



DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Informatics

# **Application of Performance Data from Production during Software Development**

Fritz Lumnitz







DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Informatics

# **Application of Performance Data from Production during Software Development**

## **Einsatz von produktiven Performanzdaten während der Softwareentwicklung**

Author:	Fritz Lumnitz
Supervisor:	Prof. Dr. Dr. h.c. Manfred Broy
Advisors:	Dr. Elmar Jürgens Dr. Benjamin Hummel
Submission Date:	May 15, 2021



I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich,

---

Fritz Lumnitz

## Acknowledgments

First of all, I would like to thank my advisors Benjamin, Andreas and Elmar for their guidance, helpful insights and valuable feedback. You made writing this thesis a great experience for me.

A big thanks goes to the CQSE GmbH, for allowing me to write this thesis in the first place. Also thank you to all the colleagues that participated in my user study.

I would also like to thank my colleagues at Munich Airport and especially my superior Martin, for providing me with the flexibility, which made this thesis and even my complete masters degree possible.

I also thank my proofreaders Nadja, Victoria and Julian. Without you, this thesis would definitely be a heap of comma errors and weird sentences.

Finally, my biggest thanks go to my girlfriend Nina. Thank you for keeping me sane during this year and for the encouragement and distraction, when I thought (again) I am facing an unsolvable problem.



# Abstract

Profilers are typically deployed by developers to detect slow or heavily used parts of code. The challenge is the construction of suitable input data, which reveals the suspected problem and is representative of the actual application. In addition, profilers are usually only applied when there is a concrete problem, as they require additional work from the developer. This thesis proposes a methodology, where the respective application is continuously profiled while running in the production environment. Since this data is generated during actual use, it is automatically representative.

The methodology uses the low-overhead sampling profiler provided by the Java Flight Recorder, which in term is integrated into the Java Virtual Machine. The profiling data is then collected into a calling context tree, which allows context aware analysis and metrics computation. The computed metrics are then presented to the developer, by integrating them at source code level into their IDE. Small word-sized graphics visualize the required runtime at each method declaration and invocation. Further runtime information, like the implementation class behind an interface, is included via various popup menus. The idea of an in situ visualization is to avoid the split-attention effect, which is caused by the presentation of required information over multiple views.

As the profiling data is gathered over a great amount of time (multiple hours, days or even weeks), the resulting performance information can be too generalized for cases, where only the information about small subsystems is required. To tackle this problem, the developer can reduce the focus to a sub-system, by utilizing a novel approach named *tasks*. This allows a metrics calculation relative to the respective task-roots, rather than the global system.

The profiling method was evaluated in a benchmark, which indicates that the performance and memory overhead induced by the profiler is low enough to be used in a real-world production environment. The final methodology was evaluated in a user-study, which showed, that the inclusion of runtime information into the IDE, can be helpful for program understanding, independent of the underlying programming task. It further revealed the need for a functionality to reduce the scope of the displayed profiling data. However, the proposed task based method seemed to be rather unhelpful due to high performance costs.





# Contents

<b>Acknowledgments</b>	<b>iii</b>
<b>Abstract</b>	<b>v</b>
<b>List of Abbreviations</b>	<b>ix</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Contribution . . . . .	2
1.2 Outline . . . . .	2
<b>2 Fundamentals</b>	<b>5</b>
2.1 Computer Performance . . . . .	5
2.2 Teamscale . . . . .	6
<b>3 Approach</b>	<b>9</b>
3.1 Gathering performance data . . . . .	9
3.1.1 Instrumenting profiler . . . . .	9
3.1.2 Sampling profiler . . . . .	10
3.1.3 The Java Flight Recorder . . . . .	11
3.2 Collecting performance data . . . . .	12
3.2.1 Call Tree . . . . .	12
3.2.2 Call Graph . . . . .	14
3.2.3 Calling Context Tree . . . . .	14
3.3 Evaluating the CCT . . . . .	17
3.3.1 Recursion detection . . . . .	18
3.3.2 Metrics calculation . . . . .	19
3.3.3 Task based evaluation . . . . .	22
3.4 Display performance results . . . . .	23
3.4.1 In situ visualization . . . . .	24
3.4.2 Integrating code changes . . . . .	26
3.4.3 Performance comparison with baseline . . . . .	26
<b>4 Implementation</b>	<b>29</b>
4.1 Architecture . . . . .	30
4.2 Process Integration . . . . .	30
4.3 Teamscale Integration . . . . .	31
4.3.1 Daemon Upload . . . . .	31
4.3.2 CCT representation . . . . .	32

4.3.3	Analysis Steps . . . . .	33
4.3.4	Source code mapping . . . . .	35
4.3.5	Source code changes . . . . .	36
4.4	IDE Integration . . . . .	37
<b>5</b>	<b>Evaluation</b>	<b>39</b>
5.1	Research questions . . . . .	39
5.2	Performance overhead . . . . .	40
5.2.1	Study design . . . . .	40
5.2.2	Results and discussion . . . . .	42
5.3	User study . . . . .	43
5.3.1	Study design . . . . .	43
5.3.2	Results and discussion . . . . .	45
5.3.3	Threats to validity . . . . .	50
<b>6</b>	<b>Related work</b>	<b>51</b>
6.1	Profiling . . . . .	51
6.2	Data structures . . . . .	52
6.3	CCT evaluation . . . . .	53
6.4	Performance visualizations . . . . .	55
6.5	IDE integration . . . . .	57
<b>7</b>	<b>Conclusion</b>	<b>59</b>
7.1	Future work . . . . .	59
	<b>List of Figures</b>	<b>61</b>
	<b>List of Tables</b>	<b>63</b>
	<b>Bibliography</b>	<b>65</b>

## List of Abbreviations

CG	call graph . . . . .	14
CCT	calling context tree . . . . .	14
CT	call tree . . . . .	12
HCCT	hot calling context tree . . . . .	53
IDE	integrated development environment . . . . .	2
JDK	Java Development Kit . . . . .	11
JFR	Java Flight Recorder . . . . .	11
JIT	just in time . . . . .	9
JVM	Java Virtual Machine . . . . .	9
JVMTI	Java Virtual Machine Tool Interface . . . . .	10
k-CCF	k-calling context forest . . . . .	52
PSI	Program Structure Interface . . . . .	25
PSS	partial stack sample . . . . .	15
RR	re-enable ratio . . . . .	51
UUID	universally unique identifier . . . . .	32
VCS	version control system . . . . .	6



# 1 Introduction

The runtime information of a program can be used in a variety of fields, such as function inlining [13], statistical bug isolation [28], object allocation analysis [35], or anomaly based intrusion detection [12, 17]. The collection of runtime data is typically carried out using so-called *profilers*. Profilers instrument the program at compile or runtime, and measure for example the execution time and invocation count of methods, or use a statistical approach and repeatedly sample the call stack.

Profilers can also be deployed by the developer to detect slow or heavily used sections of code in a program. The challenge here is the construction of suitable input data for a profiling run. This data should be able to reveal suspected performance problems well. However, it should also represent an actual usage scenario of the application. In addition, profilers are usually only used when there is a specific performance problem to be solved, as they require additional work from the developer. This makes it more difficult to proactively avoid performance problems.

An alternative approach is the measurement of performance data within the production system using continuous profiling. Since this data is generated during actual use of the respective program, it automatically represents a real-world usage scenario. However, challenges that arise with continuous profiling include:

- Profiling imposes an additional runtime overhead on the profiled application. This overhead should lie within reasonable limits, in order to be usable for applications running in production.
- Continuous profiling leads to a huge amount of data, which needs to be filtered and interpreted by a human operator or developer, usually utilizing a variety of different tools.
- This multitude of tools requires the developer to split their attention in order to completely understand the displayed information. This *split attention effect* [5], hinders information processing, and increases the cognitive load on the developer, slowing down the development process [39].
- The short lifecycle of the profiled program code introduces another challenge: Performance data measured yesterday may not be applicable anymore today, as the relevant code changed.

Even if a slow section in a program can be identified, it is necessary to understand the code and context connected to this section. For example, a sorting algorithm was identified, accounting for a majority of the applications runtime. There are several possibilities why this sorting algorithm is not performing: First, the algorithm

performs poorly because of an implementation bug. Second, the input set is very large, which results in the sorting algorithm needing a large amount of time. The calling context from where the algorithm is used, has to be considered in order to decide which of the two scenarios occurred.

### 1.1 Contribution

The aim of this thesis is to propose a method for the integration of performance data of Java (Version 11<sup>1</sup>) programs into the development process. Performance data is, hereby, automatically gathered from production. For this, an approach to instrument the Java Flight Recorder in order to continuously profile an application is proposed. Using this approach, the performance overhead induced on the profiled application lies at 1–2%. In addition, this thesis proposes the aggregation of the produced profiling data by utilizing a calling context tree, and utilizes various computations such as recursion detection and the merging of partial stack samples. This representation allows the computation of context specific metrics, while still maintaining a reasonable space overhead. In order to reduce the focus of the performance data from the global system to a user defined sub-system, a novel approach named *tasks* is introduced.

To make the resulting performance data easily available for the developer, a methodology to integrate the performance data into the integrated development environment (IDE) is proposed. This is based on the visualization proposed by Beck et al., and focuses on a black- and white-box presentation of performance metrics at method level directly in the source code [7]. This approach is expanded (as already proposed, but not implemented by Beck et al.) to allow the comparison of the current performance to a baseline version. In order to deal with source code changes, a visual indicator is provided in case the displayed source code changed since the profiled version.

The resulting methodology is evaluated for use during software development, in a user study.

### 1.2 Outline

The remainder of this thesis is structured as follows: Chapter 2 introduces the definition of performance as it is used throughout this thesis. It further introduces the Teamscale program, which provides the analysis engine used for the incremental analysis of the profiling data. The general method for collecting, gathering, processing and finally displaying performance information is discussed in Chapter 3. Chapter 4 describes the principles and limitations of the final implementation. The final tool was tested in a user study, Chapter 5 outlines the study, and discusses results regarding the use of performance information during software development. Chapter 6 discusses related work in the fields of profiling, data structures and evaluations, performance

---

<sup>1</sup><https://openjdk.java.net/projects/jdk/11/>

visualizations and finally other IDE integrations. Finally, Chapter 7 concludes this thesis.





## 2 Fundamentals

The following chapter describes some fundamentals for this thesis. Section 2.1 provides the definition of computer performance, as it is used in this thesis. Section 2.2 introduces the already mentioned *Teamscale* platform.

### 2.1 Computer Performance

Usually, performance is a non-functional requirement that software needs to fulfill. It can consist of multiple factors, for example:

- Elapsed Time, i.e., how long does a certain (batch) task need to be accomplished
- Throughput, i.e., how much work can be accomplished in a certain time
- Response Time, i.e., how much time elapses between sending a request to receiving the response

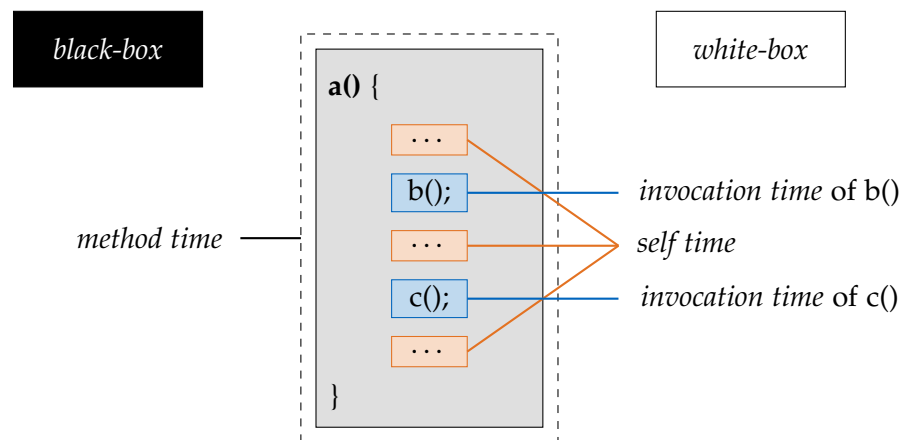


Figure 2.1: Illustrating the black-box and white-box perspective, based on [7, Figure 2]

In addition, other factors like memory consumption, bandwidth, or availability may also count towards the performance of a computer program. Arnold O. Allen defined computer performance as following:

The word performance in *computer performance* means the same thing that performance means in other contexts, that is, it means "How well is the computer doing the work it is supposed to do?" ([1])

This thesis focuses on the aspect of runtime consumption, i.e., which parts of the program spent the most time being actually executed. For this, different metrics targeted at method level exist. They provide information by analyzing a method from two perspectives, which is illustrated in Figure 2.1 [7]:

- *Black Box*: When analyzing a method as a black box, only one thing is relevant: How much execution time is spent executing this method. This is usually denoted as *method time* or *inclusive time*. Depending on the selected profiling approach (Section 3.1), this metric is calculated differently (see Subsection 3.3.2).
- *White Box*: Looking into a method, the execution time consists out of multiple factors. These can either be statements executed directly inside the method, which contribute to the so called *self time* or *exclusive time*, or calls to other methods, where each call denotes its own so called *callee time* or *invocation time*.

These two perspectives complement each other, as the method time can be used to determine, whether a method may be problematic, as well as the self time and invocation times explain where (and to which extent) the execution time is spent inside the (problematic) method. In addition, it may also be interesting to understand, from where the method is called, and coming from which call how much execution time is spent. This is described in the so called *caller time*.

## 2.2 Teamscale

*Teamscale* is a commercial software offered by CQSE GmbH (Continuous Quality in Software Engineering) and provides static source code analysis.

The main feature of Teamscale is its incremental analysis engine. This engine is connected to the version control system (VCS), which allows the analysis of each commit incrementally. This allows faster feedback and helps to reveal the root causes (commit-based) for problems.

In addition, it can be integrated into other common software used during the whole software life-cycle. For example, bug and issue trackers (e.g., Jira<sup>1</sup>) allow the provision of links between change requests, commits and changes to the overall quality system. DevOps platforms (e.g., Gitlab<sup>2</sup>, Github<sup>3</sup>, Bitbucket<sup>4</sup>) allow Teamscale to provide direct feedback in the respective merge requests. The results can also be displayed in various IDEs (e.g., IntelliJ IDEA<sup>5</sup>, Eclipse IDE<sup>6</sup>, Netbeans<sup>7</sup>), and even not yet committed code can be analyzed, in order to detect and remove findings before they even enter

---

<sup>1</sup><https://www.atlassian.com/software/jira>

<sup>2</sup><https://about.gitlab.com/>

<sup>3</sup><https://github.com/>

<sup>4</sup><https://bitbucket.org/>

<sup>5</sup><https://www.jetbrains.com/idea/>

<sup>6</sup><https://www.eclipse.org/ide/>

<sup>7</sup><https://netbeans.org/>

the VCS. Test coverage tools (e.g., JaCoCo<sup>8</sup>, Cobertura<sup>9</sup>, gcov<sup>10</sup>) provide the necessary input for the test gap analysis [45] and test impact analysis [26]. In case the Teamscale internal analysis is not sufficient, other external analysis tools (e.g., Clang-Tidy<sup>11</sup>, Pylint<sup>12</sup>) can be integrated into Teamscale.

---

<sup>8</sup><https://www.jacoco.org/>

<sup>9</sup><https://cobertura.github.io/cobertura/>

<sup>10</sup><https://gcc.gnu.org/onlinedocs/gcc/Gcov.html>

<sup>11</sup><https://clang.llvm.org/extra/clang-tidy/>

<sup>12</sup><https://pypi.org/project/pylint/>



## 3 Approach

The following chapter describes the basic approach used for the later implementation. Section 3.1 describes how performance data can be gathered from a running process, while implying as little performance overhead as possible. Section 3.2 is dedicated to the data structures used to efficiently collect the previously gathered performance data. Section 3.3 focuses on how various metrics can be calculated upon the resulting data structure. Finally, Section 3.4 presents how the calculated metrics can be displayed to the developer directly inside the IDE.

### 3.1 Gathering performance data

The first step of analyzing the performance of a program is to gather performance data about the program. This is usually performed by using profilers. The following section describes the two basic approaches of profilers. *Instrumenting profiler* are explained in Subsection 3.1.1 and *sampling profiler* in Subsection 3.1.2. The Java Virtual Machine (JVM) internal profiler, the Java Flight Recorder, is described in Subsection 3.1.3.

#### 3.1.1 Instrumenting profiler

Instrumenting profilers work by changing the byte-code of the program either at compile or at runtime, typically by using Java Agents. The byte-code is enriched with additional code that tracks the enter and exit of methods. In addition, it is possible to gather information about the method arguments, return value, and (possible) thrown exceptions.

One problem of instrumenting profilers is the altered runtime behaviour. As additional code is added by the profiler, the resulting code is larger than the original code. This can lead to a runtime difference, as the additional code has to be executed. It may also change decisions made by the just in time (JIT) compiler, which can further alter the runtime behaviour. For example, the compiler may inline small methods to avoid the overhead of a method call and return. A previously inlined method may be too large to be inlined, when enriched with the additional code by the profiler. This may cause the profiler to report a longer time for the method, as it would actually require in the unprofiled application. Method inlining is only one example of the decisions made by the JIT compiler. A general rule is that, the more code is changed by the profiler, the more the actual execution profile will change. In addition, the execution is typically much slower than the unprofiled application.

Therefore it is advised to use an instrumenting profiler only as a second step, after it is already clear at which point the profiling needs to be more accurate.

### 3.1.2 Sampling profiler

Sampling is the other approach used by profilers. It carries far less overhead than the instrumenting approach, however it is also less accurate.

Sampling profilers work by using an internal timer to periodically sample (i.e., ask) the JVM at which point the execution currently is. In addition to the current location, a stack trace is often reported, to provide more insight about the execution location. The method on top of the stack is then charged with the execution time since the last sample. Different approaches exist about to which extent the method is charged (for example, see [27]). The profiler is then executed over a long period of time, to get statistical information about what code is executed the most. One mayor drawback of sampling profilers is the lack of distinction, whether the sampled method is executed often or takes a long time to execute.

