, ℓ>) before ℓ
                insert id = 0 before ℓ
                insert id = pop().first after ℓ
        }
}
```

Listing 2. PCCE encoding algorithm

Where N is a set of nodes(functions), $< n, m, \ell >$ is a triple that represents an edge in which ℓ represents a call site where n calls m and $n, m \in N$ represent a caller and callee, respectively m.

The algorithm was implemented in OCalm using CIL as an instrumentation library. Since CIL only supports C language, thus it can instrument C programs. Although the algorithm works on the source code level, rather than on the compile level, it still shows impressive results. Sumner et al. claim that their algorithm adds an overhead of on average between 2% and 4% [5], which is really good result for calling context encoding. There is no memory overhead since the algorithm does not use any runtime data structure.

Nevertheless, this algorithm also has several limitations. It uses the analysis of a source code in advance, which makes it not applicable to some applications. This is especially the case with languages such as Java, where it would not work correctly. Even though Java compiles to a byte code, it still interprets the command using Java Virtual Machine, thus it has a lot of dynamic class loading and virtual functions that are major obstacles for PCCE.

2.3 DeltaPath

The DeltaPath algorithm [6] was developed to address the problem of calling context encoding for Object oriented languages with dynamic class loading and polyphormism. The main idea of Zeng et al. algorithm is to ensure invariant that for any given node, its encoding space is divided into disjoint sub-ranges, with each sub-range encoding calling contexts along one incoming edge of the node [6]. Zeng et al. proposed a new encoding scheme that has the possibility to recalculate the addition value when a new incoming call comes into the function.

The algorithm works as follows. Let us say that we have the situation presented in Figure 2. Where *p* is a virtual function call that can dispatch into multiple methods $n_1 \ldots n_k$. At the beginning, each node has a CAV – candidate additional value associated with it that equals 0.

If new incoming edge comes in, three following steps are happening:

1) Calculate *a* as $max(CAV[n_1] \ \ldots \ CAV[n_k])$, where *a* is the additional value
2) Update candidate values to $CAV[n_1] \ldots CAV[n_p] \ = \ ICC[p] \ + \ a$
3) After that the additional value for the last edge is calculated => $ICC[n] \ = \ CAV[n]$

The implementation of the algorithms is divided into two parts. The first is a static analysis, which builds a call graph based on Java byte code. It is implemented with the help of WALA [7] to generate the graph. The runtime part is implemented in Java and Javaassist [8]. It hooks onto every method discovered with the static analysis and instruments it.

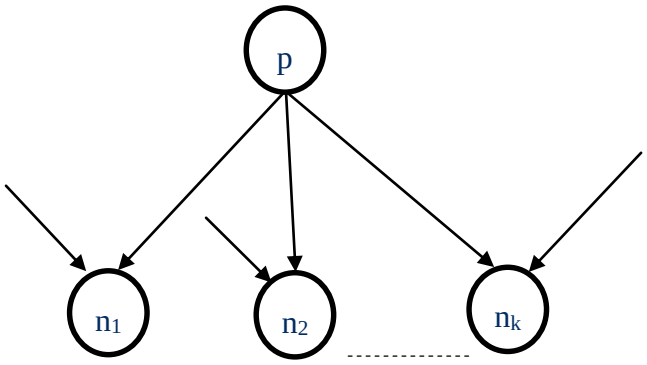

Figure 2. Situation where p is a virtual function that can be dispatched to any of n_i

SPECjvm2008 benchmark suits was used to evaluate the algorithm. The authors compared DeltaPath with the PCC (Probalistic Calling Context) algorithm, and their results have shown that DeltaPath is on average 32.51% slower [6]. This is a good trade-off as the algorithm provides reliable and precise decoding capabilities. Nevertheless, the algorithm still needs to do static analysis, which adds some startup time and increases the complexity of tools built with this algorithm.

Figure 3 contains a table with a summary of all the algorithms described above. As you can see, some of them does not work with JVM and all of them requires static code analysis in order to work. Next chapter describes the Lucce algorithms that addresses these problems.

Algorithm	**Requires static analysis**	**Precise**	**Works with JVM**	**Overhead**
PCCE	Yes	Yes	No	2-4%
DeltaPath	Yes	Yes	Yes	32%
BL	Yes	No	No	30.9%

Figure 3. Results overview from algorithms described above

3 Background

The Lucce algorithm consists of essentially two main pars. The first part is the actual implementation of Lucce algorithm. The second important part is the instrumentation of the third party code. These two parts are connected via the Java instrumentation mechanism, which we describe in more detail later.

3.1 Algorithm

Lucce is an algorithm designed specifically for Java to deal with dynamic class loading. The main idea of the algorithm is to build the calling context graph in runtime, while also storing some information as local variables in class methods. The detailed workings of the algorithm is described in the tech report on Lucce [1]. We are going to use the same notation to briefly explain the most important parts of the algorithm that are necessary for understanding the following chapters.

A call graph is a pair $\langle N, E \rangle$, where N is a set of nodes and E is a set of edges. Each node represents a class method. Each edge represents a method call and consists of three values $\langle p, n, l \rangle$, where $p, n \in N$ are caller and callee, respectively, l represents a call site of a method call from p to n. Every call to a new method adds a new node. Every new call from a different call site adds a new edge. On any of these updates the algorithm has to calculate next three values:

1. $numCC$ – is a value calculated for every node and depends on incoming edges and caller nodes.
2. Additional value or av – is a value calculated for each edge and depends on numCC values of neighbor edges, meaning edges that go to the same node.
3. $callId$ – is the encoded number, having which allows the algorithm to decode it and get the whole stack trace. It is a sum of all the additional values from the root node to the currently processed node.

Below is a more formal definition of the same values:

$$numCC(n) = \begin{cases} 1, & if\ n\ is\ a\ root\ node\ of\ GC \\ \sum_{e \in n^-} numCC(e^{\Delta}), & otherwise \end{cases}$$

where n^- is a set of incoming edges of node n, e^Δ is a caller of edge e. Here, the numCC values are dynamically updated as new call edges are visited during the program execution. Using these values the additional values can be computes as follows:

$$av(e_i) = \sum_{j<i} numCC(e_j^\Delta)$$

having that all the node incoming edges are ordered or $n^- = \{e_1, e_2, \dots, e_{|n-|}\}$. We can then define the encoding of the call path as follows:

$$callId(n, e{:}\, t) = \begin{cases} 0, if\ n\ is\ a\ root\ of\ GC \\ av(e) + callId(e^\Delta, t) \end{cases}$$

where $e{:}\, t$ denotes a path to node n, e is an edge that leeds to n and is preceded by path t.

context:	***callId***:
AB	0
ABC	0
ABCD	0
ABCDB	1
ABCDBE	1

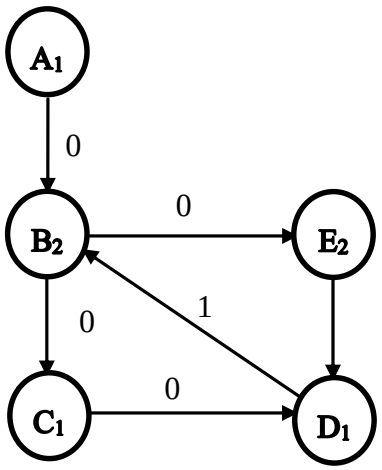

Figure 4. Example of call graph encoding

Figure 4 depicts the context encoding results and a constructed call graph. Every node and edge is annotated with *numCC* and *additional value*, respectively. *CallId* is essentially the sum of the additional values of the edges that build the path from the root to the required node. Having a current call graph and $callId$ allows the algorithm to decode it back to a path by doing a reverse version of encoding scheme.

Compare to other algorithms described before all the calculations and graph updates are done dynamically during program execution, which makes the algorithm easy to use.

Unfortunately, the original implementation produces the overhead, which is too big to use Lucce in production environments (see Figure 5)

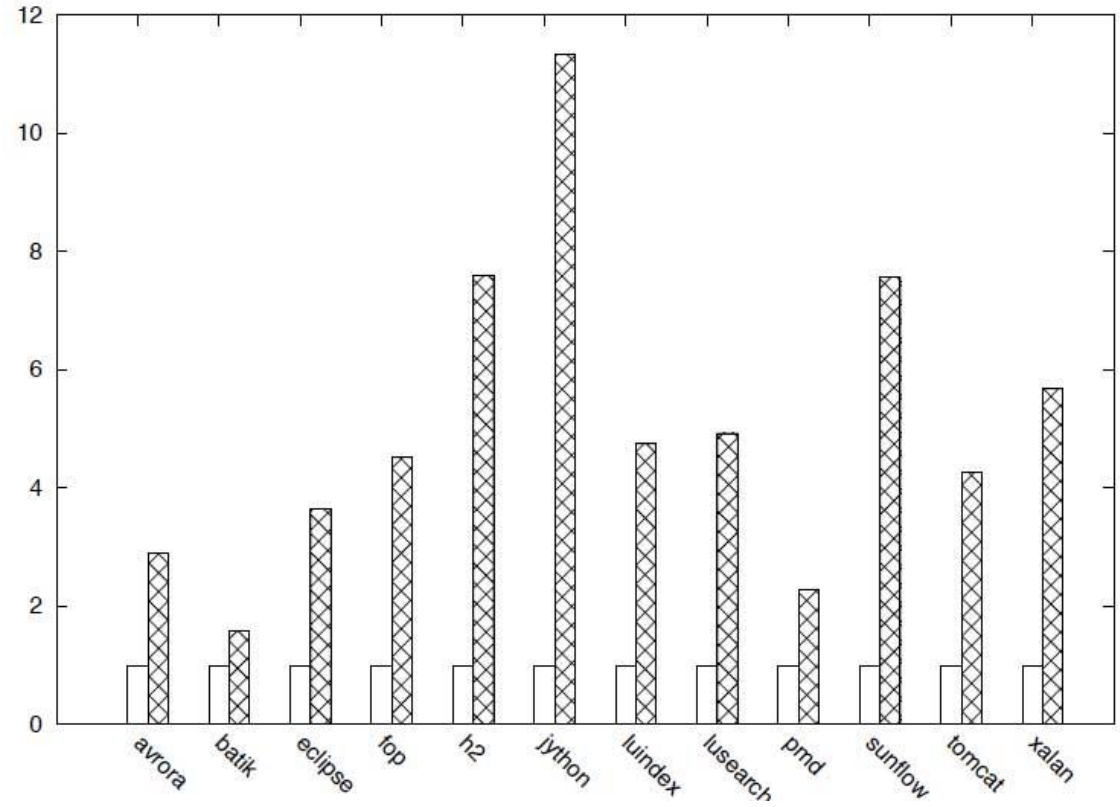

Figure 5. Benchmark result of Lucce conducted in [1] using DaCapo benchmark suit [9].

3.2 Instrumentation

Another important part is the instrumentation of the third party code. This is done with a help of Java Instrumentation Mechanism[1]. The idea behind it is that before each class is loaded by JVM, we can inject additional hooks to the code in order to execute our methods. Those changes cannot modify the application state and behavior, but they give us the necessary information to build the calling context graph.

Below is a list of hooks and variables that the algorithm injects into the third party code with the original agent [1]:

[1] https://docs.oracle.com/javase/8/docs/api/java/lang/instrument/package-summary.html [Accessed 15 May 2016]

- Every method now has two additional local integer variables: *callId* and *graphVersion*
- At the very beginning of each method *entered* hook is added
- Before any method call *calling* hook is added to notify transition to another method
- At the end of each method *returned* hook is added

