In Situ Understanding of Performance Bottlenecks through Visually Augmented Code

Fabian Beck
VISUS, University of Stuttgart
Stuttgart, Germany

Oliver Moseler and Stephan Diehl
University of Trier
Trier, Germany

Günter Daniel Rey
FernUniversität in Hagen
Hagen, Germany

Abstract—Finding and fixing performance bottlenecks requires sound knowledge of the program that is to be optimized. In this paper, we propose an approach for presenting performance-related information to software engineers by visually augmenting source code shown in an editor. Small diagrams at each method declaration and method call visualize the propagation of runtime consumption through the program as well as the interplay of threads in parallelized programs. Advantages of in situ visualization like this over traditional representations, where code and profiling information are shown in different places, promise to be the prevention of a split-attention effect caused by multiple views; information is presented where required, which supports understanding and navigation. We implemented the approach as an IDE plug-in and tested it in a user study with four developers improving the performance of their own programs. The user study provides insights into the process of understanding performance bottlenecks with our approach.

I. INTRODUCTION

Optimizing the performance of a program requires identifying performance issues and understanding the code connected to those issues. For example, a sorting algorithm might consume a considerable amount of computing time because of a bug making the implementation inefficient or just because large datasets have to be handled—code and context have to be understood to decide between the two scenarios. Hence, performance optimization is closely connected to program comprehension.

A first step towards optimizing the runtime of a program is measuring the status quo—recording the execution times of different parts of the program is called profiling (e.g., [1], [2]): either the code is instrumented to precisely measure the times or a heuristic approach is applied such as repeatedly sampling the call stack of the running program. The collected runtime information needs to be assessed by the developers and provides them with fundamental information for deciding how to enhance the performance of their software.

The central idea of this paper is reflecting profiling information into code—both have to be brought together for enabling developers to understand a performance bottleneck and to finally improve the runtime behavior of the program. Applying a form of in situ visualization [3], small visual elements augment method declarations in the code editor by runtime information (Figure 1): these visualizations show the estimated execution time of the method as well as the number and kind of threads executing the method. On demand, details such as the propagation of execution times through incoming and outgoing method calls can be retrieved. We implemented this approach as an Eclipse plug-in and tested it in realistic scenarios: users were able to quickly identify and understand performance bottlenecks; fixes and enhancements applied to the code often directly led to considerable performance improvements. The difference of our approach to existing similar profiling visualizations for IDEs such as Visual Studio and Eclipse is the richness of the in situ visualization as well as that the approach does not require any additional views or sidebars.

Fig. 1. Visually augmenting method declarations with performance data.

The remainder of this paper is structured as follows: In Section II, the propagation of runtime consumption of methods is discussed in detail based on a formal model of call trees. Section III presents the in situ visualization approach, which is depicted as a preview in Figure 1. Further, the prototype implementation of the approach as an Eclipse plug-in is documented in Section IV. The tool was tested in an explorative and formative user study, Section V reporting the results. More context is provided by discussing related work in Section VI. Finally, Section VII concludes the paper.

II. METHOD RUNTIME CONSUMPTION

Running a performance test, the total runtime consumption can be easily recorded. If the developers know which part of the software consumes most of it, they might be able to improve the respective code and make the software more efficient. For non-trivial programs, it is, however, often not clear which parts are most critical—runtime consumption needs to be recorded and presented on a finer level of granularity such as method level.

A. Profiling Perspectives

Analyzing the runtime consumption of a single method, two perspectives might be taken as illustrated in Figure 2:
Fig. 2. Illustrating a black-box and a white-box perspective on the runtime consumption of a method.

- **Black Box**—Observing a method as a black box, only two kinds of events are relevant: entering the method and returning from the method call; the timespan between the two events provides the respective runtime information necessary for executing a single method call. Runtime for multiple calls of a method can be aggregated, which we denote as *method time*.

- **White Box**—When looking into a method, further aspects become relevant: first, the statements directly executed in the method consume a certain amount of time, which is called *self time* of the method; second, the method usually calls other methods and waits for their completion, which is described by a set of *invocation times* of the method to other methods.

These two perspectives are not mutually exclusive, but rather complement each other: while a high total time points to a possible problematic method, the self time and invocation times explain whether the runtime is consumed in the particular method or propagated through method calls.