As sampling is less intrusive than instrumenting, and requires almost no overhead (see Subsection 3.1.3), this approach can be used in a production environment. Unfortunately, the statistical approach can lead to different sorts of errors.

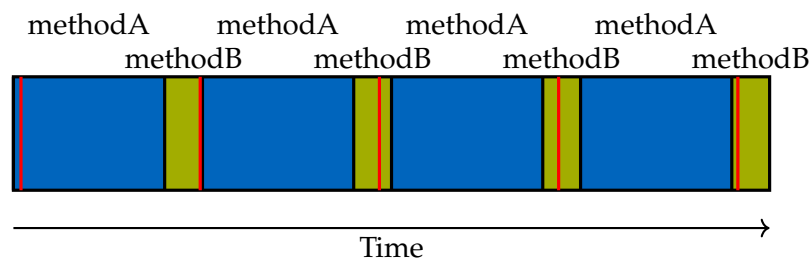


Figure 3.1: Alternating method execution

methodA (blue) alternates its execution with methodB (green), the red lines indicate sampling points.

The most prominent example is the *sampling bias* for alternating methods. Figure 3.1 illustrates a thread which alternates between two methods: methodA, which requires the greater amount of time, and methodB, with only small executions in between. The red lines indicate the sampling points. As it can be seen, a great amount of time is spent executing methodA. However, the profiler will report that methodB requires 80% of the execution. This is only an artificial example, however different sampling profilers will report different performance results [34].

Another problem with sampling profilers arises specifically for the JVM: the *safepoint bias*. Usually, sampling is performed using the Java Virtual Machine Tool Interface (JVMTI) function `JvmtiEnv::GetAllStackTraces`. This function performs a thread dump for **every** live thread in the JVM. The problem is that, “[a]ll stacks are collected

simultaneously, that is, no changes will occur to the thread state or stacks between the sampling of one thread and the next” [38]. In order to collect all stacks simultaneously, each thread must halt its execution at a so called *safepoint*. The *HotSpot Glossary of Terms* defines a safepoint as following:

A point during program execution at which all GC roots are known and all heap object contents are consistent. From a global point of view, all threads must block at a safepoint before the GC can run. [...] From a local point of view, a safepoint is a distinguished point in a block of code where the executing thread may block for the GC. ([25])

Safepoints are located at different positions throughout the code. For interpreted code, the JVM may halt between any two bytecodes. For compiled code, the JIT compiler usually inserts safepoint checks at the return of a method and at loop back jumps [42]. However, this is not guaranteed, and the compiler may omit these checks in favor of performance.

This leads to two problems. The first problem is the performance overhead. All threads need to stop, and can only resume when every thread has reported its stack. When an application has lots of threads with deep stacks (as in most modern web frameworks), the stack collection may take a long time and worsen the runtime of the program. The second problem is that only safepoints are sampled. Think of a method that performs lots of computational work. In order to speed up the computation, the compiler may remove all safepoint checks inside this method. The thread then only halts at the next safepoint, outside of the method. Although the method performs a lot of computational work, it will never be sampled.

There exist sampling profiler for the JVM, which claim to be unaffected by the safepoint bias, for example, *honest-profiler*<sup>1</sup> or *async-profiler*<sup>2</sup>. They depend on the HotSpot-specific internal API `AsyncGetCallTrace`, which only samples a single thread and is independent of safepoints. The Java Flight Recorder is another alternative, which will be described in the next subsection.

### 3.1.3 The Java Flight Recorder

The Java Flight Recorder (JFR) is a feature originally from JRockit JVM for lightweight performance analysis of a running application. Afterwards, it became a commercial feature of Oracles Java Development Kit (JDK) 8, which describes it as:

Java Flight Recorder (JFR) is a tool for collecting diagnostic and profiling data about a running Java application. It is integrated into the Java Virtual Machine (JVM) and causes almost no performance overhead, so it can be used even in heavily loaded production environments. ([36])

---

<sup>1</sup><https://github.com/jvm-profiling-tools/honest-profiler>

<sup>2</sup><https://github.com/jvm-profiling-tools/async-profiler>

In JDK 11, the JFR was open sourced, with the goal to have at most 1% performance overhead [22]. Later it was even backported to OpenJDK 8 [50], in order to be used with Java 8 without the use of the commercial Oracle JVM.

The JFR collects events throughout the application execution. These events can occur in the JVM or the Java application itself, e.g., thread sleep, monitor wait or I/O operations. The JFR also includes a periodic event for execution sampling, which provides the current stack trace of random threads. In addition, the JFR avoids the problem of safepoint bias by using OS provided facilities to freeze threads instead of the JVM internal approach. Once a thread is frozen, the address of the next instruction to be executed is retrieved from memory. This address is then converted back to the source code line number, by utilizing the byte code to machine code symbol map generated by the JIT compiler. This is done for every frame in order to resolve the complete stack trace. In order to save time, the default setting limits the maximum resolved stack depth to 64 frames, however this can be changed by the user.

One problem with this approach is that the mapping of machine code to byte code is not always accurate, as the JIT compiler may reorder instructions. In order to overcome this issue, it is possible to add the JVM flags `-XX:+UnlockDiagnosticVMOptions` and `-XX:+DebugNonSafepoints` to instruct the JIT compiler to generate a more detailed symbol map, and to have a more accurate mapping of machine code to byte code [36, 41].

Another restriction of the JFR is that it only captures running threads. Threads sleeping or waiting are not sampled, thus only the execution time can be collected, not the actual wall clock time.

## 3.2 Collecting performance data

Section 3.1 described how the performance data can be gathered from a running program. Nevertheless, this performance data is still unprocessed and in the form of simple stack traces.

This section describes various ways how the provided stack traces may be aggregated into data structures, which allow further processing. The Subsections 3.2.1, 3.2.2 and 3.2.3 describe the most prominent data structures for aggregating performance data, and how each of them can be constructed from the sampling output.

### 3.2.1 Call Tree

The call tree (CT) is the most detailed data structure to represent call traces. In a call tree, every node represents a single method invocation with the caller as its parent.



Listing 3.1: Sample program

```

void a() {
    b();
    d();
    b();
    d();
}

void b() {
    c(true);
}

void c(boolean callE) {
    if (callE) {
        e();
        e();
    }
}

void d() {
    c(false);
}

void e() {
}

```

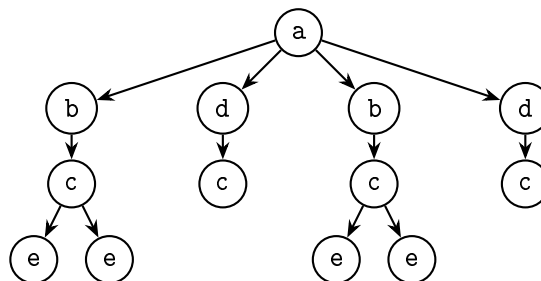


Figure 3.2: Call tree for a single invocation of method a from Listing 3.1

For instance, the sample program described in Listing 3.1 is considered, and its corresponding call tree for a single invocation of method a at Figure 3.2.

Execution times of methods can be added as payload to a node or the corresponding edge from the parent. A call tree is the most detailed representation of a program execution. However, it grows very fast and requires unbounded space, as each method invocation introduces a new node. An experimental setup showed that call trees regularly exceed several 100 millions of nodes, even for short executions around ten to twenty minutes [16]. Another problem refers to the output of sampling profilers. They provide no information, whether the same stack frame occurring in subsequent samples belongs to the same invocation, i.e., if a method is running for a long time or

is called often. Thus, it is not possible to create a CT from the output provided by a sampling profiler.

### 3.2.2 Call Graph

The call graph (CG) is a directed graph, where a node represents a method, and an edge represents a caller/callee relationship of two methods. In contrast to the CT, the call graph combines all invocations of a method into a single node. The corresponding metrics may be aggregated into the node (e.g., runtime) or the edge (e.g., call count). Therefore, a call graph is very space and time efficient, as the number of nodes is bounded by the number of methods. For example, the CGs of the previously mentioned experiment were usually in the single-digit thousands with the maximum size of about 30 000 nodes [16]. However, this also limits the information that can be extracted, as the profiling data is heavily aggregated [47, 40]. Call graphs were heavily used in the 80s by the *gprof* profiling tool [20].

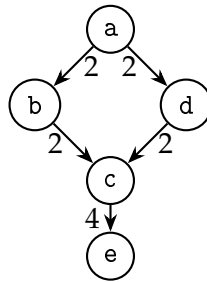


Figure 3.3: Call graph for invocation of method a from Listing 3.1

Figure 3.3 displays the call graph for the same invocation of method a as Figure 3.2. The edges contain the aggregated call count for each method. As it can be seen, the graph implies that each invocation of method c also invokes method e, which is not the case. Only invocations with the calling context of method b also lead to an invocation of e. For larger programs, the call graph is even less meaningful, as modern programs try to centralize the same functionality to a single unit in the source code, which is then called from different locations.

### 3.2.3 Calling Context Tree

The calling context tree (CCT) attempts to combine the accuracy of CTs, while still maintaining a feasible space limitation similar to CGs. Every node of the calling context tree represents a method, and the path to the root represents the calling context. Multiple invocations of a method within the same calling context are merged into one node, where metrics are aggregated. Methods with different contexts will appear multiple times in the tree.

The depth of the tree is bounded by the maximum possible stack depth of the profiled program. The width of each subtree is bounded by the method count. Thus, the overall space of the calling context tree is bounded by  $|methods|^{maximum\ stack\ depth}$ , which is still quite large. However, in reality calling context trees tend to be much smaller, because methods usually have a fixed and relatively small (in contrast to the total method count) number of callees. In the already mentioned experiment, the calling context tree node count was in the one to two-digit million range [16].

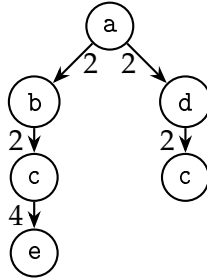


Figure 3.4: Calling context tree for invocation of method a from Listing 3.1

Figure 3.4 represents the calling context tree for the same invocation of method a as Figure 3.2. The edges contain the aggregated call count for each method. In contrast to the CG, it is now possible to identify that calls to method e only happen when c was previously called from b.

The calling context tree can be constructed and updated on the fly during the execution of a program. Each sample output is merged into the tree by traversing the stack (starting from the lowest frame, usually the main method) simultaneously with the tree (starting from the root). For each method on the stack, the corresponding child of the current node in the tree is traversed. If there exists no child for the method, a new node is created. When the top of the stack is reached, the sample count for the respective node is increased by one [3]. There exist other approaches, about how much the sample count should be increased. For example, Lee proposes an algorithm to recalculate the sample count based on the call density and sampling latency. This can correct a maybe present sampling bias [27].

### Merging Partial Stack Samples

As already mentioned, the JFR limits the stack depth at sampling events to avoid performance deterioration. Each of these incomplete samples is called *partial stack sample* (PSS). This leads to problems when constructing the CCT, as it is not possible to start from the root. To avoid this problem, the sampled stack depth can be increased. However, for large stacks this implies an unknown performance overhead, which is not acceptable in a production environment. Instead, it is possible to merge the incomplete call stack into an already present CCT by utilizing *maximal common merging* [46].

Listing 3.2: Maximal common merging algorithm

```
1  Given: CCT, PSS, matchThreshold
2
3  Node n = root of PSS
4  Set<Node> candidates = all nodes from CCT with same method as n
5  int depth = 1;
6  loop {
7      if (|candidates| == 0) {
8          // Zero candidates left, reperform merging when CCT has grown
9          return
10     } else if (|candidates| == 1 & depth > matchThreshold) {
11         mergeIntoSubtree(n, candidates[0])
12         return
13     } else if (n.child == null) {
14         // Multiple candidates (or mergeThreshold not reached) but n has no more children
15         // --> Ambiguity
16         return
17     }
18     n = n.child
19     depth += 1
20     Set<Node> childCandidates = {}
21     foreach (Node currentCandidate in candidates) {
22         Node childCandidate = currentCandidate.getChildForMethod(n.method)
23         if (childCandidate != null) {
24             childCandidates += childCandidate
25         }
26     }
27     candidates = childCandidates
28 }
```

The algorithm for *maximal common merging* is described in Listing 3.2. It requires an already present CCT, a PSS and the *match threshold* as input. The *match threshold* defines how long the matched sequence has to be at least, in order to allow the PSS to be merged. A lower value allows the algorithm to merge the PSS more aggressively but may introduce more errors.

The algorithm keeps a set of *candidates*, representing the nodes in the CCT, that may be used as root to merge the remaining PSS subsequence. At the beginning, the candidates set is initialized with all nodes from the CCT, which share the same method as the root node of the PSS. Now the PSS is iterated downwards. On every node, the candidate set is updated to contain all child nodes from the current candidates that share the same method as the PSS node. This can be seen as some kind of parallel traversal of all possible sequences in the CCT.

The iteration is stopped by one of three possibilities:

1. The candidate set has no entries left. Here, the merging cannot be performed as there exists no clear subtree. Here, the PSS may be merged at a later point in time when the CCT has more nodes.

2. The candidate set has exactly one entry, and the matched depth is larger than the match threshold. Now, the single candidate represents the root node of the subtree, that the remaining subsequence of the PSS can be merged into.
3. The sequence ended (i.e., the current PSS node has no child). The merging cannot be performed, as there are either too many candidates, or the match threshold is not yet reached. This is called an *ambiguity*, as there exists no unique subtree corresponding to the PSS. In this case, the PSS can never be merged into the CCT, as the CCT can only grow and thus the ambiguity will never be resolved. However, during the evaluation phase (see Chapter 5) an ambiguity occurred only four times.

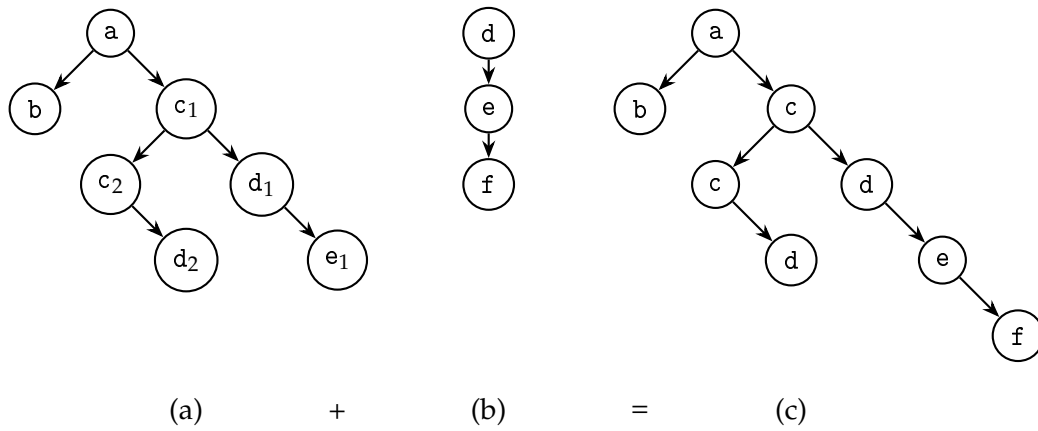


Figure 3.5: *Maximal common merging example*

The PSS (b) can be merged into the CCT (a) resulting in (c)

For an example, consider the CCT (a) and PSS (b) displayed in Figure 3.5. For simplicity the match threshold is zero. Initially the candidates set contains the nodes  $d_1$  and  $d_2$ . Then the next node ( $e$ ) from the PSS is considered. Only  $d_1$  has a child with the same method, so the candidates are reduced to only  $e_1$ . Now the iteration can stop, as a single candidate has been identified, and the subtree of  $e$  can be merged into  $e_1$ , resulting in the CCT (c).

As an example for an ambiguity, the same CCT is considered but now the PSS consists of only the nodes  $c \rightarrow d$ . After traversing the whole PSS, there exist multiple candidates  $d_1$  and  $d_2$ . There exists an ambiguity and the PSS will never be merged into the CCT.

### 3.3 Evaluating the CCT

Section 3.2 described how the gathered performance data can be collected into a data structure. This section describes how the data structure can be evaluated in order to

compute various metrics (Subsection 3.3.2). In addition, the metrics calculation must be aware of recursion (Subsection 3.3.1) and can be enriched with a novel approach, the task based calculation (Subsection 3.3.3).

### 3.3.1 Recursion detection

Before any metric for a method can be calculated, it is important to identify recursive methods. Recursive methods need special handling, as otherwise the same sample may be counted multiple times. This is not limited to methods that are directly recursive (i.e., the direct parent of a node has the same method in the CCT) and accounts for every node where the same method is encountered somewhere in its calling context.

Listing 3.3: Recursion detection algorithm for a single node  $n$

```
1 Given: Node n
2
3 Node current = n.parent
4 while (current != null) {
5     if (current.method == n.method) {
6         // Recursion detected
7         n.recursiveParent = current
8         return
9     }
10    current = current.parent
11 }
12 // Reached root -> n is not recursive
```

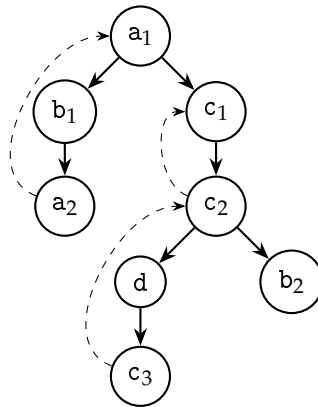


Figure 3.6: Example for a CCT enriched with recursion information

The recursive parents are referenced via dashed lines

The algorithm for detecting such a recursive parent for a node  $n$  in a CCT is described in Listing 3.3. It follows the calling context to the root. On every node it is checked if the current node has the same method as  $n$ . If the node has the same method as  $n$ , it

is considered recursive and the recursive parent for  $n$  is stored.

For an example the CCT displayed in Figure 3.6 is considered with the recursive nodes  $a_2$ ,  $c_2$  and  $c_3$ .

### 3.3.2 Metrics calculation

The metrics, which can be calculated for a method, are already described in Section 2.1. This section describes how they can be calculated using a sampling based CCT.