```
private Order createOrder(User user, Order order) {
//int callId;
// int graphVersion;
//[callId, graphVersion] = Lucce.entered(methodId);
//Lucce.calling(methodId, someCallSite);
order.setUser(user);
//Lucce.calling(methodId, someCallSite + 1);
order = repo.save(order);
//Lucce.returned(methodId)
 return order;
}
```

Listing 4. Example of a method after the instrumentation where commented lines represent injected byte code instructions

Listing 4 shows the code of Java combined with pseudo code and the changes that are being made to every method. These three hooks allows the algorithm to track all the required method calls. *Calling* and *entered are designed to* create a bridge between method call site and an actual execution. The reason for this is that it is hard to know beforehand what method is going to be executed in runtime as Java has polymorphism, inheritance and supports virtual methods.

An analysis of this part of the algorithm gives us some ideas on how it might be improved and whether it is possible to do the instrumentation in a more efficient way.

4 Validation methodology

In order to be able to optimize the algorithm and validate the result we had to do two things. Firstly, we had to make sure that every change that we make to the code does not break the functionality and the algorithms still works as it is supposed to. To ensure this, a set of unit tests were introduced. Secondly, we needed to measure the overhead and the algorithm execution time. This is also important for the validation of the optimization. This chapter outlines the three measures we have taken to insure the correctness of the algorithms and validation of the performance optimizations.

4.1 Testing

The initial optimization is planned to be performed on the algorithms itself and not on other parts, such as instrumentation. It is sufficient to test the algorithm using the "Unit test" technique [10], which is widely used among software engineers. This method involves testing different parts of the system as units. A unit can be a separate method or class. Running the unit tests usually does not take much time, usually less than 1 minute.

Since Java does not support unit tests out of the box, we have used popular libraries called TestNG[2] and HamcRest[3], which allowed us to write structured unit tests more easily using Java Annotations.

4.2 Benchmarking

There were two types of benchmarking conducted in this project. The following two paragraphs describe these types in more detail and explain how we have used them.

4.2.1 Black box benchmarking

This is a type of benchmarking when we treat the whole algorithm as a black box. We are not interested in algorithm details here. We want to see the big picture and know the performance of the algorithm as of one single system. The algorithm just gets some inputs and produces some outputs, while we or some libraries track the time. This gives us the possibility to easily see the improvement or regression of the algorithm speed.

[2] http://testng.org/doc/index.html [Accessed 15 May 2016]
[3] http://hamcrest.org/JavaHamcrest [Accessed 15 May 2016]

To perform such benchmarking we used the same method that the author of the algorithm, Salnikov-Tarnovski, used – DaCapo benchmarking suit released in 2009. It contains a list benchmarks with non-trivial memory load based on open source, real world projects. On Figure 6 is a list of benchmarks from the official DaCapo website [11]:

avrora	simulates a number of programs run on a grid of AVR microcontrollers
batik	produces a number of Scalable Vector Graphics (SVG) images based on the unit tests in Apache Batik
eclipse	executes some of the (non-gui) jdt performance tests for the Eclipse IDE
fop	takes an XSL-FO file, parses and formats it, generating a PDF file.
h2	executes a JDBCbench-like in-memory benchmark, executing a number of transactions against a model of a banking application, replacing the hsqldb benchmark
jython	interprets a the pybench Python benchmark
luindex	uses lucene to indexes a set of documents; the works of Shakespeare and the King James Bible
lusearch	uses lucene to do a text search of keywords over a corpus of data comprising the works of Shakespeare and the King James Bible
pmd	analyzes a set of Java classes for a range of source code problems
sunflow	renders a set of images using ray tracing
tomcat	runs a set of queries against a Tomcat server retrieving and verifying the resulting webpages
xalan	transforms XML documents into HTML

Figure 6. DaCapo benchmark types

This set of benchmarks should be sufficient to validate the optimization of the algorithm. One disadvantage of it is the long time of execution. It might take up to 30 minutes to conduct this kind of benchmarking. To perform benchmarking more effectively, we implemented a special class that is hooked to DaCapo project during the execution of the benchmarks. This class called DacapoCallback[4] and allows us to do a summary of the benchmarks from several projects. In our case, it helps us to calculate an average time for a specific benchmark. It also writes the result to a file and names it with a current date. If our Lucce is used during the benchmarking, the name will have "LUCCE" prefix as well. This will help us to track any improvements and makes it easier to do a performance comparison.

4.2.2 Micro benchmarking

This is a benchmarking on a level of method or classes. This method makes it easier to find and identify local problems in the algorithm. It allows us to find bottlenecks in the algorithm, try different solutions and validate the results. In this way, we are able to work on different parts of the system and see the results much faster compared to Black box benchmarking.

To implement micro benchmarking for the algorithm we used a library called JMH[5]. It is a tool provided by the OpenJDK community for creating, executing, and analyzing micro benchmarks written in Java. JMH allows us to create high tuned micro benchmarks by using Java annotation. The tool is very configurable and has many useful features, for example warm-up of the program. Below is a snippet from one of our benchmarks (Listing 3):