**B. A Call Tree Model**

We developed a formal model of a call tree, that is a hierarchical structure representing the invocation of methods as nodes. The first invocation of the main method forms the root node, methods called by the main method are inserted as child nodes, etc. Hence, if a method is called multiple times (e.g., as part of a recursion), each invocation is represented by a different node—otherwise, the call tree would become a call graph where the precise execution path is not preserved. Each node of the tree (i.e., each method invocation) can be assigned an entry time and a return time.

As a basis for the model, let $\mathcal{M}$ be the set of all method signatures and $\mathcal{L}$ be a set of all locations in source code, then $\mathcal{I} = \mathcal{M} \times \mathcal{L} \times \mathbb{R} \times \mathbb{R}$ is the set of method invocations. Here, a tuple $i = (m, l, t_1, t_2) \in \mathcal{I}$ represents the invocation of a method $m$ at location $l$ starting at time $t_1$ and returning at time $t_2$. We use the following auxiliary notations: $\text{meth}(i) = m$, $\text{loc}(i) = l$, $\text{start}(i) = t_1$, $\text{end}(i) = t_2$, $||i|| = \text{end}(i) - \text{start}(i)$; $i' \subseteq i$ if $\text{start}(i') \leq \text{start}(i) \land \text{end}(i') \leq \text{end}(i)$.

Next, we construct a call tree $ct = (i, CT)$, which consist of an invocation $i$ and a set of subtrees $CT$, which themselves are call trees. The function $\text{inv}(ct) = i$ and $\text{subtrees}(ct) = CT$ yield the respective components. As a shorthand notation, we extend the above auxiliary notation for invocations to call trees, such that they return the corresponding value of the first component of a call tree, e.g., $\text{meth}(ct) = \text{meth}(\text{inv}(ct))$. We define the set of *valid call trees* $CT$ by induction:

- For $i \in \mathcal{I}$, $(i, \emptyset) \in CT$
- For $i \in \mathcal{I}$ and $ct_1, \ldots, ct_k \in CT$,
  $$(i, \{ct_1, \ldots, ct_k\}) \in CT$$
  if $\text{start}(i) \leq \text{start}(ct_1) \land \text{end}(ct_k) \leq \text{end}(i) \land \forall 1 \leq j < k \text{ \text{end}(ct_j) \leq \text{start}(ct_{j+1})}$.

An example for a call tree is shown in Figure 3 (top). The root node $a$ represents the invocation of the main method of the program, which calls method $b$ three times, represented by separate nodes. Please note that the second and third invocation of $b$ by $a$ triggers a direct (second call) or indirect (third call) recursion of $b$. The number annotating an invocation $i$ provides the respective execution time $||i||$.

For multi-threaded programs, each executed instance of a thread creates a call tree. Since the current application does not rely on the information which threads are executed in parallel, we introduce the following simplification: call trees of the multiple threads are concatenated into a single tree. To this end, a dummy invocation $i_c$ is created and the concatenated call tree $cct = (i_c, \{ct_1, \ldots, ct_k\})$ is assembled from the call trees of the threads $ct_1, \ldots, ct_k$. In order to build a valid call tree and making the time frames of different threads free of overlap, start and end times of all invocations need to be adapted so that $\text{start}(ct_1) = 0 \land \text{end}(ct_k) = \tau \land \forall 1 \leq j < k \text{ \text{end}(ct_j) = \text{start}(ct_{j+1})}$ without, however, changing the execution times $||i||$ for any invocation $i$. Hence, $\tau$ denotes the total time of the program, which sums up the runtime of all threads.
C. Aggregated Execution Times

The call tree model forms the foundation for defining appropriate metrics to assess the black-box and white-box runtime of methods. Preparing these definitions, we introduce the following auxiliary constructs: The set of method invocations included in a call tree \( ct \) is collected by the recursively defined function

\[
I(\text{ct}) = \{\text{inv}(\text{ct})\} \cup \bigcup_{\text{ct}' \in \text{subtrees}(\text{ct})} I(\text{ct}')
\]

these invocations can be restricted to a specific method \( m \) by

\[
I(m, \text{ct}) = \{i \mid i \in I(\text{ct}) \land \text{meth}(i) = m\}
\]