For this, a CCT is expected where every node  $n$  stores:

- its own sample count ( $n.sample$ ),
- the respective method ( $n.method$ ),
- all its child edges ( $n.childEdges$ ),
- its parent node ( $n.parent$ ),
- and its recursive parent node ( $n.recursiveParent$ ).

Every child edge  $e \in n.childEdges$  stores:

- its target node ( $e.target$ ),
- and the total number of sample counts ( $e.sample$ ) of the subtree denoted by the edge target.

Subsequently, the subtree sample count  $sample_{subtree}$  of a node  $n$  can be calculated by:

$$sample_{subtree}(n) = \sum_{e \in n.childEdges} e.sample \quad (3.1)$$

The total number of sample counts  $sample_{total}$  for a node  $n$  can be calculated by:

$$sample_{total}(n) = n.sample + sample_{subtree}(n) \quad (3.2)$$

In order to calculate the metrics for a method  $m$ , the corresponding set of nodes  $\mathcal{N}$  inside the CCT has to be calculated:

$$\mathcal{N}(m) = \{n \mid n \in CCT \wedge n.method = m\} \quad (3.3)$$

From this, the set of recursion free nodes  $\mathcal{R}_{free}$  can be calculated using:

$$\mathcal{R}_{free}(\mathcal{N}) = \{n \mid n \in \mathcal{N} \wedge n.recursiveParent \notin \mathcal{N}\} \quad (3.4)$$

The method samples  $sample_{method}$  can then be calculated by summing the total sample count for every recursion free node:

$$sample_{method}(m) = \sum_{n \in \mathcal{R}_{free}(\mathcal{N}(m))} sample_{total}(n) \quad (3.5)$$

The *method time*  $t_{method}$  of a method  $m$  with respect to the root node  $r$ , is calculated by dividing the method samples of  $m$  by the total sample count of the root  $r$ :

$$t_{method}(m, r) = \frac{sample_{method}(m)}{sample_{total}(r)} \quad (3.6)$$

The result is a fraction representing the proportion of time, that was spent executing method  $m$ .

The *self time*  $t_{self}$  of a method  $m$  is calculated by dividing the self sample count of all nodes for  $m$  by the method samples of  $m$ :

$$t_{self}(m) = \frac{\sum_{n \in \mathcal{N}(m)} n.sample}{sample_{method}(m)} \quad (3.7)$$

The result is a fraction representing the proportion of *method time* that the method spent executing its own statements.

Another metric to be calculated is the *callee time*. The *callee time* is calculated for every method that has at least one node as child of any node from  $\mathcal{N}(m)$ , i.e., the callee time is calculated for every method called from  $m$ . For this, all child nodes  $\mathcal{C}$  of  $m$  must be collected:

$$\mathcal{C}(m) = \{e.target \mid n \in \mathcal{N}(m) : e \in n.childEdges\} \quad (3.8)$$

Then, all child methods  $\mathcal{C}_{method}$  of  $m$  can be computed:

$$\mathcal{C}_{method}(m) = \{n.method \mid n \in \mathcal{C}(m)\} \quad (3.9)$$

These methods are the callees of  $m$ . For each callee method  $c \in \mathcal{C}_{method}(m)$  the respective nodes  $\mathcal{C}_{nodes}$  can be computed:

$$\mathcal{C}_{nodes}(m, c) = \{n \mid n \in \mathcal{C}(m) \wedge n.method = c\} \quad (3.10)$$

In order to compute the *callee time*  $t_{callee}$ , all the samples for the invocation of  $c$  from  $m$  have to be computed:

$$sample_{inv}(m, c) = \sum_{n \in \mathcal{R}_{free}(\mathcal{C}_{nodes}(m, c))} sample_{total}(n) \quad (3.11)$$

In order to avoid duplicate counting of samples, only the recursion free set of nodes has to be considered. Finally, the *callee time*  $t_{callee}$  can be computed by:

$$t_{callee}(m, c) = \frac{sample_{inv}(m, c)}{sample_{method}(m)} \quad (3.12)$$

The result is a fraction representing the proportion of *method time* that the method  $m$  spent executing method  $c$ .

The next metric to be calculated is the *caller time*. The *caller time* is calculated for every method that has at least one node as parent of any node from  $\mathcal{N}(m)$ , i.e., the



caller time is calculated for every method calling  $m$ . For this, all parent nodes  $\mathcal{P}$  of  $m$  must be collected:

$$\mathcal{P}(m) = \{n.\text{parent} \mid n \in \mathcal{N}(m)\} \quad (3.13)$$

From these nodes, the caller methods  $\mathcal{P}_{\text{method}}$  can be calculated:

$$\mathcal{P}_{\text{method}}(m) = \{n.\text{method} \mid n \in \mathcal{P}(m)\} \quad (3.14)$$

These methods are the callers of  $m$ . For each parent method  $p \in \mathcal{P}_{\text{method}}(m)$  the *caller time*  $t_{\text{caller}}$  can be calculated by:

$$t_{\text{caller}}(m, p) = \frac{\text{sample}_{\text{inv}}(p, m)}{\text{sample}_{\text{total}}(m)} \quad (3.15)$$

The result is a fraction representing the proportion of *method time* that the method  $m$  spent while being executed from method  $p$ .

So far, it has been assumed that the samples for each node are aggregated into one value, independent of the sampled thread. Instead, the samples can be stored per thread and aggregated on demand. For this the property "sample<sub>thread</sub>" is introduced for each node and edge. It is defined as dictionary  $\mathcal{D} \subseteq K \times V$  where  $\forall(k_1, v_1) \in \mathcal{D}, \forall(k_2, v_2) \in \mathcal{D} : k_1 = k_2 \rightarrow v_1 = v_2$  over the set of all possible threads  $K$  (as keys) and the set of all possible sample counts  $V = \mathbb{N}$  (as values). The respective element of such a tuple  $p = (k, v)$  can be accessed via  $\text{key}(p) = k$  and  $\text{value}(p) = v$ . The original property "sample" is then replaced by the aggregation over all thread samples:

$$\text{sample} = \sum_{p \in \text{sample}_{\text{thread}}} \text{value}(p) \quad (3.16)$$

In order to access the samples of a single thread  $t$ , the notation  $\text{sample}_{\text{thread}}[t]$  is introduced:

$$\text{sample}_{\text{thread}}[t] = \begin{cases} \text{value}(p), & \text{if } p \in \text{sample}_{\text{thread}} \wedge \text{key}(p) = t \\ 0, & \text{otherwise} \end{cases} \quad (3.17)$$

The functions  $\text{sample}_{\text{subtree}}$  (Equation 3.1),  $\text{sample}_{\text{total}}$  (Equation 3.2) and  $\text{sample}_{\text{method}}$  (Equation 3.5) can be duplicated with a thread parameter  $t$  to access only the thread samples for  $t$ :

$$\text{sample}_{\text{subtree}}(n, t) = \sum_{e \in n.\text{childEdges}} e.\text{sample}_{\text{thread}}[t] \quad (3.18)$$

$$\text{sample}_{\text{total}}(n, t) = n.\text{sample}_{\text{thread}}[t] + \text{sample}_{\text{subtree}}(n, t) \quad (3.19)$$

$$\text{sample}_{\text{method}}(m, t) = \sum_{n \in \mathcal{R}_{\text{free}}(\mathcal{N}(m))} \text{sample}_{\text{total}}(n, t) \quad (3.20)$$

This allows the computation of the last metric, the *thread time*. For this, the set of threads  $\mathcal{T}$  the method  $m$  was sampled for needs to be calculated:

$$\begin{aligned} \mathcal{T}(m) = & \{ \text{key}(p) \mid n \in \mathcal{N}(m) : p \in n.\text{sample}_{\text{thread}} \} \cup \\ & \{ \text{key}(p) \mid n \in \mathcal{N}(m) : e \in n.\text{childEdges} : p \in e.\text{sample}_{\text{thread}} \} \end{aligned} \quad (3.21)$$

Now the *thread time*  $t_{thread}$  can be calculated for every thread  $t \in \mathcal{T}(m)$ :

$$t_{thread}(m, t) = \frac{sample_{method}(m, t)}{sample_{method}(m)} \quad (3.22)$$

The result is a fraction representing the proportion of *method time* that the method  $m$  spent being executed on thread  $t$ .

### 3.3.3 Task based evaluation

The previously introduced metrics are calculated from the complete CCT. This is useful when the execution of the complete system is considered. Sometimes this cannot be useful, especially for large systems with different modules, which are working mostly independent of another.

Listing 3.4: Task definition with a specification by prefix

```
1 TaskDefiniton: {  
2   name: "UserRegisterTask",  
3   specification: {  
4     type: "prefix",  
5     value: "com.mycompany.service.UserRegister",  
6     task-name: "user-register"  
7   }  
8 }
```

As an example, a web service providing some sort of online functionality is considered. The service can be used anonymously, but also allows users to create an account to unlock more features. Most of the execution time is spent while providing the core functionality and only a fraction ( $< 1\%$ ) is required for the user registration process. After some time, users report that the registration process takes a lot of time. When starting to search for the problem, a developer would identify the email validation requiring most of the execution time when a new user is registered. However, the email validation is also used from other parts of the system, which are not reported to be slow, so the developer cannot distinguish which part of the validation is the root cause for the slow user registration process. Only a subtree of the global CCT, which includes the user registration service, would need to be evaluated.

Listing 3.5: Task definition with a specification by regular expression

```
1 TaskDefiniton: {  
2   name: "ServiceTask",  
3   specification: {  
4     type: "regex",  
5     value: "^com.mycompany.service.(?<service-name>[^\.]*)",  
6     task-name: "service-${service-name}"  
7   }  
8 }
```

For this problem *tasks* were introduced. Tasks can be interpreted as a label for CCT nodes, identifying subtrees, which can be used for metrics calculation independent of the complete CCT. Each child node inherits all tasks from its parent and may also reference other tasks. A task always stems out of a task definition, which includes the specification for when a node is matched and how the resulting task is called. The specification can be arbitrary, for example, see Listing 3.4, which contains a specification implemented as prefix match on the complete package (`com.mycompany.service.UserRegister`), resulting in a task with a static name (`user-register`). Another possibility would be a more generic definition, for example by regular expression (Listing 3.5). This example may yield different tasks based on the matched value in the named capturing group "service-name". Now a task definition can be created, which will match the user registration service as its own task, and only display the metrics for this task. This allows the developer to easily identify the slow subroutine in the email validation and to provide a resolution.

In order to correctly reflect tasks in the metrics, the calculation has to be made task-aware. For this, the user specified global set of currently selected tasks  $\mathcal{T}$  is considered. The first thing to be changed is the set of nodes  $\mathcal{N}(m)$  corresponding to a method  $m$  (Equation 3.3). It is replaced with the task aware set of nodes  $\mathcal{N}_t$ :

$$\mathcal{N}_t(m) = \{n \mid n \in \mathcal{N}(m) \wedge n.\text{tasks} \cap \mathcal{T} \neq \emptyset\} \quad (3.23)$$

This already leads to a correct calculation of  $t_{self}$ ,  $t_{callee}$ ,  $t_{caller}$  and  $t_{thread}$ . Only the method time  $t_{method}$  (Equation 3.6) needs another change, as the global root  $r$  is not the desired reference point anymore. Instead, special task roots  $\mathcal{R}_t$  have to be used as reference:

$$\mathcal{R}_t = \{n \mid n \in CCT \wedge n.\text{tasks} \cap \mathcal{T} \neq \emptyset \wedge n.\text{parent.tasks} \cap \mathcal{T} = \emptyset\} \quad (3.24)$$

As children inherit the tasks of their parent nodes, this will always select the highest nodes, without selecting any other node from their subtree. Finally, the task-aware  $t_{method_t}$  can be calculated:

$$t_{method_t}(m) = \frac{sample_{method}(m)}{\sum_{r \in \mathcal{R}_t} sample_{total}(r)} \quad (3.25)$$

### 3.4 Display performance results

After the profiling data was gathered (Section 3.1), collected (Section 3.2) and evaluated (Section 3.3), the results have to be displayed in order to utilize them. Subsection 3.4.1 focuses on how the calculated metrics can be included into the developers IDE. Subsection 3.4.2 lays out how code changes are handled, and finally Subsection 3.4.3 introduces an approach to compare the performance metrics to those of a previous version.

### 3.4.1 In situ visualization

Multiple ways exist how the gathered and analyzed profiling information can be made accessible for the developer. Various external representations were already proposed [8, 10, 32]. However, they all suffer from the *split attention effect*, which hinders information processing by using multiple sources. These require the developer to split their attention in order to completely and correctly understand the displayed information [5]. This can increase the cognitive load of the developer and thus slow down the development process [39].

In order to avoid this, the profiling information should be as close to the actual code as possible. Nowadays, software development is mostly done by utilizing an IDE, so the profiling information was integrated using an *in situ* visualization [23]. Beck et al. already implemented such an in situ visualization for the Eclipse IDE [7].

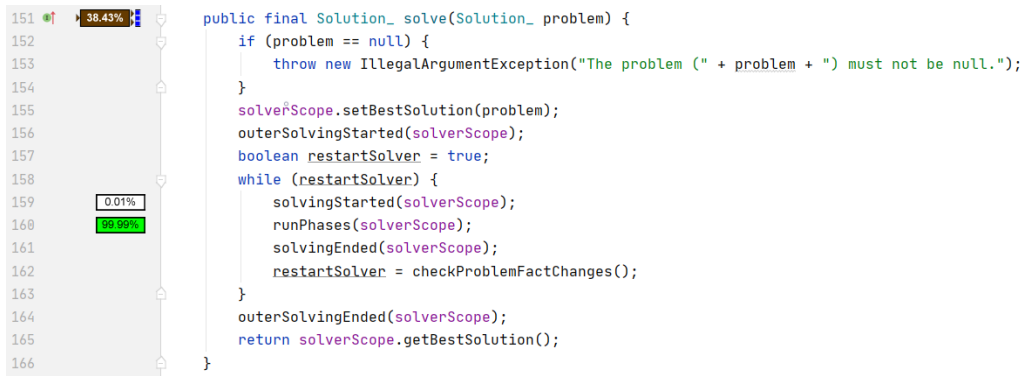


Figure 3.7: In situ visualization example

This in situ integration was re-implemented for the JetBrains IntelliJ IDEA<sup>3</sup>. It creates a small colored box for each method, which summarizes the computed metrics. Due to implementation reasons, it is not possible to display this box directly in the code (as done in the original approach), but instead it is included in the line gutter (see Figure 3.7). The box contains the method time  $t_{method}$  as percentage value. In addition, the time is mapped to a background color, using a default scale from bright green to dark red. The user may change the color mapping according to his own needs. The self time  $t_{self}$  is encoded as striped bar growing from left to right, where the value is represented as proportionally width of the bar.

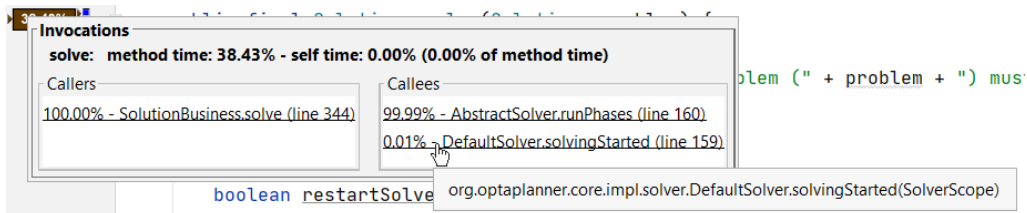


Figure 3.8: Invocation view popup example

<sup>3</sup><https://www.jetbrains.com/idea/>

A preview for the number of callers/callees is provided by small arrows on the left (callers) and right (callees) of the box. For each call an arrow is displayed, up to a maximum arrow count of three. In addition, a detailed list of callers/callees (invocation view) can be accessed by opening a popup with a left-click on the box (Figure 3.8). Similar to the arrow preview, callers are listed on the left and callees on the right. Each list element is preceded by the respective invocation time ( $t_{caller}/t_{callee}$ ) and contains the class, method name, and line number of the invocation. The complete signature of the method can be accessed via tooltip. In addition, the invocation targets were looked up in the IDE internal Program Structure Interface (PSI) in order to allow a direct jump to the respective code location via mouse click, indicated by an underlined text. Each list is sorted by the respective invocation time in descending order.

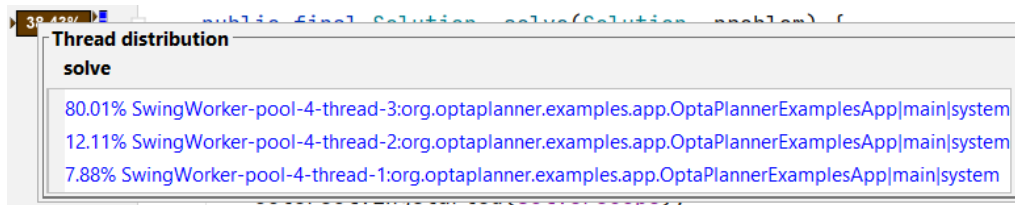


Figure 3.9: Thread view popup example

Table 3.1: Examples of in situ visualizations for methods

Icon	$t_{method}$	$t_{self}$	#callers	#callees	#threads	#thread groups
	0.024	0.0145	1	$\geq 3$	2	1
	0.051	1	2	0	4	2
	0.1248	0.7941	$\geq 3$	2	3	1
	0.3074	0.2212	$\geq 3$	1	$\geq 6$	4
	1	0	0	$\geq 3$	1	1

On the right side of the box another small diagram is displayed. It contains the threads from which the method was executed, where each thread is represented by a small square. The number of displayed threads in this diagram is limited to six. The coloring allows a differentiation between different thread groups. In the example (Figure 3.7) the method was executed by three threads from the same thread group, Table 3.1 lists more possibilities. Similar to the invocation view, a thread view popup can be opened with a right-click on the box (Figure 3.9). The popup contains the list of all threads, where each list element is preceded by the respective thread time ( $t_{thread}$ ) followed by the name and groups of the thread. The list is sorted by the respective thread time ( $t_{thread}$ ) in descending order.