```
@Benchmark
public Node CallGraphGetOrCreate(CallGraphState state) {
   return state.graph.getOrCreate(1, 1);
}
```

Listing 3. Example of a JMH benchmark

Here *@Benchmark* annotation tells JMH which method is a benchmark that has to be run. JMH also passes a state object, which we have defined before and it holds information required for the test. JMH makes it easier to write benchmarks for your code, however it still

[4] https://github.com/ostap0207/dacapo-callback [Accessed 15 May 2016]
[5] http://openjdk.java.net/projects/code-tools/jmh/ [Accessed 15 May 2016]

can be tricky and there are many things to consider. JVM itself does a lot of code optimization, which has an impact on the benchmark results. As default, Oracle JVM HotSpot has Just-In-Time compilation. This means that it does many optimizations such as inlining, constant folding, loop unrolling, and false sharing. Therefore, to avoid dead code elimination, for example, we have to return some value from the method so that JVM sees that it might be used by a caller method.

4.3 Continuous integration

In order to compare the results of the benchmarking overtime, we needed to run the tests on a machine that has the same conditions each time. In this way, we can be sure that the results are reliable and are not corrupted by some other running processes. It is also important to keep the history of the results, so that we can, for example, go back to the previous method if needed. To satisfy these two conditions, we created a continuous integration environment using the popular tool called Jenkins[6]. It runs on a virtual private server dedicated just to benchmarking. It can also run benchmarks on several branches of source code to make comparison easier. We created three Jenkins jobs for the project (see Figure 7):

- Unit tests – builds the project and runs whenever a code change is made
- JMH benchmarks – runs on demand
- DaCapo benchmarks – runs on demand

In this way, we were able to see the results and track the progress from different environments.

[6] https://jenkins.io/ [Accessed 15 May 2016]

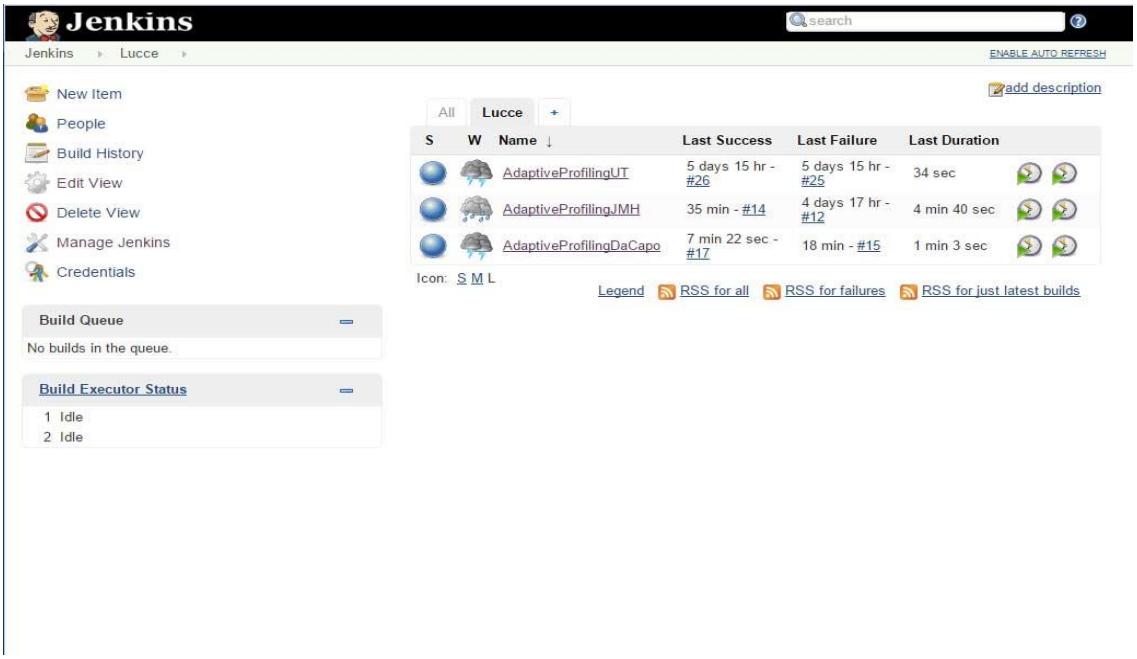

Figure 7. Jenkins dashboard

5 Preliminary benchmarking

In order to be able to validate any optimization, we need to have a snapshot of benchmark results run on our hardware. Below is a list of our benchmarks and results for unchanged version of the algorithm (Figure 8):

Benchmark	**Mode**	**Samples**	**Score(ops/ms)**	**Error(ops/ms)**
arrayContextHolderRecursion	thrpt	10	15108.158	1884.729
arrayContextHolderSequence	thrpt	10	9802.311	2589.939
GetNode.CallGraphGetOrCreate	thrpt	10	267530.500	41653.684
GraphTest.baseline	thrpt	10	756338.100	179034.223
GraphTest.createUnsafeGraphWithNodes	thrpt	10	763174.012	131408.000
GraphTest.runCallGraph	thrpt	10	14844.987	229.537
GraphTest.runGraphWithNodes	thrpt	10	85204.500	1299.930
GraphTest.runNoopGraph	thrpt	10	291639.807	15438.995
GraphTest.runUnsafeGraphWithNodes	thrpt	10	28788.708	3575.757
Traverse.traverseCallGraph	thrpt	10	6194.120	629.056
AddEdge.addEdge	thrpt	10	8866.718	1750.591
NodeTest.createNode	thrpt	10	36.729	5.817
Traverse.addEdge	thrpt	10	212034.354	17233.830

Figure 8. Preliminary JMH benchmark results

Below is an overview of the most important columns:

- Benchmark – name of a benchmark ins a format *<package>.<class>.<method>*
- Mode – defines a mode in which to show the result. Throughput mode shows number of operations per unit time. In future, some more convenient mode might be chosen.
- Samples – number of iterations per each test. At the end, an average value is calculated.
- Score – the actual result in number of operations per millisecond

We used this results as well as results of new benchmarks in order to validate the effect of changes that are made to the original implementation of Lucce.

6 Java performance tuning

At the first stage or round of optimizations, we decided to go with language and syntax specific optimization techniques. These techniques are more local and usually do not require changing the algorithm or the data flow itself. They also do not touch the instrumentation part of the algorithm. This approach was based on ideas taken from the Java Performance Tuning book [13]. While the book was published in 2000, it nonetheless has some principles that continue to be useful even now. In this chapter, we will outline which techniques were tried, which of them were successful and which failed.

6.1 Finding a bottleneck

One of the most important parts before we start any optimizations is to find a performance bottleneck. In other words, what is the part of the algorithm that JVM spends the most time on? It does not make much sense to optimize something that is already working fast or is not executed that often. For example, if a particular function takes 10% of the algorithm execution time, then a 50% optimization of this part would yield just a 5% optimization in overall. When a 50% optimization of part that takes 80% of execution time is achieved, it would result in a 40% optimization overall.

One of the ways to find the bottleneck of the whole system is to use a profiler or sampler. A profiler is a program which is attached to a Java process and injects methods, similar to what the Lucce implementation does. In this way, it can be used to see what method is executed and how long is it running. A sampler on the other hand takes snapshots of the running process in intervals and summarizes them. Standard Java 8 SDK already includes two tools that helped us to find some bottlenecks. The first one is JVisualVM[7]. It includes both a sampler and a profiler. The profiler did not work well in this instance and caused some errors during execution, possibly because the program is instrumented with our agent. The sampler, on the other hand, worked better and showed that the most busy method is *entered* (see Figure 9). This is the result we were expecting. The method *entered*, as methods *calling* and *returned*, is injected in almost every third party method and has the most code inside, compared to other injected methods. However, the JVisualVM sampler was not precise enough to provide us with sufficiently detailed information about the *entered* method. Other methods, shown in Figure 9, are from the benchmark itself.

[7] http://docs.oracle.com/javase/6/docs/technotes/tools/share/jvisualvm.html [Accessed 15 May 2016]

Figure 9. JVisualVM hot methods view

The second tool we used is Java Mission Control[8] or JMC. JMC is a commercial tool that can be used for free in the development mode. In order to be able to run it alongside the application, one has to start the observed program with two JVM flags:

```
-XX:+UnlockCommercialFeatures -XX:+FlightRecorder
```

These flags enable a feature called FlightRecorder that we are going to use. FlightRecorder observes the application, similar to the sampling for some small configurable time. In our case, we set the time to one minute. FlightRecorder allows configuring sampling with many parameters to make it less or more intensive. We decided to go with more intensive sampling, which gave us some more insights (see Figure 10).

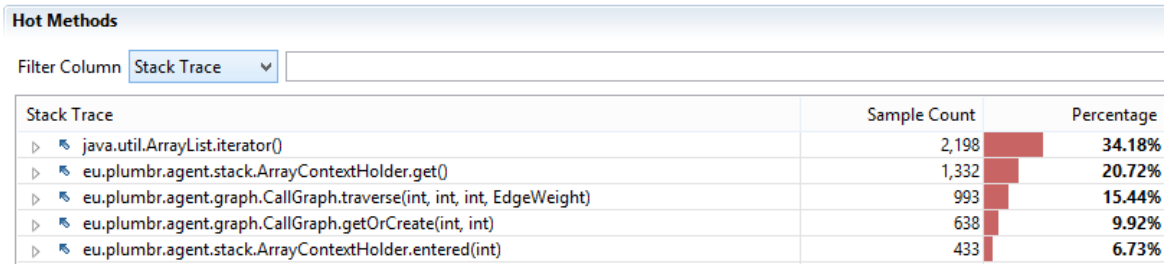

Figure 10. Java Control Mission Flight Recorder results

In addition to the *entered* method, we found three other hot methods, which need closer look. First, CallGraph#*getOrCreate* was used to get an old node from the graph or to create a new one if the node does not exist yet. Second, CallGraph#*traverse* traverses the graph and updates all the needed values. The third method, ArrayContexHolder#get, tries to get the right context for the instrumented at current moment thread. The last method is a list iterator, which the Lucce agent uses to go over all the nodes in the graph. We have tried to address this problem as well (see Chapter 6.2.4).

[8]http://www.oracle.com/technetwork/java/javaseproducts/mission-control/java-mission-control-1998576.html [Accessed 15 May 2016]

To find some more information about which part of the algorithm to look at more closely, we have run a small experiment. We removed the code from all the injected by the Lucce agent methods, *entered*, *calling* and *returned*, to see what portion of the overhead is actually taken by finding the right context for the thread and by the time of method calls. The results showed that the time of the benchmark was smaller just by about 50 – 40 %. This suggested that the instrumentation part itself is quite heavy in terms of computation and that we needed to find a way to optimize it as well.

6.2 Performance optimization techniques

Preallocating Objects

During the process of building the call graph, the algorithm has to create many objects to represent nodes and edges. The book [13] suggests that to improve the speed of creating such objects, the agent can create them in advance using an object pool. When the object is no longer needed, it can be returned back to the pool.

In our case, we created a *NodePool*. *NodePool* is a class that holds many nodes ready to be used by an agent. Its constructor accepts an initial capacity of the pool. When the pool is empty, the program populates it again. There are two ways to populate the pool: synchronous and asynchronous. *NodePoolSync* blocks the execution until the pool is full again. *NodePoolAsync* starts a separate thread to refill the pool when the pool is almost empty. In Figure 11 you can see the result of the JMH benchmarks for two types of pool and previous creating method.

Benchmark name	**Description**	**Result (ops/ms)**
testAsyncPool	Get a node from an asynchronous node pool	0.004 ± 0.001
testCreate	Create a node with constructor	0.005 ± 0.001
testSyncPool	Get a node from a synchronous node pool	0.003 ± 0.001

Figure 11. Node creating JMH benchmark results

The performance of all three methods is relatively similar. Therefore, unless we find a way to return nodes to the pool (see more in Chapter 6), it does not make sense to use any of the pools.

The benchmarks look like the one in Listing 5:

```
@Benchmark
public ArrayList<Node> testAsyncPool (AsyncState state,Blackhole bh) throws
InterruptedException {
  NodePool nodePool = state.nodePool;
  ArrayList<Node> nodes = new ArrayList<>();
  for (int i = 0; i < 1000; i++) {
    bh.consume(nodePool.get());
    Blackhole.consumeCPU(50000);
  }
  return nodes;
}
```

Listing 5. Asynchronous node pool JMH benchmark

We use the *Blackhole#consume* method provided by JMH framework to return the result of every node creation. This prevents deadcode elimination by HotSpot JVM. The *Blackhole#consumeCPU method* burns some CPU cycles. It allows us to simulate real life conditions, as the real algorithm does some other processing in between creating the nodes.

Use circuit breaker

Imagine the following '*if*' statement (Listing 6):

```
if (condition1 || condition2) {}
```

Listing 6. If statement with two conditions and operator `OR`

In the case that *condition1* is always false, JVM always has to evaluate *condition2*. On the other hand, if *condition1* is true most of the time, then JVM does not have to evaluate *condition2*. Therefore, we can manipulate the position of the conditions to reduce the amount of evaluations of unnecessary conditions. For example, in Listing 7 you can see a snippet of code from the Lucce agent that has a similar conditional statement:

```
if (lastVersion == graph.getVersion() || lastVersion >= caller.lastVersion) {
  int w = pathWeights[callSite];
  if (w != -1) {
    edgeWeight.weight = w;
    edgeWeight.graphVersion = lastVersion;
    return;
  }
}
```

Listing 7. Example of condition path optimization

At the top level '*if*' statement we have two conditions. By simply counting during the execution of the benchmarks, we were able to determine that the second condition is more likely to be true than the first one. Here are some specific numbers (Figure 12):

Condition	**Amount**
conditions1 is true, condition2 is false	0
conditions2 is true, condition1 is false	~ 2 000 000 000

Figure 12. Results of condition evaluation count

As you can see, *condition2* is never false when *contition1* is true. On the other hand, *condition1* is false many times when *condition2* is true. Therefore, it makes more sense to put the *condition2* at the beginning of the '*if*' statement, so that the JVM does not even have to evaluate *condition1*.

Array creation

In the algorithm implementation, each of the created Node objects has an array of path weights to hold the additional values of the paths. The array size is predefined and each cell is filled with '*-1'* at the Node creation time. We have tried to find and implement other methods to generate this array in a more efficient way (see Figure 13):

Method	Description
System.arraycopy	This is a native static method to copy one array into another.
Array.clone	Creates a shallow clone of an array. As our array is an array of ‘int’ (which is a primitive type), then shallow copy is good for us.
Arrays.copyOf	It is just a wrapper method around System.*arraycopy*
Arrays.fill	It is just a wrapper method around filling in a loop
Fill array in a loop	Method which was used originally

Figure 13. Different ways of filling an array with some default value

Here are the benchmark results for all the methods:

Benchmark name	Method	Result(ops/ms)
clonePathWeightArray	Array.clone	32.009 ± 10.891
copy2PathWeightArray	System.arraycopy	27.555 ± 11.484
copyPathWeightArray	Arrays.copyOf	33.732 ± 14.010
createEmptyPathWeightArray	Baseline	38.263 ± 16.505
createPathWeightArray	fill in loop	27.935 ± 11.679
fillPathWeightArray	Arrays.fill	26.598 ± 9.303

Figure 14. Array filling benchmark results

As you can see in Figure 14, the best result is the *Arrays.copyOf*. It shows a better performance than ‘*fill in loop*’ or *Array.fill*. Even *System.arraycopy* is slower than *Arrays.copyOf*. We discovered that the method *Arrays.copyOf* is an intrinsic method in Java, which means that it is specifically handled by JVM [14]. Therefore, we replaced the original ‘*for loop’* implementation with the faster and more readable ‘*Arrays.copyOf’* method.

Array iteration

Since the algorithm also has to iterate over the many arrays and collections to find the right edge or node, we tried to optimize the different ways of the iteration and compare them with the help of micro benchmarking. In order to be able to add more edges to the context graph in future, the original version of the algorithm was using *ArrayList* as a collection for the edges. The iteration was done using '*for each loop'*. We have researched some other potentially faster ways of iteration over a collection in Java (see Figure 15).

Loop type	**Example**	**Description**
For Each loop	**for** (Edge edge : state.**edges**) {…}	It is a standard Java for each loop, which was used in the original algorithm
For Loop	**for** (**int** i = 0; i < size; i++) {…}	For loop with an index
For Back Loop	**for** (**int** i = size; i > 0 ; --i) {…}	As the Java Performance Tuning [13] book suggested, the back loop might be faster as comparing to 0 is faster than to any other number.

For + try catch	**try** {
	for (i = 0; ; i++)
	bh.consume(edges.get(i));
	}
	catch (IndexOutOfBoundsException e) {}
		Here we do not have a condition at all, so we do not evaluate it. When the array ends, we get an exception.

For loop over EdgeList	EdgeList<Edge> edges = state.**edges**;
	int size = edges.size();
	for (**int** i = 0; i < size; i++)
	{
	bh.consume(edges.get(i));
	}
		EdgeList it is our own implementation of an ArrayList. The only difference is that it does not make a rage check in the *get* method.

Figure 15. Different ways of array iteration

The benchmarks showed us some good results in this case as well (Figure 16):

Benchmark name	Result(ops/ms)
forBackLoop	275.427 ± 71.310
forEachLoop	181.645 ± 61.658
forLoop	275.150 ± 56.239
forLoopEdgeList	276.808 ± 85.714
forTryCatchLoop	198.788 ± 47.268

Figure 16. Benchmark results of iteration over an ArrayList

As you can see, *forBackLoop*, *forLoop* and our own *ArrayList* implementation, without the range check, have shown the best results. *ForEachLoop*, which was used originally, shows the worst results of any of the other iteration variants. Therefore, faced with three options with almost identical performances, we decided to go with the normal *for loop* version, as it is the most readable of all the options.

In this chapter, we have found a place for four different optimization techniques, some of which developers discovered a long time. This project shows how they are still applicable to modern algorithms and for use with modern languages. Unfortunately, we did not find a usage for other techniques such as strength reduction[9] or loop unrolling[10] in the case of this algorithm.

[9] https://en.wikipedia.org/wiki/Strength_reduction [Accessed 15 May 2016]
[10] https://en.wikipedia.org/wiki/Loop_unrolling [Accessed 15 May 2016]

7 Instrumentation improvements

7.1 Instrumentation with ASM

One of the most important parts of the algorithm implementation is the instrumentation of third party code in runtime. Instrumentation is a mechanism that allows the inclusion of almost any custom code or byte code instructions in order to gather useful information about the running program. In our case, we wanted to build the calling context graph based on method invocations. Java provides a mechanism to implement such instrumentation. One has to do the following steps:

1) Implement the *transform* method of the ClassFileTransformer interface

   ClassFileTransformer[11] is an interface provided by Java, method *transform* of which is invoked by the JVM for each new class during the class loading. This method is a place to change the initial class byte code. Listing 8 contains an example. The variable *classfileBuffer* contains all the class byte code, available for reading and changing. At the end JVM expects the *transform* method to return a new version of the loaded class.

```
public interface ClassFileTransformer {
  byte[] transform(ClassLoader  loader, String  className, Class<?>  classBeingRedefined,
ProtectionDomain    protectionDomain, byte[] classfileBuffer) throws
IllegalClassFormatException;
}
```

Listing 8. Signature of ClassFileTransformet interface

2) Create a class with '*premain*' method. We have to add our implemented transformer to the Instrumentation instance inside this method. In this way, we are telling JVM what classes to use for instrumentation. It is possible to add more than one transformer.

[11] https://docs.oracle.com/javase/7/docs/api/java/lang/instrument/ClassFileTransformer.html [Accessed 15 May 2016]