The duration of a set of invocations \( I \subset \mathcal{I} \) is

\[
\|I\| = \sum_{i \in I} \|i\|
\]

The metric implementing the black-box perspective aggregates the runtime that a particular method \( m \) is active in a call tree \( ct \). Ignoring recursion, this is the sum of all execution times of the method \( \|I(m, \text{ct})\| \). However, in case of recursion, we consider certain amounts of runtime multiple times applying this simple approach. Instead, we have to filter out those invocations of the method that are (directly or indirectly) called by the same method in order to exclude overlapping time frames. Filtering a general set of invocations \( I \subset \mathcal{I} \), function

\[
F_{\max}(I) = \{i \mid 3b' \in I : i' \neq i \land i \sqsubseteq i'\}
\]

excludes those invocations already contained in other invocations of the set. Finally, the method time is the sum of durations of the filtered invocations of \( m \):

\[
\tau_{\text{meth}}(m, \text{ct}) = \|F_{\max}(I(m, \text{ct}))\|
\]

In the example of Figure 3 (bottom, left), the method time is evaluated for method \( b \)—the applied filtering leads to skipping recursively called instances of \( b \).

Taking the white-box perspective, it is of interest how much time is spent for invoking a particular method \( m' \) from method \( m \). To this end, the invocations need to be restricted further such that only invocations of method \( m' \) triggered by \( m \) are contained in the resulting set:

\[
I(m, m', \text{ct}) = \{i \mid i \in I(\text{ct}) \land \text{meth}(i) = m' \land \text{loc}(i) \in m\}
\]

In other words, the source code location \( \text{loc}(i) \) of a call to \( m' \) has to be part of method \( m \). Then, the invocation time is defined as

\[
\tau_{\text{inv}}(m, m', \text{ct}) = \|F_{\max}(I(m, m', \text{ct}))\|
\]

Figure 3 (bottom, right) provides an example for method \( b \) recursively calling itself; filtering again saves from considering certain amounts of runtime twice.

Discerning the invocation time further by location of invocation (i.e., the source of the invocation, which is identified by a position in code), the filtering

\[
I(l, m, \text{ct}) = \{i \mid i \in I(\text{ct}) \land \text{meth}(i) = m \land \text{loc}(i) = l\}
\]

can be used to define the location invocation time

\[
\tau_{\text{loc}}(l, m, \text{ct}) = \|F_{\max}(I(l, m, \text{ct}))\|
\]

Finally, the self time of a method \( m \) represents the time that the method uses itself for executing its statements, but does not spend waiting for returning calls. It can be computed by subtracting the runtime of all direct invocations triggered in \( m \) from the method time of \( m \):

\[
\tau_{\text{self}}(m, \text{ct}) = \|I(m, \text{ct})\| - \sum_{m' \in \mathcal{M}} \|I(m, m', \text{ct})\|
\]

Filtering does not need to be applied in this case because self times of method calls do not overlap by definition.

III. In Situ Visualization of Performance Data

Inspired by the increasing popularity of sparklines [4]—small word-sized graphics embedded in tables or texts like those used in Microsoft Excel—we started exploring the usage of these miniaturized visualizations for augmenting source code with additional valuable information. While Harward et al. [3] already presented an approach for visualizing software metrics in the source code editor using simple color coding, we try to elaborate on this idea of in situ software visualization by exploring other areas of application and more complex visualizations. Analyzing performance data seems to be an ideal application because it both provides complex information as well as it requires the comprehension of specific parts of the source code. In particular, our objectives for an in situ visualization of performance data were

- to display the consumed runtime for each method,
- to show the threads accessing a method and the respective computation costs they produce, and
- to reveal how the consumed runtime is distributed among the different callers and callees of a method.

While the most important information or a preview of the information should be directly visualized at each method declaration or in the method bodies in situ, detailed information can be available on demand.