While the popups provide the developer with every available information, they still have to be explicitly accessed. However, it is possible to display the callees time directly in the code. Another box is inserted for every callee at the respective code line. The box is designed similar to the one providing the method time. It



contains the callee time ( $t_{callee}$ ) as percentage value. In addition, the background is filled with a colored bar to graphically represent the value. The default color is bright green for all percentage values. Once more, the user may change the color (for the respective percentage) according to his own needs. In case multiple invocations happen at the same line, they are combined into a single box. The tooltip provides more detailed information about the invocation target and respective callee time. As the implementation uses a sampling based approach, it is possible that there does not exist information for every invocation to create this box.

### 3.4.2 Integrating code changes

The gathered performance data represents the profiled program at a single development state, namely the profiled release. However, development may continue, and different parts of the program can change. This leads to a problem, as small changes (in terms of changed text) may already have a huge impact on how the program operates. In order to avoid discrepancies between the displayed in situ visualizations and the actual code, multiple possibilities arise:

First, it is possible to hide all in situ visualizations, as soon as the code changes. This may be applicable to programs, that change very rarely. However, this only aids the first developer that changes the code. Any changes afterwards must be done without the aid of the in situ visualizations, which essentially makes them superfluous.

Second, it is possible to detect the code changes and try to infer the performance of the newly written code. This is a complex operation, which requires more in depth understanding of the analyzed code, and may lead to all kind of errors.

Third, it is possible to show a visual indicator, that the code changed, and allow the developer to make the decision, whether the provided performance data is still relevant or not. For this, a simple variation of the method visualization is sufficient. The saturation of the box is reduced to 75% and it gains a transparency of 25%, no other behaviour is changed. For an example, consider the original image: , which is then changed to: .

In addition, all invocation visualizations inside the changed method are removed as it is not possible anymore to place them with enough confidence.

The previously mentioned visualization change is only performed for already committed changes. Local not yet committed changes do not alter the visualization, as the IDE usually already provides its own visual indicators for them, and it is expected that the developer is aware of their own changes.

### 3.4.3 Performance comparison with baseline

Beck et al. proposed a visualization for comparing two different program versions by their respective performance data [7]. This may be useful when performance changes need to be detected. Reasons for that can be, for example, 1) a part of the program became slower in a new version, or 2) because performance improvements need to be validated. For this, the user has to select a version of the program he wants to use as

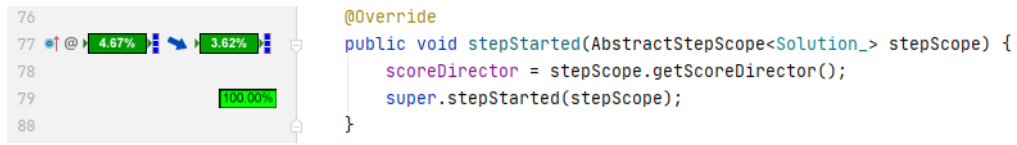


Figure 3.10: Example for a performance comparison to a baseline

baseline. The evaluation (Section 3.3) is then performed for the current version and the baseline version. The performance data for these two versions is then displayed next to each other (Figure 3.10). Each box has the full feature support as described in the previous sections.

Between the boxes an arrow is inserted, which gives a visual indication whether the performance improved, worsened or was (roughly) the same by utilizing the angle and color. The angle ranges from  $90^\circ$  (arrow pointing downwards) denoting a huge performance improvement, to  $-90^\circ$  (arrow pointing upwards) denoting a huge performance regression. The color ranges from green (performance improvement) over blue (no significant change) to red (performance regression).

Table 3.2: Examples for performance change arrow, where the baseline *method time* is always 0.1

Arrow	current <i>method time</i>	angle	red	green	blue
	0.05	90	0	255	0
	0.067	44	0	125	130
	0.083	18	0	51	204
	0.1	0	0	0	255
	0.133	-29	82	0	173
	0.167	-60	170	0	85
	0.2	-90	255	0	0

Listing 3.6: Performance change angle computation

```
1 Given: currentMethodTime, baselineMethodTime
2
3 if (baselineMethodTime == 0) {
4     if (currentMethodTime == 0) {
5         // both zero -> no change
6         return 0
7     }
8     // zero baseline, any current -> performance "regression"
9     return -45
10 } else if (currentMethodTime == 0) {
11     // zero current, any baseline -> performance "improvement"
12     return 45
13 }
14
15 // calculate relative change
16 dMethodTime = currentMethodTime / baselineMethodTime;
17
18 // limit to the interval [0.5;2]
19 dMethodTime = max(0.5, min(2, dMethodTime))
20
21 // normalize the interval to [-1;1] where [0.5;1[ -> ]0;1] and [1;2] -> [-1;0]
22 if (dMethodTime >= 0) {
23     dMethodTime = -1 * (dMethodTime - 1)
24 } else {
25     dMethodTime = (1 / dMethodTime) - 1;
26 }
27 return 90 * dMethodTime
```

Listing 3.6 describes the algorithm to compute the angle. At first, special cases are handled, where no method time was detected for the baseline and/or the current value. Afterwards, the relative change of the *method time* is computed by dividing the current value by the baseline value. This change is then limited to the interval  $[0.5, 2]$ , so the maximum detected change is a halving or doubling of the method time. This interval is then normalized to  $[-1, 1]$ , where the range  $[0.5, 1[$  is mapped to  $]0, 1]$  and  $[1, 2]$  is mapped to  $[-1, 0]$ . The resulting value can then be multiplied with the maximum angle of  $90^\circ$ .

The color can then be calculated by utilizing the already calculated angle. For this, the absolute value of the angle is mapped to the interval  $[0, 255]$ . This value is then used as red (angle  $< 0$ ) or green (angle  $\geq 0$ ) color value and is subtracted from the initial blue value (255). Table 3.2 shows some example arrows with the respective angle and color values.



## 4 Implementation

The following chapter describes the general implementation principles. Section 4.1 highlights the general architecture of the developed components. A more in-depth view about how the profiler is integrated into the running process is given in Section 4.2. Section 4.3 focuses on how the gathered profiling data is integrated and analyzed in Teamscale. Finally, the chapter is terminated by Section 4.4, which describes the integration into the developers IDE.

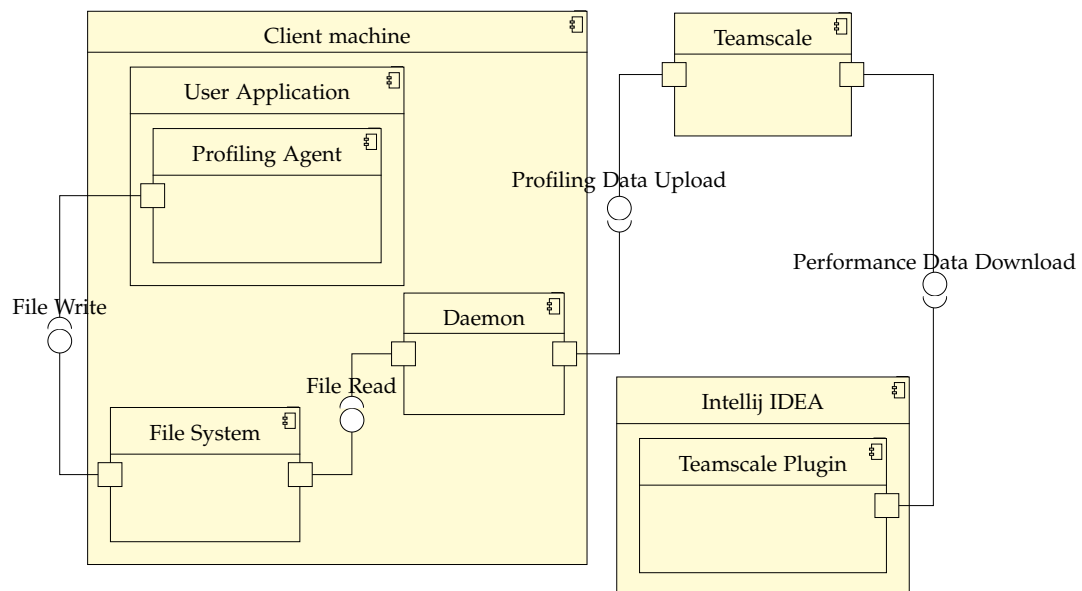


Figure 4.1: Component diagram illustrating the general solution architecture and its communication

### 4.1 Architecture

The developed solution consists of the following components:

- *Profiling Agent*: The profiling agent is responsible for the correct configuration and lifecycle of the JFR. It is implemented as Java Agent and started together with the profiled application utilizing the JVM flag `-javaagent`. As it runs in the same JVM as the profiled application, the agent must require almost no resources in order to affect the profiled application as little as possible.
- *Daemon*: The daemon is responsible for collecting the profiling output from the JFR and sending it to the Teamscale server. It runs on the same machine as the profiled application (and profiling agent) but in a different JVM process.
- *Teamscale*: The Teamscale instance is responsible for analyzing the uploaded profiling data. It usually runs in a cloud hosted environment.
- *IDE Plugin*: The IDE plugin is responsible for the communication with the Teamscale instance and for the display of the analyzed performance data. It runs inside the IntelliJ IDEA on the machine of the developer.

The interaction of these components is illustrated in Figure 4.1.

### 4.2 Process Integration

As it was already mentioned in Subsection 3.1.3, the JFR was selected as profiler. It is possible to start the JFR directly with the application by utilizing the JVM flag `-XX:StartFlightRecording`. However, it is not possible to get continuous profiling data by using this method since the final recording is only written when the application exits. This is not feasible for long-running applications.

Because of this issue the agent was introduced. Upon program start the agent creates a new JFR recording which only holds the event data in memory for a limited amount of time. The agent spawns a new thread, which periodically clones the initial recording and dumps the cloned recording data to disk. As it is not guaranteed that the cloned recording only contains events since the last dump, an artificial dump event is placed before each dump, containing the consecutive id of the dump. This later allows the daemon to filter out any previous events, that were already contained in the last dump, and any subsequent events, that will be considered in the next dump.

The data has to be written to the file system first because it is not possible to stream it to the daemon process. This was only introduced in the later JDK 14 via JFR event streaming [18].

Another change to the default JFR usage is the use of a custom configuration profile. The default JFR installation contains two predefined profiles. The `default.jfc` profile “[c]ollects a predefined set of information with low overhead, so it has minimal

impact on performance and can be used with recordings that run continuously” ([37]), and the `profile.jfc` profile “[p]rovides more data than the `default.jfc` profile, but with more overhead and impact on performance” ([37]). Both profiles have over 100 events enabled for recording. However, only one event is actually required, the `jdk.ExecutionSample` (and the always activated artificial dump event). The sampling interval for the `default.jfc` is set to 20 ms and for the `profile.jfc` to 10 ms. In order to record only the wanted sampling event, a new profile was introduced which only enables the `jdk.ExecutionSample` event. Due to this reduced recording behaviour, it is possible to increase the sampling interval to 10 ms while still keeping a low overhead (see Section 5.2).

## 4.3 Teamscale Integration

The following section describes how the profiling data is integrated and analyzed in Teamscale. For this, the profiling data has to be uploaded as described in Subsection 4.3.1. Subsection 4.3.2 summarises how the CCT is represented in Teamscale. Subsection 4.3.3 provides an overview of the various analysis steps. Subsection 4.3.4 describes how the source-code is mapped to CCT nodes. Finally, Subsection 4.3.5 illustrates how changes in the source code are handled.

### 4.3.1 Daemon Upload

In order to make the profiling data available for Teamscale, it has to be uploaded to the respective instance. This could be done by the profiling agent (instead of dumping the events to disk), but a new process was chosen for multiple reasons:

- The performance impact on the profiled application should be as small as possible
- The daemon can be started/stopped independently of the profiled application, allowing custom usage patterns
- The upload may require special network configuration (e.g., proxy) that is different from the network configuration of the profiled application
- The upload of every sample event is not feasible because of the high frequency and amount of data

The daemon collects dumped execution samples from disk and merges them into an in-memory CCT. In case the sample is truncated, i.e., when the stack is deeper than the maximum configured JFR depth, it is merged into a separate CCT with an artificial root. These CCTs are uploaded in regular intervals (e.g., 1 hour) to the configured Teamscale instance. After they were uploaded, the CCTs are cleared so that every upload represents only the profiling data since the last upload.

### 4.3.2 CCT representation

Teamscale uses a key-value store with range and prefix query support which is basically an abstraction of a NoSQL database. In addition, the data is branched, i.e., the data is organized in individual commits that can have arbitrary commits as parents. A commit stores only the data that has changed compared to its parent. Other values are retrieved from previous commits.

In order to effectively represent a CCT in this kind of storage system, the following layout was chosen: Every node is stored as an own key-value entry. The key of a node is defined by:

1. the fully qualified name of the declaring class, e.g., `com.mycompany.MyClass`
2. the name of the method, e.g., `myMethod`
3. the method parameters, e.g., `(MyModelClass, MyOtherModelClass)`<sup>1</sup>
4. the line number from where the node was called in the parent method (see later)
5. the Execution Id (see Subsubsection 4.3.2)
6. the depth inside the CCT
7. a universally unique identifier (UUID)

The ordering of the key items and the support for prefix queries allow fast access for all nodes based on their package, class or even method. A UUID is created for every node in order to avoid collisions with other nodes that by coincidence have the same key but come from a different calling context.

The value is the node metadata. This includes the key of the parent node, the amount of self samples (per thread), and all child node keys with their respective subtree sample count (per thread).

Each CCT node is also distinguished by the line number from where it was called. This allows a more fine-grained evaluation as a method may contain multiple calls to another method at different locations, where each call has a different execution profile.

#### Execution Id

A single Teamscale instance is able to manage multiple projects. Each project manages source files, which may be retrieved from a code repository, and different configurations for this project. The CCTs are always project specific. However, a single project can manage multiple programs (e.g., backend server and client application) which are executed and thus also profiled independently of another. Because of this, the programs should also yield independent CCTs as they may affect each other otherwise.

---

<sup>1</sup>due to implementation reasons it is not possible to use the fully qualified name of the parameter classes, see Subsection 4.3.4

For example, the backend process may be running for a long time producing a huge amount of samples, whilst the client application is not running all the time producing way less samples. If these CCTs were mixed, the reported metrics of the client application will be much smaller than they are in reality.

In addition, each program may have multiple released versions. A program in combination with the respective release commit (usually a tagged commit) is called *executable*. Different executables should not be mixed in order to be able to identify changes and different runtime behaviour between them.

Each executable may also run on multiple machines, which can be under different loads. When the developer is evaluating the results, he may want to see only the performance of one or multiple specific machines. The combination of an executable and the respective runtime instance is called *Execution Id*. As all CCTs for a project are stored inside the same storage system, each node contains its respective Execution Id. Upon upload by the daemon, the respective Execution Id has to be provided. Upon retrieval of the results, the executable and the wanted instance(s) have to be provided.

### 4.3.3 Analysis Steps

The analysis engine in Teamscale is an event-based execution system triggered by changes in the storage system (similar to traditional database triggers). Different analysis steps write to different parts of the storage system (also called index), which again may trigger the execution of other analysis steps.

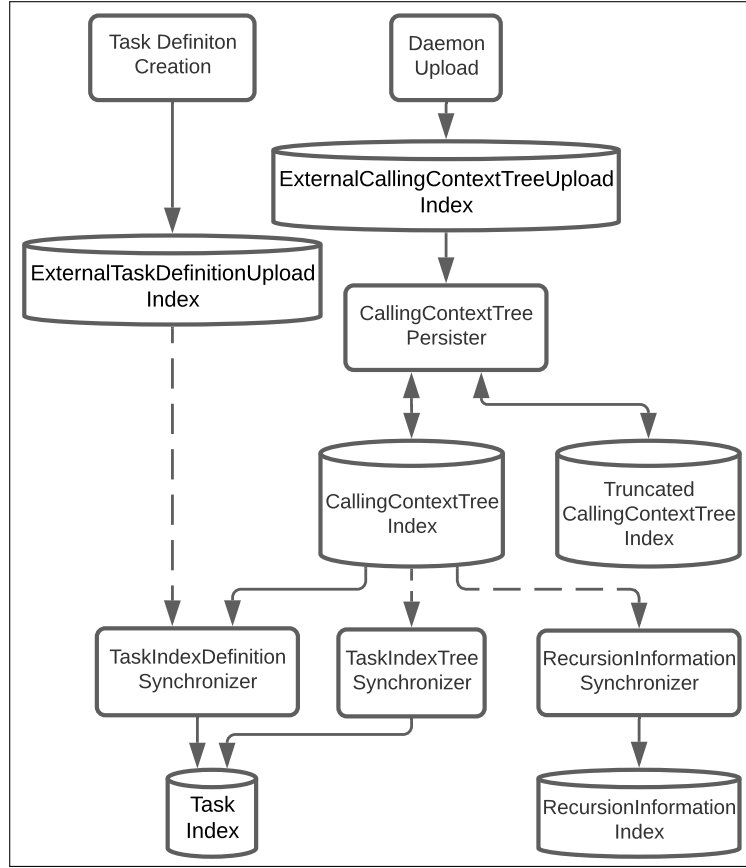


Figure 4.2: Flow chart illustrating analysis steps and their respective interactions  
Dotted lines indicate a trigger and data read, solid lines indicate a data read/write

Figure 4.2 illustrates the triggers and indices as used by the current implementation. Upon upload of a new CCT by the daemon, a new commit with the uploaded data is created and the uploaded CCT is stored in the special *ExternalCallingContextTreeUploadIndex*. This triggers the execution of the *CallingContextTreePersister*, which loads the CCT from the upload store and merges it into the internal *CallingContextTreeIndex* and *TruncatedCallingContextTreeIndex*. Afterwards, it tries to merge the truncated nodes from the *TruncatedCallingContextTreeIndex* into the *CallingContextTreeIndex*. After this, the *RecursionInformationSynchronizer* calculates the recursion information for every new node and stores it in the *RecursionInformationIndex*. Simultaneously, the *TaskIndexTreeSynchronizer* calculates the tasks for every new node and stores the result in the *TaskIndex*.