```
public static void premain(String agentArgs, Instrumentation inst) {
  inst.addTransformer(new MethodEnteredAlert());
}
```

Listing 9. Example of `premain` method implementation

3) One then has to build it into the JAR file and run it with any program as a Java agent with: *java –javaagent:agent.jar –jar Program.jar*

Such flexibility enables us to do almost any byte code manipulations we might need or want. However, it is still not a simple task to change byte code in runtime, as the human brain does not understand byte code that well. Therefore, there are libraries that provide more comprehensible Java syntax. One of the tools that helps with instrumentation is ASM [15]. It has been chosen by the author of the algorithm and makes instrumentation easier.

ASM is an all-purpose Java byte code manipulation and analysis framework [15]. It provides an API that helps to change byte code during the process of class loading. Figure 17 illustrates how it works. ASM uses a Visitor pattern [18] to go through all the class and its method. First, it visits class specific sections such as class attributes, annotations or fields. After that, it visits the methods using the same principle. All the developer has to do is to implement the required Visitor interface. For example, if you want to modify a method you have to implement the *MethodVisitor* class, if you need to instrument class annotations you have to implement the *AnnotationVisitor* class and so on. If some of the interfaces are not implemented, ASM simply skips them and keeps the corresponding code untouched.

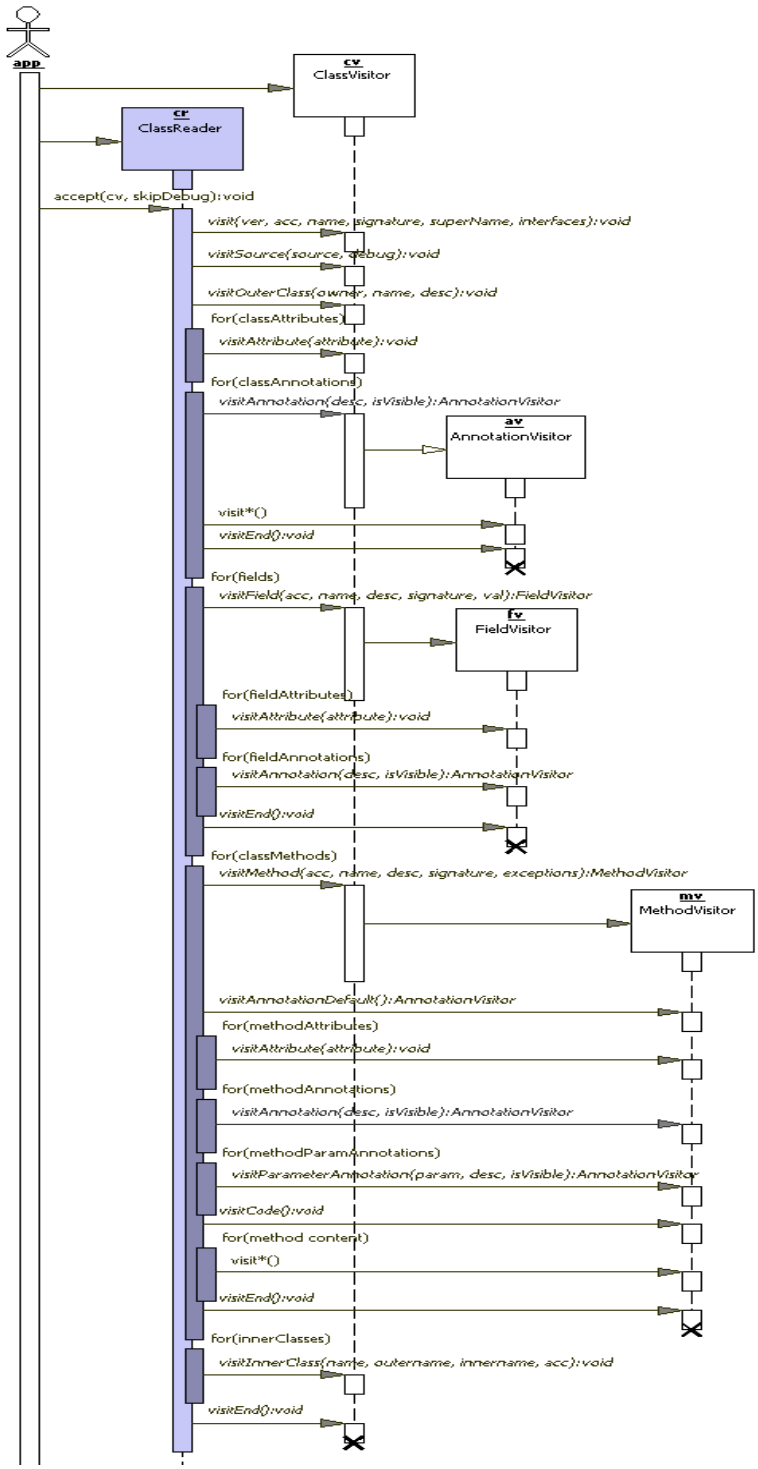

Figure 17. ASM visiting sequence diagram [15]

Here are few example from the actual implementation (Listing 10):

```
public class MethodEnteredAlertMethodVisitor extends MethodVisitor {
. . .
  public MethodEnteredAlertMethodVisitor(GeneratorAdapter mv,
                                         String sourceClassName,
                                         String sourceMethodName,
                                         String sourceMethodSignature,
                                         ClassPredicate classPredicate) {
    super(Opcodes.ASM4, mv);
    this.mv = mv;
    this.classPredicate = classPredicate;
    this.methodCode = ConcurrentMethodEncoder.encode(sourceClassName + "." +
sourceMethodName + sourceMethodSignature);
  }

  @Override
  public void visitCode() {
    super.visitCode();
    mv.visitLdcInsn(methodCode);
    mv.visitMethodInsn(INVOKESTATIC, trackerClassName, "entered", "(I)[J");
    currentContext = mv.newLocal(Type.getObjectType("[J"));
    mv.storeLocal(currentContext);
  }
. . .
}
```

Listing 10. Example of MethodVisitor implementation

As you can see, the '*MethodEnteredAlertMethodVisitor'* constructor takes the following arguments:

Argument	Description
GeneratorAdapter mv	It is a convenient class that implements the MethodVisitor interface. It allows us to generate the most common code more easily. For example, the *newLocal* and *storeLocal* methods allow us to create a local variable.
String sourceClassName	It is an owner class of the method we are visiting.

String sourceMethodName	The actual name of the method we are visiting. It is used to build a unique method identifier.
String sourceMethodSignature	Describes the input and output types of the method. It is also needed to generate the unique method id, as Java allows method overloading.
ClassPredicate	It is used to identify a class that we do not want to instrument. For example, internal Java classed are not instrumented as we are interested just in the third party code.

Figure 18. MethodVisitor incoming arguments

Below the constructor, you can see overridden method *visitCode*. This method is called at the beginning of every method and is a good place to call our `entered` method.

mv.visitLdcInsn(methodCode);	Load the constant (methodCode) on the stack.
mv.visitMethodInsn(INVOKESTATIC, trackerClassName, "entered", "(I)[J");	Invokes the method.

Figure 19. Explanation of ASM instructions

In Figure 19 you can see how injected and invoked Lucce method *entered*. The first parameter of *visitMethodInsn* is a method type and could be one of *INVOKEVIRTUAL*, *INVOKESPECIAL*, *INVOKESTATIC* or *INVOKEINTERFACE*. In our case, we call a static method. `*trackerClassName*` is the name of the class that we are going to call method from, followed by a method name. The last parameter is a signature. JVM uses special notation to make a signature more compact.

Below is a mapping of Java types to signature types (Figure 20) [16]:

Java Type	**Type Signature**
boolean	Z
byte	B

char	C
short	S
int	I
long	J
float	F
double	D
fully-qualified-class	L fully-qualified-class ;
Array of type	[type

Figure 20. Conversions from Java Type to JVM Type Signature [16]

Based on this table, signature of the method '***long****[] entered(****int*** *method)'* transforms to: *'(I)[J'* – same as we have in the example.

'visitMethodInsn' loads needed amount of values from the stack, executes the method and puts the result back to the stack.

After that we created an index for the new local variable and stored it as a local variable by calling '*newLocal*' and '*storeLocal*' respectfully. In such a way, we modified every required method to invoke *entered* method of the Lucce agent and stored a result of its execution as a local variable that the algorithm use later. More information about ASM instrumentation you can find in their Official guide[12].

The next section shows what ideas we tried to implement in order to improve the algorithm's implementation. For each idea, we explain what it tries to address, propose an implementation and benchmark results for comparison. Most of the ideas solved the given problem. However, some of them unfortunately did not work as expected.

7.2 Attempted optimizations

Calling/return wrapping elimination

In the original implementation every method call made inside an instrumented class is wrapped in a *calling* / *return* block, as described earlier in Chapter 3.2. That is not needed in the case the Lucce agent is not going to instrument the class, the method of which is called, at all. For example, if a method calls some Java system class, wrapping every method

[12] http://download.forge.objectweb.org/asm/asm4-guide.pdf [Accessed 15 May 2016]

of that class in *calling*/*return* block is redundant, because the *entered* method will not be called from this system method anyway. The change looks like this:

```
-       if (opcode == INVOKEVIRTUAL || opcode == INVOKESTATIC || opcode == INVOKEINTERFACE || opcode == INVOKESPECIAL) {

+   if (classPredicate.accept(owner) && (opcode == INVOKEVIRTUAL
+           || opcode == INVOKESTATIC
+           || opcode == INVOKEINTERFACE
+           || opcode == INVOKESPECIAL)) {
```

Listing 11. Using of class predicate

Here is a list of excluded classes:

```
excludedPackages.add("java");
excludedPackages.add("sun");
excludedPackages.add("oracle");
excludedPackages.add("com/oracle");
excludedPackages.add("com/sun");
excludedPackages.add("eu/plumbr/agent");
excludedPackages.add("ch/qos/logback");
excludedPackages.add("org/objectweb");
excludedPackages.add("org/slf4j");
excludedPackages.add("gnu/trove");
excludedPackages.add("org/codehaus");
excludedPackages.add("org/jctools");
excludedPackages.add("groovy");
excludedPackages.add("org/dacapo/harness");
excludedPackages.add("org/netbeans/lib/profiler");
excludedPackages.add("org/dacapo/harness/TeeOutputStream");
```

Listing 12. List of Java packages excluded for instrumentation

As this change would eliminate all the *calling/returned* calls for each of this classes, it would definitely speed up the algorithm.

More efficient hash map usage

As discussed earlier in Chapter 3.1, the algorithm has to encode every class method into a single number. In addition, it supposed to have the possibility to decode a number back to the class and method name. Originally, the encoding is done by increasing *AtomicIteneger* instance to produce the number and storing the result along with the method name in two maps for forward and reverse index. As this is global for all the threads, any map modifications have to work correctly in the concurrent environment. To ensure this, the author of the algorithm used Java keyword *synchronized*, which blocks a whole object for the time of modification. A better solution would be to use a special map implementation such as ConcurrentHashMap. It is an optimized for concurrency map which does not block itself for value retrievals. By using it, we do not need to synchronize the method anymore.

We compared the original and our implementations using the JMH framework. You can see the results in Figure 21. As expected, the benchmark shows that our new approach has a better performance.

Benchmark name	**Result(ops/ms)**
concurrentMethodEncoder	2557.974 ± 175.316
syncMethodEncoder	2285.331 ± 203.279

Figure 21. Method encoder benchmarks

As this change touches just instrumentation part, it might not give a performance boost during program execution, but it decreases the startup time of the application with the agent.

Storing thread id as a local variable

Any Java application can have more than one thread running at any point of time, and the Lucce agent has to deal with this as well. The author of the algorithm chose to address this by having a separate graph context for each thread. The context for each graph is stored in an array and identified by thread id. Therefore, the class that manages all Lucce contexts has the name *ArrayContextHolder*. The problem with this solution is that every time we call the *entered*, *calling* or *returned* methods, we have to find the current thread, get its id and then find the right context by getting it from the array by thread id.

To partially solve this problem we used a new approach. The idea was to store a thread id in the local method variable in the instrumented third party code. As the variable is local, it cannot be shared between the threads, but it can still be passed to other inner methods inside one thread. The implementation of this idea involves several steps:

1. Return current thread id from *entered* method together with the other, previously used, values (call id, graph version)
2. Store current thread id as a local variable using ASM.
3. Pass previously stored thread id to the following *calling* and *return* methods using ASM.
4. Change signature of methods *calling* and *returned* to accept current thread id and use it instead of calculating this id every time.

After introducing this change, getting the current thread graph inside *calling* and *returned* methods is as fast as just getting a value from an array.

Benchmark name	**Description**	**Result(ops/ms)**
arrayContextHolderRecursion	Recursive calls with the original context holder	9373.204 ± 3424.306
arrayContextHolderSequence	Sequential calls with an original context holder	10279.200 ± 4150.990
localThreadIdHolderRecursion	Recursive calls with storing thread id locally	15464.941 ± 5674.530
localThreadIdHolderSequence	Sequential calls with storing thread id locally	13928.705 ± 2954.018

Figure 22. JMH benchmark result of different context holder implementations

The benchmarks in Figure 22 are testing context holders' *entered* and *calling* methods for recursive and sequential calls. As expected, our approach shows better results than the original.

Thread state identification

As we have shown before, for every thread we store call context in the array using thread id as an index. However, what happens when the thread has finished its execution? We need to clean that array cell. The threads, we are talking about here, are created by main

program to which we attach Lucce agent, so we do not have control over it. The only thing we can do is to check the thread's state by pooling it in a separate thread. That is the approach that author of the algorithm went with.

Originally, author used method *Thread#isAlive* to check if the thread has already stopped or still running. We have discovered another way of getting thread's state such as: *Thread#getState() State*, where State is an Enum and by Java specification [19] can be one of values presented in Figure 23:

NEW	A thread that has not yet started is in this state.
RUNNABLE	A thread executing in the Java virtual machine is in this state.
BLOCKED	A thread that is blocked waiting for a monitor lock * is in this state.
WAITING	A thread that is waiting indefinitely for another thread to perform a particular action is in this state.
TIMED_WAITING	A thread that is waiting for another thread to perform an action for up to a specified waiting time is in this state.
TERMINATED	A thread that has exited is in this state.

Figure 23. Java Thread possible states

At any point in time thread can exist exclusively in one of these states.

State.TERMINATED is a state that would allow us to know when the state finished as well as *State#isAlive*. To choose the best method we wrote two simple JMH benchmarks (see Listing 12 and Listing 13):

```
@Benchmark
public boolean getState() {
  return Thread.currentThread().getState() == Thread.State.TERMINATED;
}
```

Listing 12. Get Thread state JMH benchmark

Benchmark results in Figure 24 show that using `*Thread#getState*` and checking it with `*Thread.State.TERMINATED*` is an order of magnitude faster than using `*Thread#isAlive*`.

```
@Benchmark
public boolean isAlive() {
  return Thread.currentThread().isAlive();
}
```

Listing 13. Is Thread alive JMH benchmark

Benchmark name	Result(ops/ms)
ThreadBenchmark.getState	631210.341 ± 9065.791
ThreadBenchmark.isAlive	39562.790 ± 4785.081

Figure 24. Thread state JMH benchmarks

Graph change checking

During the program execution there are situations where no new nodes or edges are being added to the call graph. In such cases, the algorithm requires fewer calculations as we do not need to update the whole graph. In the original implementation the checking, either graph has changed or not, was deep inside the algorithm and required to know both caller and callee of a method. During our research, we found a way to do this during the early stage of the algorithm in a way that required just callee.

Firstly, we restructured the code and moved the checks to a high level taking into account short-circuit evaluation. Secondly, we found out that the caller graph version is already being passed in the calling method, which means that we do not need to find the caller to get the graph version every time to check the graph, unless the graph has actually changed.

Local variable storing optimization

Originally, the *entered* method would return an array of *long* values, which would be stored as local variables in instrumented methods and passed as arguments to *calling* and *returned* methods. We decided not to store them separately, but to store just one array of

long values and pass it as an argument to other functions. This make instrumentation more readable and increases performance of the algorithm, since we do not have to create other local variables for every instrumented method.

Java Native calls

One of our ideas was to implement some parts of the algorithms as native operation system functions. This is possible with Java Native Interface or JNI. JNI is a mechanism that allows calling functions in C or C++ from Java program and the vice versa. Next we will next describe the process of writing a native implementation based on one of algorithms functions. A good candidate for a native call was graph traversal. It has the most intensive functionality and depends just on the input parameters and instance variables. To call a native function from Java code, one has to define a method as a native, without the method body as following:

```
public native void traverse(Node caller, int callSite, CallGraph graph, EdgeWeight edge-
Weight);
```

Listing 13. Declaration of native method in Java

The most difficult part is to implement the desire functionality in C or C++. First, you have to generate a header class by running the following commands:

```
javac Node.java – generates Node.class
javah Node – generates Node.h
```

Listing 14. Generation of C/C++ header file for the native method

Node.h contains something similar to:

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class eu_plumbr_agent_stack_Node */

#ifndef _Included_eu_plumbr_agent_stack_Node
#define _Included_eu_plumbr_agent_stack_Node
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     eu_plumbr_agent_stack_Node
 * Method:    traverse
 * Signature: (Leu/plumbr/agent/stack/Node;ILeu/plumbr/agent/graph/Call-
Graph;Leu/plumbr/agent/stack/EdgeWeight;)V
 */
JNIEXPORT void JNICALL Java_eu_plumbr_agent_stack_Node_traverse
  (JNIEnv *, jobject, jobject, jint, jobject, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

Listing 15. Generated header file

Here you can see the declaration for our future function. The first parameter '*env*' is an object that allows interaction with Java. The second parameter is our Node instance, an object the native method of which is invoked. The other four parameters are the same as in our original Java method. Any parameter can have just one of the predefined types that correspond to Java primitive types. For example, if a Java function accepts *int*, then a native function would accept jint. Below is a full list of available types (Figure 25):

Java type	**Native type**
Custom Complex Type	jobject
<primitive_type>	j<primitive_type>, Example: jint
< primitive _type>[]	j<primitive_type>Array, Example: jintArray
String	jstring

Figure 25. Java types to JNI types conversion table

After that, you have to create a cpp file and write an implementation for the method. Here we show how it looks at the beginning:

```
#include <jni.h>
#include <stdio.h>
#include "eu_plumbr_agent_stack_Node.h"

JNIEXPORT void JNICALL Java_eu_plumbr_agent_stack_Node_traverse(JNIEnv * env,
jobject node, jobject caller, jint callSite, jobject graph, jobject edgeWeight) {
}
```

Listing 15. Example of C/C++ body file

Using *JNIEnv* we can get any field's data or call any method we need. It requires several method calls to do that, for example to get 'node.lastVersion' requires following actions:

```
jclass NodeClass = env->GetObjectClass(node);
jfieldID fidLastVersion = env->GetFieldID(NodeClass, "lastVersion", "I");
jint lastVersion = env->GetIntField(node, fidLastVersion);
```

Listing 16. Native code for getting `node.lastVersion` value

First, we get a class of the object. Then, using *env* we get a field id specifying the field name and type using Java type signature. Finally, we can get the field's value using the type specific method GetIntField of env.

To implement all the functionality in native would require a lot of work. Therefore, we decided first to implement just a small part of it and check the benchmark results to see if it gives any improvements. Unfortunately, our partial native implementation did not improve the performance in a significant way, therefore we decided to put this idea aside.

Node pool

Previously, in Chapter 6.2, we described how we tried to eliminate node creation with a node pool. We found a way to get nodes from the pool, but we could not return them back. In this section, we will describe how we were actually able to release nodes and put them back to the pool and what performance improvements it gave us.

The first step was to implement the cleaning mechanism. This can be done by implementing the Composite pattern [20]. Our original class structure is well suited for this. All we had to do is to add a method *clear* to all the three main classes, such as *Context*, *Graph* and *Node*, and to implement these method according to the Composite pattern. Our final class diagram looks like this (Figure 26):

Figure 26. Implementation of 'Composite pattern'

'Context#clear' calls '*Graph#clear'* which calls '*Node#clear'*, which can potentially clear the edges as well.

The next was to decide where and when to clear the graph. The right direction would be when the thread has finished its execution. Originally, to find out when the thread has been terminated, it was monitored by another thread that checks all the running threads and frees resources for dead threads. Another good implementation would be if we did not have to monitor the threads at all and every graph knew itself when the thread has finished.

Since there were many ideas, which we wanted to try here, some of which worked together and some of which did not, we decided to run trials between different implementations and decide the winner at the end based on the results. Here are all the ideas that took part in the trials:

1. Monitor cleaning – this is the improvement with the least changes after introducing context cleaning. The only change is that instead of deleting old contexts in the monitor thread, this version would clean them and put nodes back to the node pool, where they can be reused.
2. Context pre-initialization – every context is stored in the array by threadId. Originally, to get a context every time, the algorithm had to check if the context had already been created and if not, create it. To eliminate this check we could just create a context for every array cell and use the same Monitor cleaning idea.

3. Edge pool utilization – this is the same as Monitor cleaning, but with a change that every node would not delete the edges, but put them back to the Edge pool.
4. Depth analysis – we found a way to remove the monitor by checking the depth of the graph in the returned method. When *depth* is zero it means that the program execution came to the root of the thread, which means that all the previous nodes are not needed and can be recycled.
5. Returned redesign – this is a redesign of the way the *return* method is invoked. We will describe in greater detail in the next section.

Trial 1

Candidates\Benchmark	**Batik**	**Avrora**
Monitor cleaning	1745	14014
Context pre-initialization	1801	13912
Winner	Monitor cleaning	Monitor cleaning

Figure 27. Trial 1 results

Trial 2

Candidates\Benchmark	**Batik**	**Avrora**
Monitor cleaning	1770	12587
Edge pool utilization	1805	13847
Winner	Monitor cleaning	Monitor cleaning

Figure 28. Trial 2 results

Trial 3

Candidates\Benchmark	Batik	Avrora
Monitor cleaning	1815	12626
Depth analysis	2079	13786
Winner	Monitor cleaning	Monitor cleaning

Figure 29. Trial 3 results

Trial 4

Candidates\Benchmark	Batik	Avrora
Monitor cleaning	1849	12645
Returned redesign	2072	13754
Winner	Monitor cleaning	Monitor cleaning

Figure 30. Trial 4 results

The results among the trials can vary a great deal because of the different system load between the runs. Nevertheless, the winner of the trials is Monitor cleaning. It is the first and the easiest improvement, which has given us the best results.

Returned redesign

In the previous chapter we described how we wanted to clear the call contexts with Depth analysis and Returned redesign. The reason why Depth analysis has not shown better results might be that we clean the graph too often. This is happening because, in fact, every method is wrapped in calling/return block, except the first ever called method of the thread or program. In addition, because of that, we do not know when the first method has finished its execution. To fix that we decided to make a small redesign of the instrumentation. Using the ASM library we moved *return* to the end of every method, similar to *entered* (see Listing 4 and Listing 17). In this way, we were able to know when the first method finished the

execution. Unfortunately, even with this change it still could not beat the Monitor cleaning version.

```
private Order createOrder(User user, Order order) {
//int callId;
// int graphVersion;
//[callId, graphVersion] = Lucce.entered(methodId);
order.setUser(user);
order = repo.save(order);
//Lucce.returned(methodId);
 return order;
}
```

Listing 17. Instrumentation after the Returned redesign change

Context holder inlining

As we demonstrated earlier, there is an intermediary class between instrumentation and algorithm called ArrayContextHolder. All the calls from the third party application go through it. We wanted to remove it by inlining its functionality into instrumentation. Instead of calling ContextHolder, we would call directly calling and returned methods, which should have in theory saved us some time. The hardest part was to get the current thread id using ASM. Below is a snippet, which does this:

```
mv.visitFieldInsn(GETSTATIC, trackerClassName, "array", "[L" +
defaultCallContextName+";");
int array = mv.newLocal(Type.getObjectType("[" + defaultCallContextName));
mv.storeLocal(array);
mv.loadLocal(currentContext);
mv.visitInsn(ICONST_2);
mv.visitInsn(LALOAD);
mv.visitInsn(L2I);
int currentThreaId = mv.newLocal(Type.INT_TYPE);
mv.storeLocal(currentThreaId);

mv.loadLocal(array);
mv.loadLocal(currentThreaId);
mv.arrayLoad(Type.getObjectType(defaultCallContextName));
callContext = mv.newLocal(Type.getObjectType(defaultCallContextName));
mv.storeLocal(callContext);
```

Listing 18. Inlining of *entered* method using ASM

First, we get a value of the static variable with the name *array* and type *long[]*. After that we create a new local variable and store it locally. At line number 8, we create a new local variable currentThreadId and store its value there. Then we get a call context for the current thread and store it locally, so it could be used by *calling* and *returned* methods later. Unfortunately, in reality this change did not show any improvement. One possible reason for that is that JVM already does some code optimizations or inlining of static methods.

8 Results

To do a final validation of the improvements and optimizations made to the algorithm, we used DaCapo benchmark suit version 9.12 (released in 2009) and a similar process to the one the author of the algorithm used. Each of the benchmarks has been run with our improved agent, with the original agent and without an agent at all. The number of iterations is 31, where the first iteration result is always rejected, as it is a warm-up. All the calculations were done using two Java versions: Java 7 and Java 8 on a laptop with following characteristics (Figure 30):

Model	MacBook Pro
Processor name	Intel Core i7
Processor speed	3,1Ghz
RAM	16 GB

Figure 30. Characteristics of the environment for running the benchmarks

To make all the calculations run automatically, we wrote a small bash script (see Appendix 11.1) that runs all the required benchmarks. At the end, we got a file with a result for every benchmark. To concatenate all the files we used a UNIX command *paste* (see Listing 19).