A. Psychological Motivation

Profiling information could be presented to program code either by in situ visualizations or by multiple representations. Psychological research has identified numerous difficulties for users and learners using multiple representations [5], [6], [7], [8]. For example, mental connections between different multiple representations do not occur spontaneously [9]. But most importantly, a split-attention effect may arise, i.e., an effect that impairs information processing by using formats that require persons to split their attention between multiple sources of information, where each source of information is essential for understanding the information [10]. This effect postulated in the cognitive load theory [11] assumes that the split of the users’ attention can lead to extraneous cognitive load (i.e., a working memory load affected by the manner in which information are presented and which can reduce
public EntitySet getIncludedEntities()
{
    EntitySet entitySet = new EntitySet();
    entitySet.addEntitySet(directlyIncludedEntities);
    for (Cluster cluster : subClusters)
    {
        EntitySet includedEntitySet = cluster.getIncludedEntities();
        entitySet.addEntitySet(includedEntitySet);
    }
    return entitySet;
}

self time \( \tau_{self}(m, cct) \) / \( \tau_{method} \)
callee invocation time \( \tau_{inv}(m', m, cct) \) / \( \tau_{method} \)
method time \( \tau_{method} \)
invocation time \( \tau_{inv}(m, m', cct) \) / \( \tau_{method} \)
self time \( \tau_{self}(m, cct) \) / \( \tau_{method} \)

Fig. 4. In situ visualization approach enriching the source code by performance data an providing additional information on demand in tooltips; formulas refer to the call tree model defined in Section II-B.

cognitive resources for information processing) [12]. In contrast to multiple representations, in situ visualizations integrate different representations physically and, therefore, avoid a split-attention effect. Therefore, extraneous cognitive load is reduced and cognitive resources for information processing is released, resulting in improved user’s information processing [10], [13].

B. Method Time and Self Time

Visualizing profiling information in the source code view requires mapping the metric measures defined in Section II-C to the entities that can be found in the code. Since all introduced metrics attribute certain runtime consumption to aggregated invocations of methods, small visualizations are attached to each declaration of a method and calls of methods.

Most central metric is the method time \( \tau_{method} = \tau_{method}(m, cct) \), which summarizes all invocation of the same method \( m \) for a recorded concatenated call tree \( cct \in CT \). This metric can be easily matched with the method declaration of \( m \) and visualized next to it: As shown in Figure 4 (top) for a method \( \text{getIncludedEntities}() \), a colored box is added at the end of the line containing the declaration. The box provides the method time in relation to the total time \( \tau \) of the program as a percentage value in numbers. Additionally, this information is redundantly encoded in the background color of the box using a color scale from bright green to dark red (Figure 4, right).

What is more, the self time \( \tau_{self}(m, cct) \) of method \( m \) is visualized as a striped bar filling the box according to its relative size in contrast to the method time \( \tau_{method} \). The example in Figure 4 shows a low percentage of self time. Moreover, details on the self time are provided in the header of the tooltip that appears moving the mouse over the visualization as depicted in Figure 4 (bottom, left).

C. Callers and Callees

The method time provides a first impression (black-box perspective), but the core of the approach is also visualizing the propagation of runtime through method calls (white-box perspective). Each method can be invoked by a set of other methods (callers) and itself might also call further methods (callees). Considering methods as a nodes, method invocations hence form edges of a graph. This graph is a partly aggregated representation of the introduced call tree. The invocation time \( \tau_{inv}(m, m', cct) \) can be assigned to an edge from \( m \) to \( m' \).

Edges in graph visualizations are usually drawn as graphical links. This, however, is not feasible in the current scenario because source and target of the edge (i.e., the two methods) are usually not visible on the same screen. Instead, we decided to work with different forms of adjacency list representations for visualizing consumed time of specific method invocations as described in the following.

1) Overview: First, a preview of incoming calls and outgoing calls is already depicted in the miniature visualizations attached to the method declarations. Small arrows on the left indicate invocations of the method by other methods (callers), small arrows on the right represent respective outgoing calls (callees). For simplification, only three states are discerned: no arrow \( \rightarrow \) no calls; single arrow \( \rightarrow \) single call, and three arrows \( \rightarrow \) multiple calls. Figure 4 shows an example with multiple callers and multiple callees; further instances with other properties can be found in Table I.

Additionally, a detailed list of all callee and callers of a method is provided in the tooltip of the method time
visualization: callers of the particular method are listed on the left, which is an adjacency list of incoming edges, callees of the method are provided on the right, which forms an analogous list of outgoing edges. For each calling or called method, a relative invocation time is reported. In case of incoming calls from a method $m'$ to method $m$, that is the invocation time $\tau_{m'}(l' \in m', m, cct)$ in relation to the method time of $m$. Analogously, the invocation time consumed by outgoing calls $\tau_m(m, m'', cct)$ to a method $m''$ is also contrasted to the method time. Both lists are ordered by runtime consumption in decreasing order.