In case a new task definition is introduced by a user, it is stored (similar to the CCT upload) inside the *ExternalTaskDefinitionUploadIndex*. The *TaskIndexDefinitionSynchronizer* then evaluates the new task definition for the current CCT and stores the result in the *TaskIndex*.

During the preparation for the evaluation phase it became obvious, that the merging of the truncated nodes requires too much time for processing every node. Because of this, the runtime of the *CallingContextTreePersister* was limited to 50 minutes. Not yet

processed truncated nodes were kept in the *TruncatedCallingContextTreeIndex* to may be merged in a later iteration.

#### 4.3.4 Source code mapping

Until now, the CCT was evaluated independently of the actual source code. In order to be able to calculate the method metrics, the textual source code has to be matched to the runtime information available in the CCT nodes.

Teamscale already performs calculations based on the source code. This allows a lookup of all classes inside a specific file and a lookup of all methods within a specific class. However, the method information only contains the name and some internal attributes. In order to correctly map a source code method to the runtime method call, the fully qualified names of the method declaration and parameters have to be extracted.

When extracting the method information, Teamscale uses a self-implemented *shallow parser* to parse the textual source code into an abstract model. This model can be used to further identify the method parameters and declaration. The declaration can be easily resolved by prepending the fully qualified name of the declaring type to the method name.

Listing 4.1: Example class for parameter resolution

```
package com.mycompany;

import com.mycompany.mypackage.*;

class MyClass implements MyInterface<Model> {

    void method1(Model model, String anyString) { /* ... */ }

    <Model> void method2(Model model) { /* ... */ }

}
```

However, the parameters require a more complicated approach. Listing 4.1 provides an example class with some methods. The CCT node for method1 could possibly have this method identifier:

```
com.mycompany.MyClass.method1(com.mycompany.Model, java.lang.String).
```

However, when using the shallow parser, no resolution is performed. This makes it impossible to know what the runtime class name of *Model* is. It could possibly be *com.mycompany.Model* or *com.mycompany.mypackage.Model*. If generics are part of the class declaration another possibility arises: *method1* could be part of the interface *MyInterface*, where the first parameter type is the type of the generic variable. Due to type erasure, this would become

```
com.mycompany.MyClass.method1(java.lang.Object, java.lang.String) at runtime.
```

Only for *method2* it is possible to say that the parameter will be of type *java.lang.Object* at runtime as the *Model* type is shadowed by the *Model* type variable.

These problems required the change of two things. First, the CCT nodes do not contain the fully qualified class name for every parameter (as already mentioned in Subsection 4.3.2). Instead, the names are shortened to the last type name, e.g., `java.lang.Object` will be transformed to `Object`. Even for inner classes only the inner class name is used, e.g., `com.mycompany.MyClass$MySubclass` will be transformed to `MySubclass`. Second, the parameter resolution stores how likely the parameter type is a true type or a generic type variable, which will be erased during runtime.

When the CCT is searched for all nodes for a specific method, a prefix search based on the method declaration and parameters is performed. For each parameter it is checked, whether it is a true type, possibly generic, or definitely generic. For true type, the type is used and for definitely generic `Object` is used. When the type is possibly generic, one search for the found name and one for `Object` is performed. In case multiple parameters are possibly generic, a search is performed for every possible type/`Object` combination.

Listing 4.2: Example class for parameter resolution

```
class MyClass implements MyInterface<Model> {  
  
    // First case: overloaded method with different inner classes, which have the same name  
    // CCT nodes for these methods will be the same  
    void method1(SomeClass.Inner inner) { /* ... */ }  
    void method1(SomeOtherClass.Inner inner) { /* ... */ }  
  
    // Second case: overloaded method, where the possibly generic parameter is overloaded with type Object  
    // Yields different CCT nodes, but lookups for the first method  
    // will also include all nodes of the second method  
    void method2(Model model, String anyString) { /* ... */ }  
    void method2(Object mode, String anyString) { /* ... */ }  
}
```

This approach yields results that are not 100% accurate for some overloaded methods. The first change has the effect that the CCT contains the same nodes for methods with parameters, which are inner classes with the same name but different outer classes. The second change has the effect that for possibly generic methods with an overloaded method where one or multiple parameters are `Object`, the nodes of the overloading method are also included in the result nodes for the overloaded method. Examples are provided in Listing 4.2. As this is rarely the case, it is a reasonable limitation.

### 4.3.5 Source code changes

Due to the nature of Teamscale, it is already known when and how a specific method changed. As the developer is always interested in the performance metrics of their current state, the node lookup is performed based on the methods of the current development state. Of course, it is possible that methods got removed since the last profiling run. For them, performance metrics will not be shown anymore. It



is also possible that methods were added. For these, the performance will always be 0% as they were not called in the profiled application. For methods that got changed, Teamscale already provides information about when these methods were last changed. In case the change happened after the profiled state, the method is marked as changed in the performance result. This is important as the developer may draw wrong conclusions from the displayed profiling information otherwise. Small refactorings that do not change the semantics of the code, like parameter or variable renaming, insertion/deletion of the diamond operator, `final` keyword or `this` qualifier, are already detected by Teamscale and not reported as changed code.

## 4.4 IDE Integration

After the profiling data has been collected by the profiler, uploaded by the daemon and analyzed by Teamscale, it has to be integrated into the developers IDE. For this, the already existing *Teamscale Integration*<sup>2</sup> plugin for IntelliJ IDEA version 2020.1+ was extended.

The plugin listens on the *file opened* event, which is fired when the user opens a file in the editor. In case the event is fired, the plugin determines the current VCS commit of the project and triggers a performance data load from the configured Teamscale server using the configured executable, baseline executable, instances and tasks. The result is then stored in a local cache, which is limited to the performance metrics of 1000 methods. Afterwards, the respective in situ visualization icons are created and each icon is stored in a new temporary file. Unfortunately, this is necessary as IntelliJ only supports the loading of icons from files and not directly from memory.

The icons are then integrated into the line gutter. In order to place them at the correct line, the Teamscale server already included the expected line based on the VCS commit. In case the file changed locally, the lines are adjusted based on a locally computed token difference of the unchanged file and the current changed file.

Referenced callers/callees are looked up in the IntelliJ internal PSI.

In addition, the plugin listens on the *selection changed* event, which is fired when the user switches the currently displayed file to another already opened file. Afterwards, the performance data is also loaded from the Teamscale server. However, if the cache already contains performance data for the respective file, the loading is skipped and the cached value is re-used.

The mentioned implementation has some limitations:

- Lambdas and anonymous inner classes are not resolved and therefore do not show any performance metrics. The resolution would require additional steps as lambdas and anonymous inner classes use a dynamic runtime name based on their declaration order.

---

<sup>2</sup><https://plugins.jetbrains.com/plugin/9431-teamscale-integration>

- Static initializer blocks are not supported as they are not identified by name, which results in the same CCT node for every block. This could be avoided by moving all the initializer code into a static method and only calling the method from the block.
- Constructors of non-static inner classes are not resolved. This is due to the fact that at runtime the constructor gains an additional parameter with the type of the enclosing class. The parser is currently not able to identify the difference between static and non-static inner classes, therefore the constructor is never found in the CCT.
- Line numbers in the invocations view are not updated when the respective code changed (either locally or in an already analyzed commit). As a result, jumps to callers by using the invocation view will jump to the displayed line number. This may not be the correct location anymore as the caller class changed. In order to overcome this issue, every caller class must be checked for changes and the line numbers re-computed.

## 5 Evaluation

The following chapter describes how the implemented approach was evaluated. Section 5.1 describes the research questions that were used as foundation for the following studies. Section 5.2 analyzes the performance overhead on the client machine induced by the implemented profiling and daemon approach. The implemented IDE integration is evaluated in a user study, which is described in Section 5.3.

### 5.1 Research questions

The design of the following studies and the evaluation was driven by the following research questions:

#### **RQ 1: How usable is the implemented approach in real-world scenarios?**

The goal of this thesis is to propose an approach that can be used in real-world production environments. These environments introduce certain limitations on the host machine:

1. The proposed profiling approach must not slow down the profiled application significantly (meaning below 2% additional performance overhead)
2. The daemon implementation should not require a huge amount of main memory

In addition, the analysis of the uploaded runtime data should be reasonably fast, whereby the maximum time is denoted by the amount of time until the next daemon upload occurs.

#### **RQ 2: Does the inclusion of runtime information in the code provide added value for the developer?**

Including the runtime information into the IDE, should aid the developer and not distract them. For this, the runtime information should provide additional information that is not visible from the static source code and can also be used effectively. There are many ways in which the assistance could be provided and different developers may find different parts distracting or helpful.

**RQ 3: Does the inclusion of tasks provide a good aid in order to delimit the information to certain use cases?**

The newly introduced system of *tasks* should provide an useful tool to filter the resulting CCT for specific use cases. The use should be intuitive and the developer should be able to filter the CCT according to their needs. In addition, it should be possible to use them in the normal development workflow without causing interruptions or stalls.

## 5.2 Performance overhead

The following section focuses on the various overheads induced on the client machine running the profiled application. Subsection 5.2.1 focuses on the carried out studies and Subsection 5.2.2 describes and discusses the various results.

### 5.2.1 Study design

The performance overhead of the JFR profiling approach was analyzed by utilizing the DaCapo Benchmark Suite [11] Version 9.12-MR1-bach. In order to compare the performance, each benchmark was executed for each of these four different configurations:

1. Without any profiling, to establish a baseline
2. With the implemented agent approach (see Section 4.2)
3. With the default JFR configuration using the JVM flag:  
`-XX:StartFlightRecording=settings=default`
4. With the extended JFR configuration using the JVM flag:  
`-XX:StartFlightRecording=settings=profile`

Each benchmark was run for 30 successive iterations with each configuration in a single OpenJDK 11 instance. The data from the first 20 iterations was discarded to compensate the JVM's startup phase. Each benchmark was further executed 10 times (with 30 iterations each) to ensure the results are not biased by any optimization decision the JVM had made in the warm-up phase. The benchmarks *batik*, *eclipse*, *tomcat*, *tradesoap* and *tradebeans* were not executed as they are currently not executable<sup>1</sup>. All tests were executed on a system with a quad-core Intel Core i7-10510U processor with 16 GB of memory running Ubuntu 18.04 LTS. In order to further stabilize the results, hyper-threading, turbo boost and dynamic frequency scaling was disabled. No other applications except required system services were running while

---

batik/eclipse:	<a href="https://github.com/dacapobench/dacapobench/issues/175">https://github.com/dacapobench/dacapobench/issues/175</a>
<sup>1</sup> tomcat:	<a href="https://github.com/dacapobench/dacapobench/issues/184">https://github.com/dacapobench/dacapobench/issues/184</a>
tradebeans/tradesoap:	<a href="https://github.com/dacapobench/dacapobench/issues/189">https://github.com/dacapobench/dacapobench/issues/189</a>

the benchmarks were executed.

As the daemon process is running idle in the background for most of the time, the influence on the CPU and thus runtime of the profiled application is negligible. However, more interesting at this point is the consumption of main memory. The daemon accumulates the various JFR dumps into an in-memory CCT and occupies memory that otherwise could be used by the profiled application. In order to benchmark the memory consumption, some JFR dumps that were produced during an earlier test execution were stored and reused for later analysis. The daemon was started with enabled JFR memory profiling and continuously supplied with the previously stored dumps. A new dump was provided every second and the upload was triggered every six minutes. This corresponds to a behaviour ten times as fast as it would occur in the real application, where a dump is provided every ten seconds and the upload is performed every hour. The daemon was not stopped for the whole benchmark in order to visualize the effect of a long-term running instance. The benchmark was executed for about one hour, which corresponds to the amount of data of 10 hours. Afterwards, the memory usage was analyzed using JDK Mission Control 7.

### 5.2.2 Results and discussion

Table 5.1: Average runtime of various DaCapo benchmarks in milliseconds, while being profiled using different approaches.

The percentage values indicate the relative performance change to the unprofiled benchmark.

Profiling Approach Benchmark	unprofiled	agent	default.jfc	profile.jfc
avrorra	1900	1892 (-0.4%)	1897 (-0.2%)	1905 (+0.3%)
fop	174	186 (+6.9%)	177 (+1.7%)	181 (+4.0%)
h2	2537	2579 (+1.7%)	2568 (+1.2%)	2657 (+4.7%)
jython	1243	1257 (+1.1%)	1257 (+1.1%)	1741 (+40.1%)
luindex	365	372 (+1.9%)	371 (+1.6%)	369 (+1.1%)
lusearch	643	650 (+1.1%)	653 (+1.6%)	659 (+2.5%)
lusearch-fix	643	654 (+1.7%)	663 (+3.1%)	664 (+3.3%)
pmd	710	732 (+3.1%)	722 (+1.7%)	756 (+6.5%)
sunflow	2325	2300 (-1.1%)	2331 (+0.3%)	2327 (+0.1%)
xalan	595	607 (+2.0%)	611 (+2.7%)	636 (+6.9%)
combined	1113	1123 (+0.8%)	1125 (+1.0%)	1189 (+6.8%)

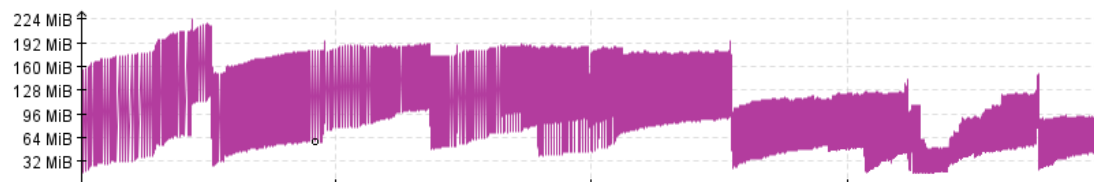


Figure 5.1: Timeline indicating the used heap memory of the running daemon application

Table 5.1 illustrates the performance of the various profiling configurations. As it can be seen, the agent approach is almost always inside the desired 2% performance overhead. The only exceptions are the benchmarks `fop` and `pmd`. When looking at the console output of these benchmarks, one sees that it contains several additional lines that are not present in the unprofiled (and other JFR) runs. As console output introduces additional performance overhead, this could be the reason for the higher runtime. However, why this additional output is produced remains unclear. Furthermore, when utilizing the profiling agent, the benchmarks `avrorra` and `sunflow` were executed faster than the unprofiled versions. One explanation could be a variation in the JVM decisions and thus performance improvement. However, a final explanation cannot be provided as too many factors can change the performance of a Java program.

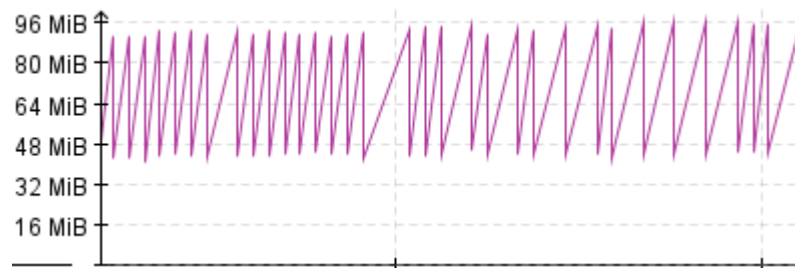


Figure 5.2: Zoomed in timeline demonstrating the memory spikes produced by file parsing

Finally, it can be concluded that the agent approach seems to have a similar runtime overhead to the `default.jfc` configuration, while maintaining a doubled sampling rate, and a better overhead than the `profile.jfc` configuration, which uses the same sampling rate but many additional JFR events.

Figure 5.1 illustrates the monitored memory consumption of the daemon. It can be seen that the memory consumption is moving gradually upwards as the CCT size gradually increases with every new processed dump. After an upload has been performed, the memory of the CCT can be reclaimed, which results in the various larger drops. Additional memory is required for reading and parsing the respective JFR dump file and can be easily reclaimed after the parsing has been completed, which results in the huge variation in short amount of time (for a zoomed in image see Figure 5.2). However, the maximum amount of required memory never exceeds 225 MiB and later seems to further stabilize below 150 MiB. As modern systems often have multiple GiB of main memory, the daemon should not interfere with the execution of other programs.

These results indicate that the suggested profiling approach in combination with the daemon can be used effectively in a real-world production environment.

## 5.3 User study

The following section describes how the implemented IDE integration was evaluated. The design and execution of the user study is described in Subsection 5.3.1, and the results are discussed in Subsection 5.3.2. Finally, Subsection 5.3.3 completes the section by describing possible threats to validity of the results.

### 5.3.1 Study design

An user study with 7 developers was performed over a period of four weeks. During the time of the study, the participants were asked to use the newly developed IDE plugin in their daily working routine. Since a goal was to explore the usefulness of the tool for normal programming activities, no special development tasks or artificial problems had to be solved.

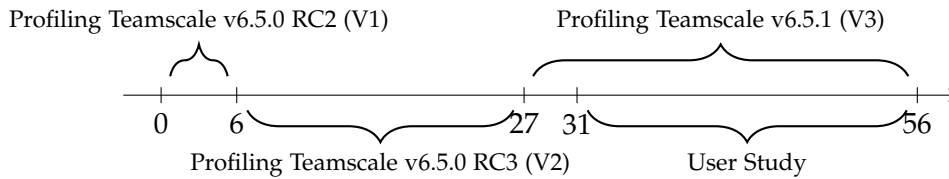


Figure 5.3: Timeline indicating the respective profiling, and study intervals  
Unit of time is in days

Table 5.2: Overview of study participants.

The experience is given in years

Participant	programming experience	Java experience	Teamscale experience	used profilers	reason for profiler usage
P1	4–5	4–5	2–3	none	none
P2	9+	2–3	2–3	JFR, Visual VM, JProfiler, YourKit	Detect/Fix performance problem
P3	4–5	4–5	2–3	JFR, Visual VM	Detect/Fix performance problem
P4	9+	9+	4–5	Visual VM, YourKit	Detect/Fix performance problem
P5	6–8	6–8	0–1	JFR	Detect performance problem
P6	2–3	2–3	0–1	none	none
P7	6–8	6–8	2–3	Visual VM	Detect/Fix performance problem

In order to gather enough profiling information, three different versions of Teamscale have been profiled over a period of eight weeks. Two versions were already profiled when the study started and the last version was profiled and analyzed while the study was running. For the sake of simplicity, these versions will from now on be called V1, V2 and V3. Figure 5.3 shows a timeline of the profiling and study period. The profiled instance was a *shadow* instance, meaning it is set up exactly like the production instance, but does not receive the main traffic from the developers. The profiling data was uploaded to another Teamscale instance, which was set up especially for this study. This instance then processed the uploaded data and served requests from the IDE plugin. After the study period of four weeks, an interview was performed with each study participant individually.