```
paste –delimeters “\n” * > results.txt
```

Listing 19. Concatenate all results in one file

As the benchmark results we received were similar across the two Java versions, in Figure 31 you can see the results only for Java 8 as it is the most recent version. We normalized every benchmark by its time without the agent. There is a definite performance improvement on all the benchmarks. However, the improvement differs for the various benchmarks, as well as differs an actual overhead of the Lucce agent. In Figure 32, there is a percentage comparison of the optimization. The smallest percentage of improvement of around 20% is for *batik*. The reason for this can be that the *batik* benchmark - as we later checked - does not support concurrency, which means that our optimizations for multithread

applications do not have a big effect. The biggest optimization is for the *tomcat* benchmark at around 58%.

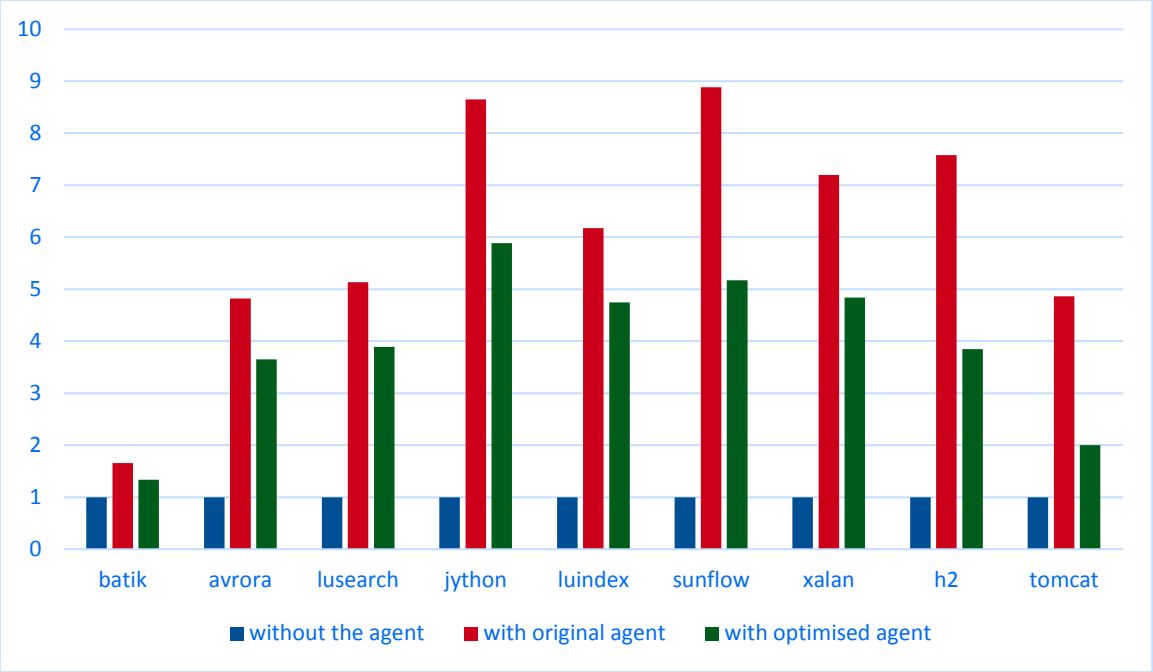

Figure 31. Results of DaCapo 9.12 benchmarks

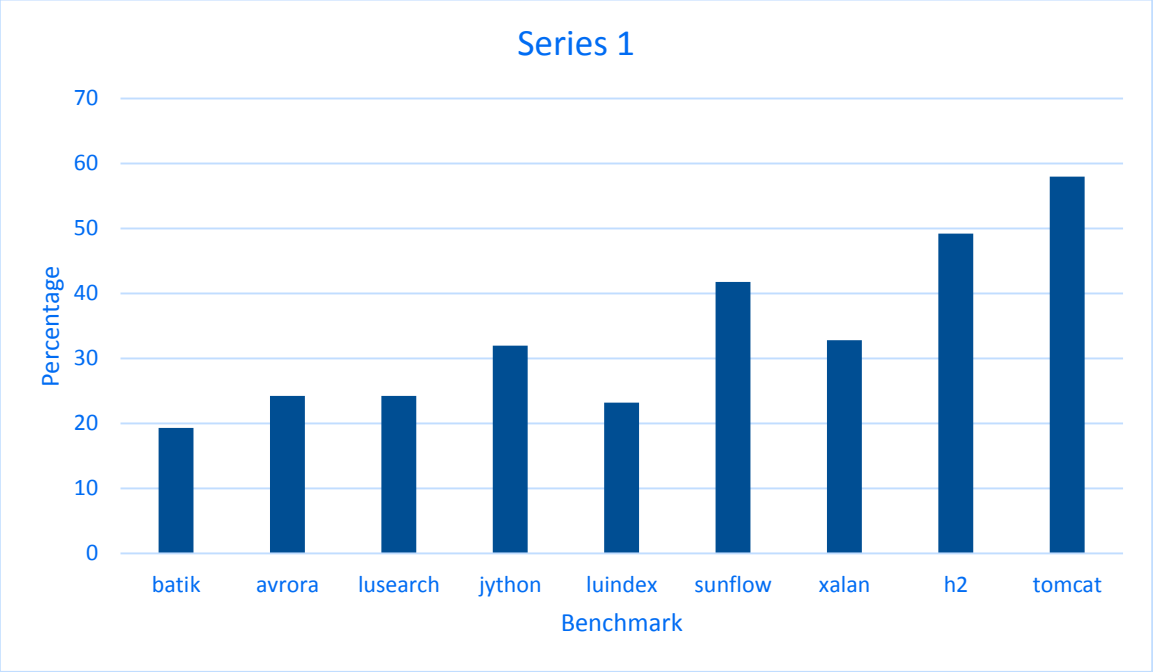

Figure 32. Lucce performance optimization results

9 Conclusions

This thesis investigated a promising new algorithm from a family of call context encoders called Lucce. The main idea of the algorithm is to store a one number with each method as a local variable, which can be decoded to a full stack trace at any time. The advantages of such encoding over other similar algorithms were outlined at the beginning of the thesis. Unfortunately, the first version of the implementation developed by the author of the algorithm is not fast enough for the algorithm to be used in the production environment.

Both the theoretical and practical aspects of implementation were presented in order to provide some background for further reading. We then proceeded to build a validation methodology, which included unit tests, black box and micro benchmarks, and the continuous testing environment. This enabled us to be more confident in the validation of our results and reduced the time required to run the benchmarks.

Prior to start any optimization we tried to find the bottlenecks, as they are the best place to start the optimization. We researched old and new optimization techniques and applied them to the implementation. Firstly, we tried to find more local optimizations, without moving code around with the help of micro benchmarks. Secondly, we came up with several ideas on the level of instrumentation. We verified them by running both types of benchmarks. Using a system version control, we were able to develop different ideas in parallel, compare their results and merge them at the end if needed. This also means that the approach developed in this thesis is accessible and traceable for other scientists and developers interested in the Lucce agent.

At the end, we ran all the benchmarks again with different versions of Java and collected the results. The overall performance improvement ranged from between 20 to 58 percent, depending on the benchmark and benchmark concurrency usage. The results of the benchmarks were passed to the author of the algorithm and met with positive feedback. Considering that the aim of the improvements was not to change the actual logic of the algorithm, the received results can be considered pretty good. Further work to improve the algorithm even further might extend to changing the logic of the algorithm and reducing the work that is done, for example storing edge values in the call sites and detecting graph changes even earlier in the algorithm process.

The whole optimization process proposed and described earlier in this thesis is not attached just to the Lucce agent. The discussed stages of the optimization such as unit testing, black box and micro benchmarking, finding a bottleneck, local optimization technics and instrumentation improvements can be used to optimize any Java agent as well as any Java program in general, by excluding the instrumentation part.

10 References

1. N. Salnikov-Tarnovski and V. Vojdani. Lazy uninstructive calling context encoding for Java. 2015. [ONLINE] Available at:
https://bitbucket.org/plas/lucce/downloads/lucce.pdf [Accessed 18 May 2016]

2. T. Ball and J. R. Larus. Efficient path profiling. In MICRO 29, pages
46-57. IEEE Computer Society, 1996.

3. N. Salnikov-Tarnovski and V. Vojdani. Lazy uninstructive calling context encoding for Java, page 17. 2015. [ONLINE] Available at
https://bitbucket.org/plas/lucce/downloads/lucce.pdf [Accessed 18 May 2016]

4. T. Ball and J. R. Larus. Efficient path profiling. In MICRO 29, page 45. IEEE Computer Society, 1996.

5. W. N. Sumner, Y. Zheng, D. Weeratunge, and X. Zhang. Precise calling
context encoding. In ICSE '10, pages 525-534. ACM, 2010.

6. Q. Zeng, J. Rhee, H. Zhang, N. Arora, G. Jiang, and P. Liu. DeltaPath: Precise and scalable calling context encoding. In CGO '14, pages 109-119. ACM, 2014.

7. T. J. Watson Libraries for Analysis (WALA). [ONLINE] Available at: http://wala.sourceforge.net/. [Accessed 15 May 2016]

8. S. Chiba. Javassist - a reflection-based programming wizard for java. In Proceedings of the ACM OOPSLA Workshop on Reflective Programming in C++ and Java, 1998.

9 . S. M. Blackburn et al. The DaCapo benchmarks: Java benchmarking development and analysis. In OOPSLA '06, pages 169-190. ACM, 2006.

10. T. Kaczanowski. Practical Unit Testing. Cracow, 2012.

11. DaCapo Project. The DaCapo Benchmarks, 2009. [ONLINE] Available at http://www.dacapobench.org [Accessed 15 May 2016]

12. Microsoft. Introduction to Instrumentation and Tracing, 2016. [ONLINE] Available at: https://msdn.microsoft.com/en-us/library/x5952w0c(v=vs.110).aspx

13. J. Shirazi. Java Performance Tuning. O'Reilly Media, 2000

14. K. Mok. Intrinsic Methods in HotSpot VM, 2013. [ONLINE] Available at: http://www.slideshare.net/RednaxelaFX/green-teajug-hotspotintrinsics02232013

15. ASM. 1999-2009. [ONLINE] Available at: http://asm.ow2.org/. [Accessed 15 May 2016]

16. Oracle and/or its affiliates. JNI Types and Data Structures. 1993, 2016. [ONLINE] Available at:

http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/types.html#wp276 [Accessed 15 May 2016]
17. E. Bruneton. ASM4.0, A Java bytecode engineering library
18. E. Gamma, R. Helm, R. Johnson and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional. pages 366-381. 2004
19. Oracle and/or its affiliates. Enum Thread.State. 1993-2016. [ONLINE] Available at: https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.State.html [Accessed 15 May 2016]
20. E. Gamma, R. Helm, R. Johnson and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. pages 183-195. Addison-Wesley Professional, 2004

11 Appendix

11.1 Bash script to run DaCapo benchmarks

```
1.  key="$1"
2.
3.  case $key in
4.      -a|--algorithm)
5.      ALGORITHM="$2"
6.      ;;
7.  esac
8.
9.  echo ALGORITHM  = "${ALGORITHM}"
10.
11. for benchmark in batik avrora h2 jython luindex luserach sunflow tomcat xalan
12. do
13.    echo Running $benchmark
14.    java -Xmx4g -javaagent:./agent-${ALGORITHM}.jar -jar ./dacapo-9.12-bach.jar \
15.    -n 31 $benchmark -c LucceCallback
16. done
```

11.2 Glossary

Instrumentation Refers to an ability to monitor or measure the level of a product's performance, to diagnose errors and to write trace information [12].
Garbage collection Process of allocation, managing and releasing memory for a program.
Method overloading Allows us to created several functions with the same name, but with different incoming and out coming arguments.
Short-circuit evaluation A kind of boolean expressions evaluation in which every next condition is evaluated, just in case the previous condition could satisfy the expression result.
Hook A method or API call that is included in the application code to trigger some external event without changing the actual program flow.
Chord An edge in the graph that does not belong to the spanning tree of this graph.

11.3 License

Non-exclusive licence to reproduce thesis and make thesis public

I, **Ostap Maliuvanchuk** (date of birth: 02.07.1993),

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:

    1.1. reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and

    1.2. make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright,

of my thesis

Performance optimization of a Java instrumentation agent for calling context encoding,

supervised by Vesal Vojdani and Nikita Salnikov-Tarnovski,

2. I am aware of the fact that the author retains these rights.

3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, **23.05.2016**