2) In Situ Callee Visualization: While the adjacency lists contained in the tooltip make the time propagation through method calls accessible, it still requires some interaction to retrieve this information. Applying the in situ visualization paradigm, similar information can be also represented in the code—visualizations of the respective invocation time are attached to each call of another method in the body of a method. To express their relation to the visual representation of the method time, those visualizations are designed similarly to the ones augmenting the method declaration: a small box provides the runtime as the percentage of the location invocation time $\tau_m(l, m'', cct)$ to the method time where $m''$ is the called method and $l$ is the specific location in $m$. A colored bar additionally visualizes this percentage in the background of the box; the color of the bar is the same as used for the background of the method time visualization of the enclosing method.

The representations of called methods in the tooltip and in the code complement each other: both are forms of adjacency lists, but while the one in the tooltip provides a compact and ordered representation of all callees, the representation in the code puts the consumed runtime into the direct context of the calls. Another difference is that multiple lines calling the same method are aggregated in the tooltip, but discerned for different locations in the code. In contrast to visualizing outgoing calls, it is not possible to embed analogous visualizations for incoming calls into the code because those calls do not have a representation in the declaration or body of a method.

3) Recursive Calls: For non-recursively used methods (neither direct nor indirect recursion), the self time plus the callee invocation times of a method add up to the method time. In case of recursion, however, this equation does not hold: different outgoing calls might be active at the same time when multiple instances of the method are active concurrently. For instance, the method getIncludedEntities() as depicted in Figure 4 includes recursion: the sum of all callee percentage plus respective self time is clearly more than 100% of method time. Already the recursive calls of getIncludedEntities() cover over 60% of the method time, which means that in about two thirds of the method time the direct recursion is active. Nevertheless, a single callee invocation time is never larger than 100% because it cannot be active longer than the calling method. Analogous statements can be made for the caller invocation time, except that self time does not need to be considered.

D. Multiple Threads

Next to each box visualizing the method runtime, the user finds another diagram. It represents the threads that executed the method where each small square stands in for an instance of a thread; the color discerns the type of the thread (i.e., the class the thread is an instance of). In the example of Figure 4 (top), two threads accessed method getIncludedEntities(), both having different types. The details on types and distribution of runtime consumption of the threads can be retrieved as part of a tooltip that appears on demand when the user is hovering the thread visualization as shown in Figure 4 (bottom, right). In particular, the method time $\tau_{meth}(m, ct_j)$ restricted to call tree $ct_j$ of a specific thread is contrasted to the non-restricted method time $\tau_{meth}$. The threads are listing together with resulting percentage values in the tooltip grouped by thread type; all thread method times add up to 100%.

E. Examples

Table I lists a collection of methods and attached visualizations of a ray-tracing software (the program is also part of the user study: User 2, parallelized version). They show the diversity in runtime characteristics of methods that can be quickly retrieved by just browsing through the code. While the color coding allows the users to scan for significant runtime consumption such as in methods hitObject() and lighting(), the percentage values tell the precise numbers. Self time as well as caller/callee visualizations give a first impression on how the runtime consumption is propagated through this method: for instance, method main() act as a starting point as it does not consume considerable time itself, has no incoming and only one outgoing call; in contrast, method cross() is a kind of end point because hit has no outgoing calls and hence consumes all runtime itself. Method hitObject() can be directly identified as a critical method because of its high method time which is consumed in the method to a large extent (high self time); several incoming and outgoing calls hint at the centrality of the method. Multiple threads executing the method show that it is already parallelized.

### Table I

<table>
<thead>
<tr>
<th>method</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>main()</td>
<td>no callers, one callee, no considerable self time</td>
</tr>
<tr>
<td>paint()</td>
<td>one caller, one callee, some self time</td>
</tr>
<tr>
<td>lighting()</td>
<td>multiple callees, four threads of the same type</td>
</tr>
<tr>
<td>hitObject()</td>
<td>high method time, multiple callers and callees, high self time</td>
</tr>
<tr>
<td>cross()</td>
<td>low method time, no callees, only self time</td>
</tr>
</tbody>
</table>
F. Extensions

The core approach as presented so far specifies the visualization embedded into the code, but it can be extended into different directions.