Before the study started, each developer was asked to fill out a survey indicating their experience in general programming, programming in Java and programming for Teamscale. In addition, the participants were asked about their previous experience and usage of profilers. The results are represented in Table 5.2.

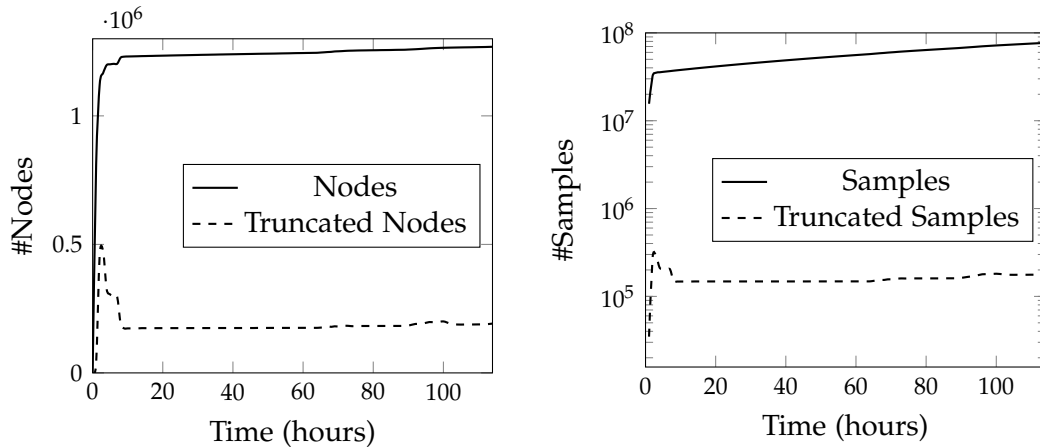


### 5.3.2 Results and discussion

The results produced by the study can be analyzed from two perspectives: First, a technical analysis on the resulting CCTs and the respective analysis provides an overview about how the approach performs in a real-world scenario. Secondly, the results provided by the study participants offer information about how useful the computed results can be used in the development process.

#### Technical Analysis

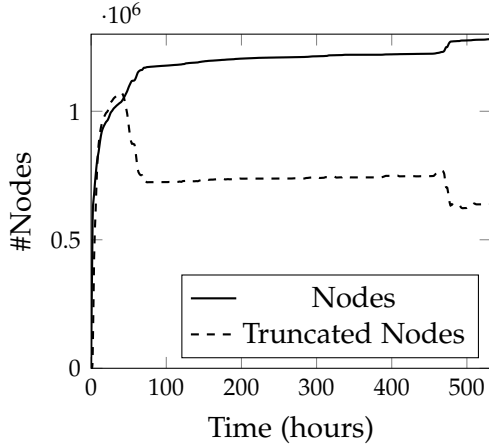
During the course of the study it became evident that the performance of the analysis was too slow in its current state, with the goal to be useful in a real-world scenario. One main reason for this was the huge amount of new CCT nodes uploaded during the first hours of profiling. For example, the first daemon upload already resulted in about 810 000 new nodes (64% of total nodes) for V1, 570 000 (45%) for V2 and 480 000 (33%) for V3. This results in long runtimes for the `RecursionInformationSynchronizer` and `TaskIndexTreeSynchronizer` as they both require around  $O(|node_{new}| \times |node_{all}|)$  time, which maximizes when  $|node_{new}| = |node_{all}|$ . For example, the maximum runtime of the `RecursionInformationSynchronizer` for a single daemon upload was around 31 hours, and for the `TaskIndexTreeSynchronizer` around 4 hours, while the average runtime was around 8 minutes (recursion) and 14 minutes (task). This in turn delays all other scheduled or upcoming analysis, for example, when new code gets pushed to the repository, or the daemon uploads new profiling data, as it can only continue when the previous upload was fully analyzed. When the profiling period continues, the amount of new nodes becomes gradually smaller and the analysis can catch up again. In this case, the profiling of V3 started only a few days before the study and the analysis did only catch up after a full week.



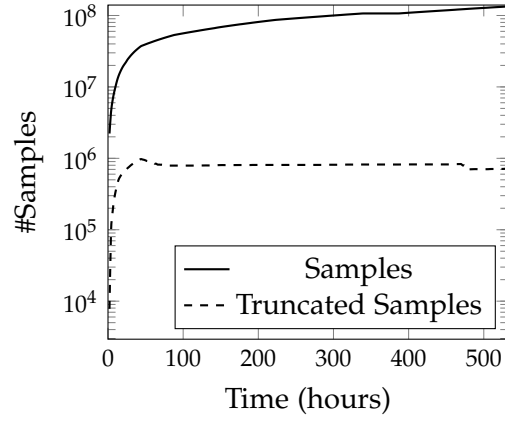
(a) Node count over time

(b) Sample count over time

Figure 5.4: CCT statistics of the Teamscale v6.5.0 RC2 (V1) instance

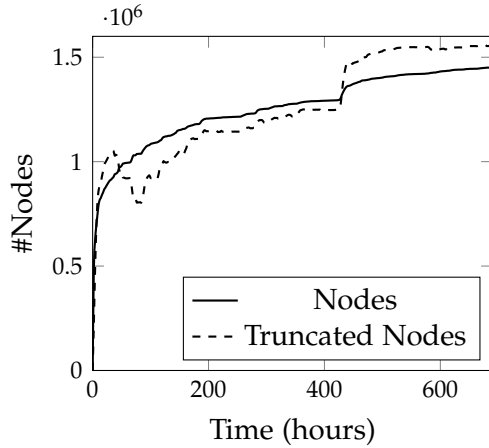


(a) Node count over time

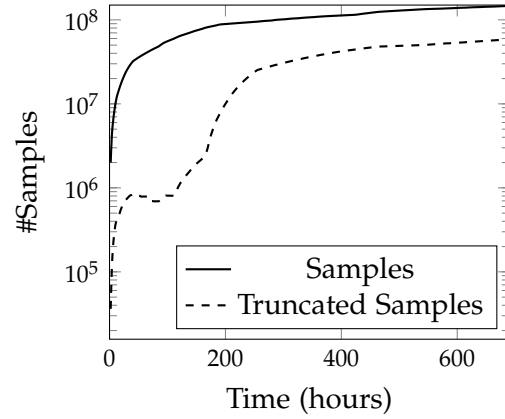


(b) Sample count over time

Figure 5.5: CCT statistics of the Teamscale v6.5.0 RC3 (V2) instance



(a) Node count over time



(b) Sample count over time

Figure 5.6: CCT statistics of the Teamscale v6.5.1 (V3) instance

Another thing to note is the amount of truncated nodes and samples. The Figures 5.4, 5.5 and 5.6 display the amount of nodes and truncated nodes (a), and samples and truncated samples (b) for the respective profiling runs. As it can be seen, after an initial rise, the analysis was able to quickly merge a large part of the truncated nodes into the full CCT. For the first two profiled versions (V1 and V2), the amount remained pretty stable afterwards and did not greatly increase or decrease anymore (Figure 5.4a/Figure 5.5a). While the share of truncated nodes remained comparatively high (around 12% for V1 and 33% for V2), the amount of samples associated to these

truncated nodes remained rather low (around 0.25 - 0.5%) (Figure 5.4b/Figure 5.5b). This is due to the fact that the samples are mostly contained in the leaves of the truncated CCT, yielding a high node count for a relatively low sample count. However, for V3 the amount of truncated nodes and samples further increased until 52% of all nodes and 29% of all samples were not merged into the complete CCT anymore (Figure 5.6). For which reasons exactly the analysis was not able to keep its promising results from the first versions remains unclear. Maybe the additional workload introduced by the study participants accessing the analyzing instance slowed down the merging process, which was limited to 50 minutes per daemon upload. Another reason could be that the storage system was not able to efficiently handle the amount of nodes anymore, as the CCTs of the previous profiled versions remained in the store. One option to reduce the number of truncated nodes and samples could be to increase the maximum stack depth in the JFR profile. This would result in less truncated samples, and the remaining truncated samples could be merged more easily as the CCT gains additional precision and the total amount of truncated nodes decreases, leaving more computation time for the remaining.

Another important thing is the performance on the proposed task analysis. As already stated, the synchronization of tasks while the CCT grows is (apart from the initial analysis) comparatively fast. However, the initial analysis - when a new task definition is introduced - requires also a great amount of time. For example, the introduction of a new task definition at the end of V3 profiling required around 14 hours. Tasks were supposed to be usable as ad-hoc feature while analyzing the performance of a system. A runtime above a few minutes renders them unusable for the original purpose.

### **Interview Feedback**

Out of the seven participants, three did not use the plugin at all (P1, P4, P6) and the remaining four (P2, P3, P5, P7) only partially. The main reason was technically driven as an error in the study setup only allowed the profile information to be viewed when working on the master branch or the release branch of Teamscale v6.5.x. As most of the development work occurs on different branches, the plugin was not usable there. P1 gave as an additional reason that they were working only on completely new code, so the profiling information would not have been of much use anyway. P4 also mentioned that they were not actively programming during the study time. P6 left the study after two weeks because their working contract ended. Because of this, only P2, P3, P5, and P7 are considered for the remaining discussion. P2 and P7 used the plugin most of the time when developing. However, their absolute usage time was rather low as they were mostly busy with other non-development related tasks. P3 and P5 were also working on mostly new code and different branches, where no profiling information was available.

P3, P5, and P7 described the invocation bars inside the method (white-box perspective) being helpful, because they indicate at which call site the most actual work is performed. They also commented that the method performance (black-box perspec-

tive) is rather uninteresting as it was (almost) always 0.00%. Here, a more detailed information could be useful, for example "x out of y samples".

P2 was particularly interested in the caller/callee perspective as it gave clues towards the actual classes/implementations regarding interfaces and inheritance, which is not always clear from the static source code and the runtime information provides a better certainty. However, they also pointed out that it is important to know the source of the runtime information in order to correctly interpret the shown information.

A similar remark was given by P3, where they wanted to investigate a performance problem regarding a service call. However, it seemed that this particular service was not called in the profiled instance, so they had to fall back to a local test setup. They commented, that it would be very helpful to import a local profiling result into the IDE, and skip the remote analysis by Teamscale or rather perform it locally in the IDE. The performance impact on the IDE of such a local analysis is unclear and would require additional investigation.

P7 had a similar problem, where they wanted to investigate a particular call chain, which resulted in performance problems on a customers instance. However, again this chain was not called in the profiled instance, and the provided information was of not much use. They also mentioned, that if the profiling information of that particular instance would be available, it would have been of great use.

Regarding the newly introduced task feature, only P2 used this feature. P3, P5, and P7 had no specific use case, and P3 pointed out, that the feature was currently too slow (as already mentioned) in order to be used efficiently. P2 only used it to try it out and also had no specific use case.

Overall, the reception of the plugin was rather positive. It was only damped by the technical problems and reduced reliability of the profiling data due to the scope of the profiled instance. Every participant noted, that they are very interested in future development regarding the inclusion of runtime information. Only P7 remarked, that the displayed information is in general rather distracting, as it is not required in 99% of use cases.

## **Final discussion**

Revisiting the proposed research questions, following answers can be given:

### **RQ 1: How usable is the implemented approach in real-world scenarios?**

Both the profiling and daemon approach require few additional resources on the executing machine. Benchmarks indicate, that they can be safely used in a production environment, as they do not slow down the profiled application or require much additional main memory. On the other hand, the analysis of the produced profiling data requires more tuning. Especially, processing the first uploads, which introduce a great amount of new CCT nodes, is currently too slow. Whether the underlying

algorithms or the respective implementation is the main factor, requires further investigation.

**RQ 2: Does the inclusion of runtime information in the code provide added value for the developer?**

Every participant noted that in some way the runtime information was or could be helpful for development. Whether it can only be used in certain situations or as an always-on feature remains a subjective manner. In terms of performance, it became evident that for large systems, the runtime of a single method in regard to the global execution time seems to be rather uninteresting and not helpful. However, the white-box perspective was perceived very positive, as it seems to give additional information about how a method spends its own execution time, i.e., performs most of its work. This seems to be useful even outside a performance related perspective, as more work usually means more business logic. In addition, the inclusion of runtime callers and callees was also considered helpful and supplementary to the static code analysis provided by the IDE.

Important however, is the source of the runtime information. This is especially crucial for systems with many different application areas such as Teamscale. Here, one instance can behave completely different from another, as customer A for example uses mainly Python code in combination with Gitlab integration, and customer B uses a C++ codebase with a Bitbucket integration. In case the Gitlab integration needs to be inspected, the profiling information from customer B is rather useless. This was already considered with the inclusion of different instances, but an evaluation with multiple instances is required to verify the approach.

**RQ 3: Does the inclusion of tasks provide a good aid in order to delimit the information to certain use cases?**

Tasks were considered a tool to avoid the already noted global execution time. As the usage in the user study was almost non-existent, a final answer can not be provided. However, it became clear, that some mean must exist in order to delimit the CCT for specific use cases. Currently, this is done by executing only certain parts in an application with a tailored test case, and then only inspecting the specific results. In a continuously profiled application this is not possible. However, it already became clear, that the current approach is too slow in order to be effectively used by the developer in their standard workflow.

Another possibility could be a more dynamic approach. For example, the developer could mark the current method as new "task root" and further performance data is computed relatively to this root until a new root is chosen, or the current choice is removed. This approach could perform way faster, as the nodes of a single method can be looked up faster than the current full CCT scan approach. The respective performance data can then be computed by following the branches of the marked

subtrees. Maybe a graph based storage system (for example, neo4j<sup>2</sup>) could perform better than the current key-value based system of Teamscale.

### 5.3.3 Threats to validity

Reflecting the character of the user study, collected results can only be considered a groundwork for future research.

Particularly the internal validity might be affected: The general usage of the approach was rather low, and only four out of seven participants could provide helpful information. With this small number of participants, a subjective bias cannot be excluded. The results could further be biased, as all participants worked for the same company (CQSE GmbH) and one participant was even an advisor to this thesis. In addition, all results were conducted from an interview resulting only in opinions without quantitative measurable content. A new study with more participants, longer period and improved tool usability, favourably also using different profiled applications, could provide more insights regarding the usefulness of the implemented approach.

Despite the limited amount of participants, a certain external validity can be achieved: The approach was applied on a large-scale application and executed in a real world environment. A limitation could be, that the resulting data was not analyzed in a production environment. However, the results already indicate, that the current analysis performance is not good enough in order to be used in such an environment. Further, the implementation is limited to Java applications only. While the suggested approach is independent of the respective programming language, some implementation decisions were made deliberately with Java in mind. How good the integration could work with other languages requires further investigation.

---

<sup>2</sup><https://neo4j.com/>

## 6 Related work

The field of performance profiling and development process integration produces a variety of research. The following chapter describes some connected research to this thesis. Section 6.1 focuses on profilers. A good profiler should have as less performance impact as possible, while still providing good accuracy. Section 6.2 focuses on the research regarding data structures for the emitted profiling data, and Section 6.3 describes some approaches that can be used for evaluating the final CCT. Various approaches about how the obtained information can be presented to the developer, are described in Section 6.4. Finally, Section 6.5 focuses about other approaches how runtime data can be integrated into the developers IDE.

### 6.1 Profiling

Profiling is nowadays performed either via sampling or via code instrumentation. Both approaches yield their own advantages and disadvantages.

Zhuang et al. introduce a new technique called *adaptive bursting*, which utilizes both sampling and instrumenting in order to combine their advantages [52]. Adaptive bursting utilizes the fact, that application control flow is highly repetitive. In this approach, sampled profiling is used to identify calling contexts. In case a new calling context is identified a *burst* for this context is initiated. A burst is a short period of time (e.g., around 0.2 ms) where the instrumenting profiler is used to collect accurate call/return samples. In case the sampled calling context is already known, a burst may be randomly performed or skipped based on a user defined *re-enable ratio* (RR). This RR is also used to perform a weight compensation for the edge weights of the resulting calling context tree. This addresses two problems: 1) as runtime behaviour may change, new call patterns will not be represented in the calling context tree, and 2) by disabling bursting for common calling contexts, the respective edge weights may become inaccurate, as these common call/return sequences are sampled with the same frequency as other more rare call/return sequences.

For example a RR of 0.25 is chosen. A burst is then performed for roughly every 4th sample of the same calling context. For every burst, the edge weight is then multiplied by  $\frac{1}{RR} = \frac{1}{0.25} = 4$  in order to compensate for the other times when the burst is skipped. In case the calling context was not previously encountered, no edge compensation is performed, as a burst is definitely executed.

The RR value represents the trade-off between accuracy and performance overhead. A higher RR yields more bursts, which makes the result more accurate, but also requires more performance overhead.

Experimental results show that the adaptive bursting approach (with  $RR = 0.05$ ) achieves 85% degree of overlap to the complete calling context tree while maintaining a moderate slowdown of 20%. In contrast, sampled stack walking had an overlap below 50% and the complete instrumentalization approach yields a slowdown of 5000%, with an accuracy comparable to the adaptive bursting approach. The authors expect the slowdown to further reduce with a more efficient instrumentation implementation.

This thesis also focuses on the selection of a correct profiler. In fact, the author has tried to implement the adaptive bursting approach by utilizing the JFR. However, due to the asynchronous nature of the JFR, it was not possible to introduce an adaptive bursting mechanism without further changes inside the JVM. Future work may greatly improve from this mechanism, as it provides more detailed execution information.

## 6.2 Data structures

The produced profiling data can be collected into various data structures. Structures, which keep the data in a calling context sensitive way, have been identified to be useful for further analysis.

M. Serrano et al. proposed an approach to build an approximate calling context by using partial call traces [46]. In this approach, small call traces with no fixed start or end point (thus partial) are captured. This can be done via different approaches, for example, by dynamically enabling and disabling an instrumenting profiler. Some processors even provide hardware support without the need of any instrumentation. These partial call traces are then merged into few partial CCTs by: 1) concatenating call traces based on their common sequences, then 2) converting these call traces into partial CCTs, and finally 3) merging these partial CCTs through maximal matching. The final partial CCTs capture only the hot portions of the full CCT. Thus, they are much smaller, as the colder paths of the full CCT rarely get executed. The authors tested their approach on several Java benchmarks and showed that their merging strategy exhibit a maximum 1% inaccuracy when compared to the exact solution. The approximate calling context has many applications, where the complete context is not necessarily required, for example, context sensitive inlining or object lifetime prediction.

This thesis borrows the proposed maximal common merging approach, in order to merge truncated stack samples emitted by the JFR into the complete CCT.

Ausiello et al. studied a generalization of edge and context-sensitive profiles by introducing the novel data structure called *k*-calling context forest (*k*-CCF) [4]. The nodes of a *k*-CCF associate performance metrics to paths of length at most *k*, that lead to each distinct routine of the program. A *k*-CCF provides *vertex profiling* [20] for  $k = 0$ , *edge profiling* [20] for  $k = 1$  and full context-sensitive profiling [2] for  $k = \infty$ , as well as any other intermediate point. The *k*-calling context of a routine is defined as:

Let  $\Delta = \langle r, \dots, v \rangle$  be a calling context of a node *v* starting from the root *r*, then the



k-calling context of  $v$  in  $\Delta$  is the maximal suffix of  $\Delta$  with length at most  $k$  (not counting  $v$ ). For example if  $\Delta = \langle r, a, b, c, v \rangle$  then the 2-context of  $v$  is  $\langle b, c, v \rangle$ .

The authors showed that the k-CCF can provide effective space-accuracy tradeoffs regarding inter-procedural contextual profiling. In addition, the k-CCF yields useful clues to the hot spots of a program, that may be hidden in a CCT, while using less space for small values of  $k$ . In contrast to the CCT as chosen in this thesis, the k-CCF provides a more method-centric representation with special focus on the calling context. This can be particularly useful, when a specific method is analyzed as a black-box, and the calling context is of more interest. However, due to its nature, the callees of a method are not represented in a compact way and have to be collected over all k-CCFs, which renders the data-structure mostly unusable for the white-box perspective.

Another data structure is proposed by D’Elia et al. They introduce the *hot calling context tree* (HCCT), that represents all the hot calling contexts of a program execution in a compact way [16]. The HCCT is a subtree of the complete CCT, that only includes hot nodes and their ancestors. It can be build on the fly by using data streaming algorithms, without loosing any frequent items (i.e., hot nodes) and only containing some additional false positives (i.e., nodes that are not hot, but close to). The HCCT cannot be calculated exactly in small space. Instead an *approximate hot calling context tree*  $(\phi, \epsilon)$ -HCCT can be computed by using the two parameters  $\phi$  and  $\epsilon$ . First, the frequency threshold  $\phi \in [0, 1]$  defines when a calling context is regarded as hot. A calling context is regarded as hot, if the frequency count of the corresponding CCT node is larger than  $\lfloor \phi N \rfloor$  where  $N$  is the sum of the frequency counts of all CCT nodes. The second parameter  $\epsilon < \phi$  controls the degree of approximation. It guarantees the frequency count of the false positives to be at least  $\lfloor (\phi - \epsilon)N \rfloor$ .

An accurate choice of  $\phi$  and  $\epsilon$  greatly affects the space used by the construction algorithm and the accuracy of the final solution. The required space is roughly proportional to  $\frac{1}{\epsilon}$ , and experimental results of the authors suggest that it is sufficient to use  $\epsilon = \frac{\phi}{5}$ . However, the choice of  $\phi$  is more difficult, as the total frequency count  $N$  is unknown before profiling. The performed experiments suggest, that an appropriate choice of  $\phi$  is mostly independent of the specific benchmark and stream length. For example, an assignment of  $\phi = 10^{-4}$  corresponds to mining roughly the hottest 1000 calling contexts.

The authors show that the  $(\phi, \epsilon)$ -HCCT achieves a similar precision as the full CCT in a much smaller space.

## 6.3 CCT evaluation

Modern, large-scale, object-oriented programs often consist of thousands of methods. Software quality standards require these methods to be relatively small, which leads to complex calling patterns at runtime. In addition, the use of generalised frameworks and design patterns lead to many layers of abstraction. As a result, the runtime is thinly distributed across many locations. This leads to difficulties when trying

to identify performance improvement opportunities, as traditional profilers provide method-centric feedback, which is too fine-grained.

Maplesden et al. introduce a novel approach to identify entry points to repeated patterns of method calls called *subsuming methods* [31]. The performance cost over these subsuming methods can be aggregated, which gives the developer hints for additional improvement opportunities. Effectively, subsuming methods define a new way to partition the CCT at a coarser granularity than the initial method level partitioning.

For this, each node in the CCT gets labeled either as subsuming or as subsumed. The costs of a subsumed node are attributed to their parent node until a subsuming method node is reached, where the cost is aggregated into the so called *induced cost*. The induced cost of a method is the sum of all induced costs of the respective nodes in the CCT.

In order to identify subsuming methods, the authors considered two characteristics of a method  $m$ : First, the method height  $H(m)$ , which denotes the maximum height of each subtree of each node of  $m$ . The second characteristic is the minimum dominating method height  $dmd_{\min}(m)$ . A method  $p$  dominates a method  $m$ , if  $p \neq m$  and  $\forall v \in nodes(m)$  exists a node  $n$ , such that the method of  $n$  is  $p$  and  $n$  is an ancestor of  $v$ . The distance function for the dominating method  $p$  of a method  $m$  is then defined as:

$$dmd(p, m) = \max_{v \in nodes(m)} d(p, v)$$

where  $d(p, v)$  denotes the length of the path from  $v$  to the first ancestor node with method  $p$ . The minimum dominating method height for a given method  $m$  is then:

$$dmd_{\min}(m) = \min_{p \in M} dmd(p, m)$$

where  $M$  is the set of all methods in the CCT.

The set of all subsuming methods  $S$  can then be calculated based on the user defined minimum height  $H_{bound}$  and minimum dominating method height  $D_{bound}$ :

$$S = \{m \mid m \in M \wedge H(m) > H_{bound} \wedge dmd_{\min}(m) > D_{bound}\}$$

The authors evaluated their approach in an experiment using the DaCapo-9.12-bach suite [11] using a  $H_{bound}$  and  $D_{bound}$  of four. Across the evaluated benchmarks a median of 6.12% of all methods were subsuming methods, and the median size of the subsuming CCT was 11.82% of the full CCT. Using subsuming methods, they were able to improve the performance of the benchmarks h2 (by 17%), fop (by 22%) and tomcat (by 57%).

The authors further performed an industrial case study with the developers of Letterboxd<sup>1</sup> [30]. Using the subsuming method approach, the developers were able to reduce the CPU load by 54.8% and the average response time by 49.6%.

---

<sup>1</sup><https://letterboxd.com/>

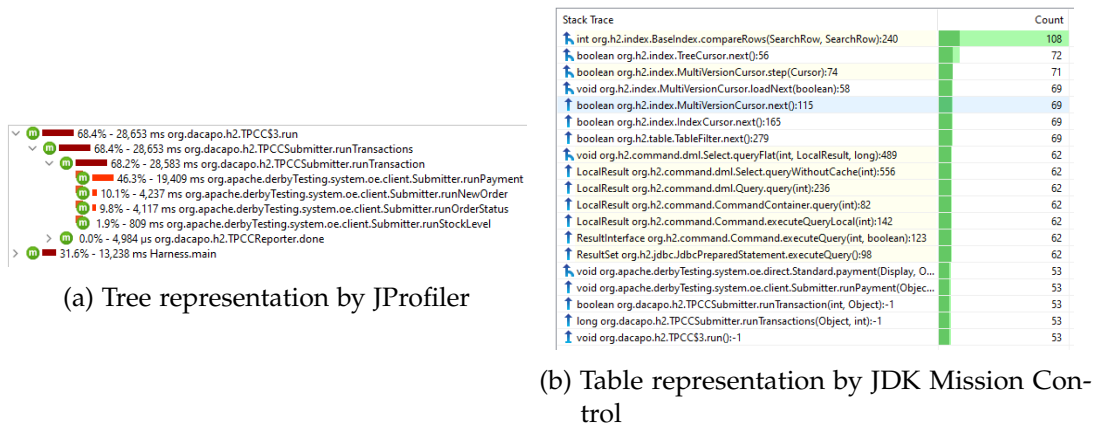


Figure 6.1: Text-based views by JProfiler and JDK Mission Control

## 6.4 Performance visualizations

CCTs are generally huge, containing millions of nodes, which makes analysis and especially visualization difficult. Current tools (such as JProfiler<sup>2</sup> or JDK Mission Control<sup>3</sup>) typically use a straight-forward approach to represent CCTs, such as text-based tree views or tree tables. Figure 6.1 displays some examples.

Despite its simplicity, there are drawbacks to this simplistic approach [10]:

- All nodes have the same size, independently of their respective metrics. Coloring can help, but too much may also clutter the display.
- Related functions (e.g., of the same class) are visually separated.
- Interaction problems: Either the whole CCT is already expanded, resulting in a visualization, that is usually too huge for a human to understand, or users must expand each tree branch individually, which often requires a great amount of clicks to reach the desired target.

<sup>2</sup><https://www.ej-technologies.com/products/jprofiler/overview.html>

<sup>3</sup><https://www.oracle.com/java/technologies/jdk-mission-control.html>

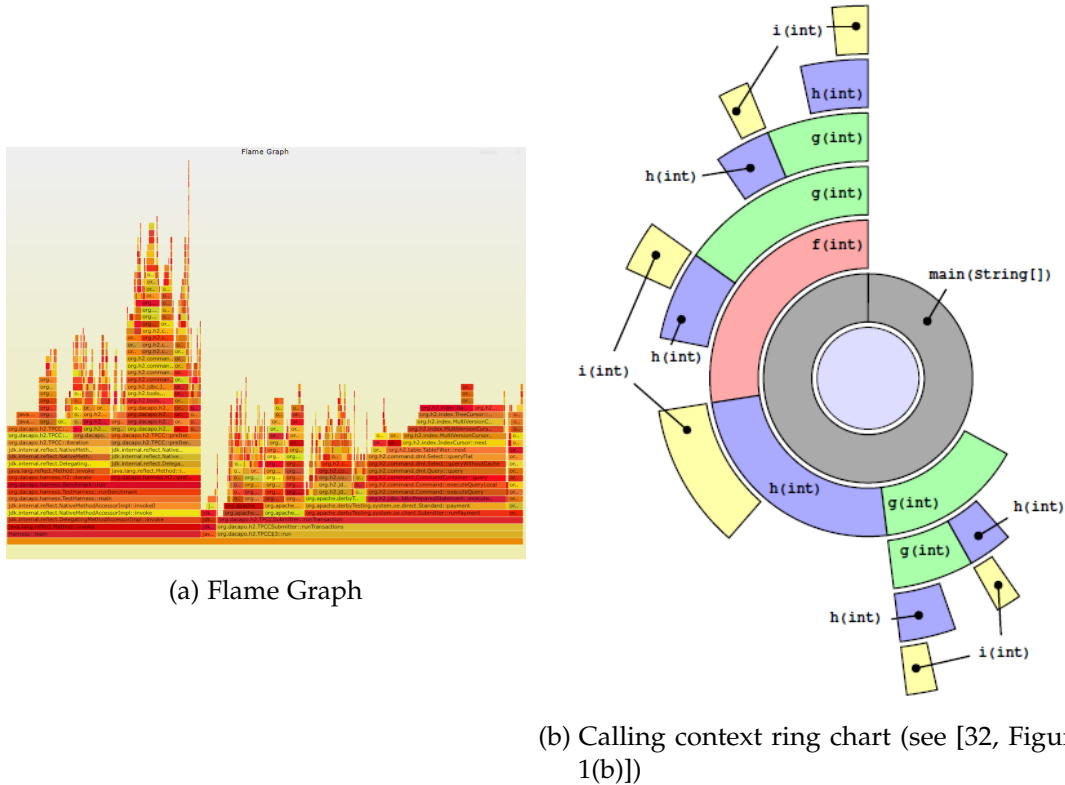


Figure 6.2: Visualizations where the respective size is computed based on the selected metric

Over the past years, flame graphs [21] have become more and more popular and are used in profiling tools such as YourKit<sup>4</sup> or as standalone<sup>5</sup>. A flame graph represents the call tree via hierarchy and the respective execution time via size. For an example see Figure 6.2a. Of course, it is also possible to size the respective method blocks via other metrics. One drawback is again the required space to display the possibly huge structure.

A more method centric approach is the use of ring charts [33], which provide a radial view, where callees are represented by ring segments around the callers core segment (Figure 6.2b). The respective arcs and area of the segments can be computed based on different metrics. The displayed depth can also be limited, in order to save space.

Bergel et al. even introduce a complete framework, which allows the transition between various complementary interactive visualizations [10]. The combination of these visualizations describes a flow that covers the CCT trees, the function graph, the processes or functional units, and the metrics list.

Another approach is taken by Beck et al. They propose a so called *method execution*

<sup>4</sup><https://www.yourkit.com/>

<sup>5</sup><https://github.com/brendangregg/FlameGraph>

*report* [8], which summarizes the execution of a specific method using natural-language text and embedded word-sized graphics [9, 19]. The report includes an overview of the dynamic calls and runtime-consumption of the selected method. In a user study, the report was positively perceived, however it was noted, that the information would be more useful directly in the source code and not in a distinct web page. This thesis provides similar metrics as proposed in the method execution report. However, the information is encoded into word-sized graphics embedded in the source code. Further information, such as the actual callers, callees, and threads, can be retrieved by utilizing popups.

## 6.5 IDE integration

The integration of runtime information into the developers IDE is no new approach. However, there exist multiple different approaches about what information and how it is integrated. Beck et al. propose the integration of small, word-sized graphics for numeric variables [6]. These graphics display the number of read and write accesses, in addition to the respective value changes, on a timeline. Another approach is taken by Winter et al. They do not integrate performance metrics, but display how often a log statement or exceptions happen in production [51]. The integration of runtime information via source code annotations is proposed by Sulär et al. [48]. This has the advantage, that the annotations can be checked into a VCS, and thus shared with the whole development team. However, the authors also note that this approach is not suitable for every kind of runtime information, as it could lead to unnecessary pollution of the source code and problems when merging different code states. Lieber et al. introduce a special extension for JavaScript, which includes dynamic information like function call count, arguments, return values, and even complete call traces [29]. They identify their tool as debugging tool, as it is used by every developer independently, and only with locally gathered data.

A similar approach to this thesis is taken by the tool *Senseo* [43, 44]. It integrates multiple runtime metrics into the Eclipse IDE, such as actually invoked methods behind interfaces, method arguments, return types, number of invocations, and number of created objects inside a method in combination with the allocated memory. These metrics are then displayed using hovers and ruler columns and are further integrated into the package explorer. However, such detailed metrics require a more intrusive approach in the profiled application, which makes it less suitable for use with applications running in production.

Another similar tool called *PerformanceHat* [14, 15] goes beyond just visualizing runtime information. The idea is to use runtime information provided by profiling the application in production. This information is then used in combination with the abstract syntax tree of the underlying IDE, in order to provide an incremental performance analysis and predict the performance of newly written code. This thesis only focuses on the actually measured runtime. As the context of a method call can be crucial regarding the performance, the runtime prediction of new code is left for

the developer, who usually knows the context of the code they currently write.

## 7 Conclusion

In this thesis an approach to integrate the performance data of Java programs into the developers IDE was proposed. In order to efficiently gather profiling data, the JFR was instrumented to continuously profile an application. The performance impact of this approach was evaluated in a benchmark study, which resulted in around 1–2% runtime overhead, a reasonable overhead for use in real-world production environments.

The resulting profiling data was then collected and aggregated into a calling context tree using Teamscale. Various computations such as recursion detection and the merging of truncated stack traces were proposed. This allowed the computation of context specific metrics, such as caller and callee times. In order to reduce the focus of the performance data from the global system to a user defined sub-system, a novel approach named *tasks* was introduced. In this approach CCT nodes get labeled based on user provided task definitions, which allows a metric computation relative to the task roots, rather than the global CCT root.

In order to make the resulting performance data easily available for the developer, an integration into the IntelliJ IDEA IDE was implemented. This approach was based on the visualization proposed by Beck et al. [7]. It was further enhanced to allow the comparison of different profiled program versions (as already proposed but not implemented by Beck et al.). As source code is changed over time, a visual indicator is provided in case the displayed source code changed since the profiled version.

The resulting visualization was evaluated in a small user study. The study showed that the inclusion of runtime information can be helpful for program understanding, independent of the underlying programming task. However, as the information is gathered continuously over a long period of time, the performance of a single method can be neglected as it is almost always a very small percentage of the global execution time. This further shows, that an approach to filter the metrics calculation on specific CCT subtree(s) is required. The proposed task based approach was rather unhelpful, as the usability suffered from performance issues.

### 7.1 Future work

Several interesting research opportunities arise from this work. First of all, the size of the CCT posed challenging during the analysis regarding recursion and tasks. D’Elia et al. propose an approach to drastically decrease the size of the CCT [16], which could improve current and further analyses regarding the complete CCT.

In addition, the accuracy of profiling data provided by the JFR could be enhanced

by utilizing the bursting mechanisms proposed by Zhuang et al. [52]. This approach would make it possible to include method parameters or return values into the profiling data. However, the overhead imposed by additional profiling must still remain competitive.

During the evaluation period, it became clear that it should be possible to reduce the focus of the performance data from the global system to smaller user defined sub-system(s). The task based approach, proposed in this thesis, could be replaced by a more dynamic approach, where the necessity for a full CCT scan is no longer needed.

Another future implementation could use the commit based storage feature of Teamscale. This allows "time traveling", where data for an earlier commit is queried. As profiling data is uploaded over a long period of time, it is distributed over multiple commits. This potentially allows to analyze performance and/or usage on basis of a time-line, where performance changes over time could be identified. This may, for example, help with performance bug detection where problems arise only at certain time frames.

This thesis re-used the in situ visualizations proposed by Beck et al. [7] in order to include the performance information into the IDE. Other visualizations exist [8, 10, 24, 29, 48, 49], however most of them suffer from the distance to the actual source code. Further research on developing visual cues supporting the analysis of source code, could provide a better integration of performance data and further aid program understanding.

This thesis focused on analyzing a single performance indicator of a program: runtime consumption. However, other performance indicators exist that are also important to monitor during the profiling process. This includes: object allocation count and object lifetime, memory consumption, I/O operations, (amount of) thrown exceptions, or synchronisation and lock times. These indicators could also be profiled, analyzed, and integrated into the IDE.



## List of Figures

2.1	Illustrating the black-box and white-box perspective, based on [7, Figure 2] . . . . .	5
3.1	Alternating method execution . . . . .	10
3.2	Call tree for a single invocation of method a from Listing 3.1 . . . . .	13
3.3	Call graph for invocation of method a from Listing 3.1 . . . . .	14
3.4	Calling context tree for invocation of method a from Listing 3.1 . . . . .	15
3.5	<i>Maximal common merging</i> example . . . . .	17
3.6	Example for a CCT enriched with recursion information . . . . .	18
3.7	In situ visualization example . . . . .	24
3.8	Invocation view popup example . . . . .	24
3.9	Thread view popup example . . . . .	25
3.10	Example for a performance comparison to a baseline . . . . .	27
4.1	Component diagram illustrating the general solution architecture and its communication . . . . .	29
4.2	Flow chart illustrating analysis steps and their respective interactions . . . . .	34
5.1	Timeline indicating the used heap memory of the running daemon application . . . . .	42
5.2	Zoomed in timeline demonstrating the memory spikes produced by file parsing . . . . .	43
5.3	Timeline indicating the respective profiling, and study intervals . . . . .	44
5.4	CCT statistics of the Teamscale v6.5.0 RC2 (V1) instance . . . . .	45
5.5	CCT statistics of the Teamscale v6.5.0 RC3 (V2) instance . . . . .	46
5.6	CCT statistics of the Teamscale v6.5.1 (V3) instance . . . . .	46
6.1	Text-based views by JProfiler and JDK Mission Control . . . . .	55
6.2	Visualizations where the respective size is computed based on the selected metric . . . . .	56



# List of Tables

- 3.1 Examples of in situ visualizations for methods . . . . . 25
- 3.2 Examples for performance change arrow, where the baseline *method time* is always 0.1 . . . . . 27
- 5.1 Average runtime of various DaCapo benchmarks in milliseconds, while being profiled using different approaches. . . . . 42
- 5.2 Overview of study participants. . . . . 44



# Bibliography

- [1] A. O. Allen. *Introduction to computer performance analysis with Mathematica*. Computer science and scientific computing. Boston: AP Professional, 1994. ISBN: 0120510707.
- [2] G. Ammons, T. Ball, and J. R. Larus. “Exploiting hardware performance counters with flow and context sensitive profiling.” In: *ACM SIGPLAN Notices* 32.5 (1997), pp. 85–96. ISSN: 0362-1340. DOI: 10.1145/258916.258924.
- [3] M. Arnold and P. F. Sweeney. *Approximating the calling context tree via sampling*. 2000.
- [4] G. Ausiello, C. Demetrescu, I. Finocchi, and D. Firmani. “k-Calling context profiling.” In: *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*. Ed. by G. T. Leavens. New York, NY: ACM, 2012, p. 867. ISBN: 9781450315616. DOI: 10.1145/2384616.2384679.
- [5] P. Ayres and J. Sweller. “The Split-Attention Principle in Multimedia Learning.” In: *The Cambridge handbook of multimedia learning*. Ed. by R. E. Mayer. Cambridge handbooks in psychology. Cambridge: Cambridge University Press, 2014, pp. 206–226. ISBN: 9781107035201.
- [6] F. Beck, F. Hollerich, S. Diehl, and D. Weiskopf. “Visual monitoring of numeric variables embedded in source code.” In: *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*. Ed. by A. Telea. Piscataway, NJ: IEEE, 2013, pp. 1–4. ISBN: 978-1-4799-1457-9. DOI: 10.1109/VISSOFT.2013.6650545.
- [7] F. Beck, O. Moseler, S. Diehl, and G. D. Rey. “In situ understanding of performance bottlenecks through visually augmented code.” In: *2013 IEEE 21st International Conference on Program Comprehension (ICPC)*. Ed. by H. Kagdi. Piscataway, NJ: IEEE, 2013, pp. 63–72. ISBN: 978-1-4673-3092-3. DOI: 10.1109/ICPC.2013.6613834.
- [8] F. Beck, H. A. Siddiqui, A. Bergel, and D. Weiskopf. “Method Execution Reports: Generating Text and Visualization to Describe Program Behavior.” In: *2017 IEEE Working Conference on Software Visualization (VISSOFT)*. 2017, pp. 1–10. DOI: 10.1109/VISSOFT.2017.11.
- [9] F. Beck and D. Weiskopf. “Word-Sized Graphics for Scientific Texts.” In: *IEEE transactions on visualization and computer graphics* 23.6 (2017), pp. 1576–1587. DOI: 10.1109/TVCG.2017.2674958.

- [10] A. Bergel, A. Bhatele, D. Boehme, P. Gralka, K. Griffin, M.-A. Hermanns, D. Okanović, O. Pearce, and T. Vierjahn. “Visual Analytics Challenges in Analyzing Calling Context Trees.” In: *Programming and performance visualization tools*. Ed. by D. Böhme, J. A. Levine, M. Schulz, and A. Bhatele. Vol. 11027. LNCS sublibrary. Cham: Springer International Publishing, 2019, pp. 233–249. ISBN: 978-3-030-17871-0. DOI: 10.1007/978-3-030-17872-7\_14.
- [11] S. M. Blackburn, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. Eliot, B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, R. Garner, D. von Dincklage, B. Wiedermann, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, and D. Frampton. “The DaCapo benchmarks.” In: *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*. Ed. by P. Tarr. New York, NY: ACM, 2006, p. 169. ISBN: 1595933484. DOI: 10.1145/1167473.1167488.
- [12] M. D. Bond and K. S. McKinley. “Probabilistic calling context.” In: *ACM SIGPLAN Notices* 42.10 (2007), pp. 97–112. ISSN: 0362-1340. DOI: 10.1145/1297105.1297035.
- [13] P. P. Chang, S. A. Mahlke, W. Y. Chen, and W.-M. W. Hwu. “Profile-guided automatic inline expansion for C programs.” In: *Software: Practice and Experience* 22.5 (1992), pp. 349–369. ISSN: 00380644. DOI: 10.1002/spe.4380220502.
- [14] J. Cito, P. Leitner, M. Rinard, and H. C. Gall. “Interactive Production Performance Feedback in the IDE.” In: *ICSE 2019*. Los Alamitos, CA: IEEE Computer Society, 2019, pp. 971–981. ISBN: 978-1-7281-0869-8. DOI: 10.1109/ICSE.2019.00102.
- [15] J. Cito, P. Leitner, C. Bosshard, M. Knecht, G. Mazlami, and H. C. Gall. “PerformanceHat – Augmenting Source Code with Runtime Performance Traces in the IDE.” In: *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*. 2018, pp. 41–44.
- [16] D. C. D’Elia, C. Demetrescu, and I. Finocchi. “Mining hot calling contexts in small space.” In: *Software: Practice and Experience* 46.8 (2016), pp. 1131–1152. ISSN: 00380644. DOI: 10.1002/spe.2348.
- [17] H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and W. Gong. “Anomaly detection using call stack information.” In: *Proceedings / 2003 IEEE Symposium on Security and Privacy, May 11 - 14, 2003, Berkeley, California, USA*. Los Alamitos, Calif.: IEEE Computer Society, 2003, pp. 62–75. ISBN: 0-7695-1940-7. DOI: 10.1109/SECPRI.2003.1199328.
- [18] E. Gahlin and M. Grönlund. *JEP 349: JFR Event Streaming*. 2017. URL: <https://openjdk.java.net/jeps/349> (visited on 01/14/2021).
- [19] P. Goffin, J. Boy, W. Willett, and P. Isenberg. “An Exploratory Study of Word-Scale Graphics in Data-Rich Text Documents.” In: *IEEE transactions on visualization and computer graphics* 23.10 (2017), pp. 2275–2287. DOI: 10.1109/TVCG.2016.2618797.

- 
- [20] S. L. Graham, P. B. Kessler, and M. K. McKusick. "Gprof: A call graph execution profiler." In: *ACM SIGPLAN Notices* 17.6 (1982), pp. 120–126. ISSN: 0362-1340. DOI: 10.1145/872726.806987.
- [21] B. Gregg. "The flame graph." In: *Communications of the ACM* 59.6 (2016), pp. 48–57. ISSN: 0001-0782. DOI: 10.1145/2909476.
- [22] M. Grönlund and E. Gahlin. *JEP 328: Flight Recorder*. 2017. URL: <https://openjdk.java.net/jeps/328> (visited on 01/14/2021).
- [23] M. Harward, W. Irwin, and N. Churcher. "In Situ Software Visualisation." In: *2010 21st Australian Software Engineering Conference (ASWEC 2010)*. Ed. by C. Fidge. Piscataway, NJ: IEEE, 2010, pp. 171–180. ISBN: 978-1-4244-6475-3. DOI: 10.1109/ASWEC.2010.18.
- [24] J. Hoffswell, A. Satyanarayan, and J. Heer. "Augmenting Code with In Situ Visualizations to Aid Program Understanding." In: *Engage with CHI*. Ed. by R. Mandryk and M. Hancock. New York, New York: The Association for Computing Machinery, 2018, pp. 1–12. ISBN: 9781450356206. DOI: 10.1145/3173574.3174106.
- [25] *HotSpot Glossary of Terms*. URL: <https://openjdk.java.net/groups/hotspot/docs/HotSpotGlossary.html> (visited on 11/24/2020).
- [26] E. Juergens, D. Pagano, and A. Goeb. *Test Impact Analysis: Detecting Errors Early Despite Large, Long-Running Test Suites*. Whitepaper. CQSE GmbH, 2018.
- [27] B. Lee. "Adaptive Correction of Sampling Bias in Dynamic Call Graphs." In: *ACM Transactions on Architecture and Code Optimization* 12.4 (2016), pp. 1–24. ISSN: 1544-3566. DOI: 10.1145/2840806.
- [28] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. "Bug isolation via remote program sampling." In: *ACM SIGPLAN Notices* 38.5 (2003), pp. 141–154. ISSN: 0362-1340. DOI: 10.1145/780822.781148.
- [29] T. Lieber, J. R. Brandt, and R. C. Miller. "Addressing misconceptions about code with always-on programming visualizations." In: *CHI 2014, one of a CHIInd*. Ed. by M. Jones, P. Palanque, A. Schmidt, and T. Grossman. New York, NY: Assoc. for Computing Machinery, 2014, pp. 2481–2490. ISBN: 9781450324731. DOI: 10.1145/2556288.2557409.
- [30] D. Maplesden, K. von Randow, E. Tempero, J. Hosking, and J. Grundy. "Performance Analysis Using Subsuming Methods: An Industrial Case Study." In: *Second ACM International Conference on Mobile Software Engineering and Systems - MOBILESoft 2015*. Piscataway, NJ: IEEE, 2015, pp. 149–158. ISBN: 978-1-4799-1934-5. DOI: 10.1109/ICSE.2015.143.
- [31] D. Maplesden, E. Tempero, J. Hosking, and J. C. Grundy. "Subsuming Methods: Finding New Optimisation Opportunities in Object-Oriented Software." In: *ICPE '15*. Ed. by L. K. John, C. U. Smith, K. Sachs, and C. M. Lladó. New York, NY: ACM, 2015, pp. 175–186. ISBN: 9781450332484. DOI: 10.1145/2668930.2688040.
-

- [32] P. Moret, W. Binder, D. Ansaloni, and A. Villazon. "Visualizing Calling Context profiles with Ring Charts." In: *5th IEEE International Workshop on Visualizing Software for Understanding and Analysis, 2009*. Ed. by M. Lanza. Piscataway, NJ: IEEE, 2009, pp. 33–36. ISBN: 978-1-4244-5027-5. DOI: 10.1109/VISSOF.2009.5336425.
- [33] P. Moret, W. Binder, A. Villazón, D. Ansaloni, and A. Heydarnoori. "Visualizing and exploring profiles with calling context ring charts." In: *Software: Practice and Experience* (2010), n/a–n/a. ISSN: 00380644. DOI: 10.1002/spe.985.
- [34] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. "Evaluating the accuracy of Java profilers." In: *ACM SIGPLAN Notices* 45.6 (2010), pp. 187–197. ISSN: 0362-1340. DOI: 10.1145/1809028.1806618.
- [35] N. Nethercote and J. Seward. "Valgrind: a framework for heavyweight dynamic binary instrumentation." In: *ACM SIGPLAN Notices* 42.6 (2007), pp. 89–100. ISSN: 0362-1340. DOI: 10.1145/1273442.1250746.
- [36] Oracle, ed. *About Java Flight Recorder*. 2015. URL: <https://docs.oracle.com/javacomponents/jmc-5-5/jfr-runtime-guide/about.htm> (visited on 03/19/2021).
- [37] Oracle, ed. *Java Flight Recorder Command Reference: Release 10*. 2018. URL: <https://docs.oracle.com/javacomponents/jmc-5-5/jfr-command-reference/JFRCR.pdf> (visited on 03/05/2021).
- [38] Oracle, ed. *JVM(TM) Tool Interface 1.2.3*. URL: <https://docs.oracle.com/javase/8/docs/platform/jvmti/jvmti.html> (visited on 03/05/2021).
- [39] F. Paas and J. Sweller. "Implications of Cognitive Load Theory for Multimedia Learning." In: *The Cambridge handbook of multimedia learning*. Ed. by R. E. Mayer. Cambridge handbooks in psychology. Cambridge: Cambridge University Press, 2014, pp. 27–42. ISBN: 9781107035201.
- [40] C. Ponder and R. J. Fateman. "Inaccuracies in program profilers." In: *Software: Practice and Experience* 18.5 (1988), pp. 459–467. ISSN: 00380644. DOI: 10.1002/spe.4380180506.
- [41] A. Ragozin. *Lies, darn lies and sampling bias*. 2019. URL: <http://blog.ragozin.info/2019/03/lies-darn-lies-and-sampling-bias.html> (visited on 11/09/2020).
- [42] A. Ragozin. *Safepoints in HotSpot JVM*. 2012. URL: <http://blog.ragozin.info/2012/10/safepoints-in-hotspot-jvm.html> (visited on 11/24/2020).
- [43] D. Rothlisberger, M. Harry, W. Binder, P. Moret, D. Ansaloni, A. Villazon, and O. Nierstrasz. "Exploiting Dynamic Information in IDEs Improves Speed and Correctness of Software Maintenance Tasks." In: *IEEE Transactions on Software Engineering* 38.3 (2012), pp. 579–591. ISSN: 1939-3520. DOI: 10.1109/TSE.2011.42.



- 
- [44] D. Rothlisberger, M. Harry, A. Villazon, D. Ansaloni, W. Binder, O. Nierstrasz, and P. Moret. "Senseo: Enriching Eclipse's static source views with dynamic metrics." In: *IEEE International Conference on Software Maintenance*, 2009. Piscataway, NJ: IEEE, 2009, pp. 383–384. ISBN: 978-1-4244-4897-5. DOI: 10.1109/ICSM.2009.5306314.
  - [45] J. Rott, R. Niedermayr, E. Juergens, and D. Pagano. "Ticket Coverage: Putting Test Coverage into Context." In: *2017 IEEE/ACM 8th Workshop on Emerging Trends in Software Metrics - WETSoM 2017*. Piscataway, NJ: IEEE, 2017, pp. 2–8. ISBN: 978-1-5386-2807-2. DOI: 10.1109/WETSoM.2017.1.
  - [46] M. Serrano and X. Zhuang. "Building Approximate Calling Context from Partial Call Traces." In: *Proceedings of the 7th annual IEEEACM International Symposium on Code Generation and Optimization*. Washington, DC: IEEE Computer Society, 2009, pp. 221–230. ISBN: 978-0-7695-3576-0. DOI: 10.1109/CGO.2009.12.
  - [47] J. M. Spivey. "Fast, accurate call graph profiling." In: *Software: Practice and Experience* 34.3 (2004), pp. 249–264. ISSN: 00380644. DOI: 10.1002/spe.562.
  - [48] M. Sulăr and J. Poruban. *Exposing Runtime Information through Source Code Annotations*. 2017.
  - [49] M. Sulír, M. Bačíková, S. Chodarev, and J. Porubän. "Visual augmentation of source code editors: A systematic mapping study." In: *Journal of Visual Languages & Computing* 49 (2018), pp. 46–59. ISSN: 1045926X. DOI: 10.1016/j.jvlc.2018.10.001.
  - [50] M. Torre. [JDK-8223147] *JFR Backport*. 2019. URL: <https://bugs.openjdk.java.net/browse/JDK-8223147> (visited on 01/14/2021).
  - [51] J. Winter, M. Aniche, J. Cito, and A. van Deursen. "Monitoring-aware IDEs." In: *ESEC/FSE '19*. Ed. by M. Dumas. New York, NY: The Association for Computing Machinery, Inc, 2019, pp. 420–431. ISBN: 9781450355728. DOI: 10.1145/3338906.3338926.
  - [52] X. Zhuang, M. J. Serrano, H. W. Cain, and J.-D. Choi. "Accurate, efficient, and adaptive calling context profiling." In: *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*. Ed. by M. Schwartzbach and T. Ball. New York, NY: ACM, 2006, p. 263. ISBN: 1595933204. DOI: 10.1145/1133981.1134012.