1) Overview Panel: The in situ visualization makes profiling information available for the current context. But sometimes it is important to first gain an overview of the whole runtime consumption before going into the details. In this case, an overview representation is required. This can be a list of hot methods (those methods having the highest self time) as it is usually provided by profiling tools. But also the already available general views of an IDE can be employed for presenting an overview. For instance in Eclipse, the package explorer (organization of all projects and packages in the workspace) or the outline view of a class (list of fields and methods of a class) can be leveraged. This idea is sketched for an outline view in Figure 5. Overview visualizations like this might be important for selecting a starting point for the analysis or for comparing absolute runtime consumption across different subsystems. Probably, they are particularly relevant for developers less familiar with the code.

2) Comparison to a Reference: Another application, only partly supported by the core approach, is observing changes in the runtime consumption after applying a fix. When trying to optimize the performance of a program, often small changes are made, the program is executed again, and the effect is analyzed—we observed this procedure frequently in the conducted user study (Section V). Supporting this use case by visualization, the runtime consumption might be recorded for the analysis or for comparing absolute runtime consumption across different subsystems. Probably, they are particularly relevant for developers less familiar with the code.

The core approach as presented so far specifies the visualization embedded into the code, but it can be extended into different directions.

IV. Tool Implementation

The in situ visual profiling approach has been implemented as a plug-in of the Eclipse IDE (version: Indigo) for profiling Java applications. It includes the visualization technique and interactive features as described, except the suggested extensions, which are not yet implemented.

The proposed approach does not depend on a specific profiling technique, but is open for any technique supporting the collection of runtime information required for the metrics defined in Section II-C. Since instrumenting the code and measuring the precise runtime of methods is slowing down execution too much, the tool implements stack trace sampling, that is, a lightweight heuristic repeatedly retrieving the stack trace of the running software. Random intermediate times between the samples were applied in order to prevent biases [2]; an average sampling rate of 10 ms (normal distribution, standard deviation: 5 ms) is set as a default—sampling only increases the runtime of the profiled program moderately (factor 1.16 to 2.17 in the user study, see Table II). Though being estimated, the values measured with this heuristic approach proved to be good enough for finding relevant performance bottlenecks. The concrete stack traces were retrieved through the Java Debug Interface (JDI), which is the highest layer of the Java Platform Debug Architecture (JPDA).

The result of the sampling process are lists of stack traces of all threads running at sampling time. For each trace, a fixed runtime divided by the number of active threads is assigned to each method in the trace for computing the method time. The method on the stack that has been called most recently (i.e., is at the top of the stack) is the currently active method and additionally receives the same amount of time as self time. For each pair of subsequent methods on the stack, the time is counted as invocation time; location invocation times are handled alike.

Profiling a specific program, it only needs to be started setting a few parameters for the virtual machine. Our Eclipse plug-in connects to the machine and records stack traces as described above. When the execution of the program is completed, the performance visualization are activated and deactivated by pressing a button. The visualizations appear in the standard Eclipse editor without interfering with other features of the editor. They are implemented by using functionality of the class StyledText, which is a basic class of the editor.
V. USER STUDY

We performed an explorative user study [15] in order to increase our general understanding on how and in which scenarios the proposed visualization approach is applicable. We conducted the study in parallel to the implementation of the tool and thus were able to immediately feed back preliminary results into the development process (formative study): while User 1 only had some core functionality available, User 4 already worked with the nearly finished tool (only self times were not yet visualized as striped bars and callers/callees were not yet additionally indicated by arrows). We started off with two of the co-authors of this paper applying the tool (User 1 and User 2), and then additionally asked two external developers for participating (User 3 and User 4).

As it is the first study testing the approach, the focus was on the general question whether and how the approach can be used for improving the performance of existing software. To evaluate the approach in realistic scenarios, all developers tried to optimize a software system that they originally developed themselves. While gained performance improvements (Table II) were a positive side effect, we were mainly interested in questions of applicability, strategies to use the approach, problems of usability and intuitiveness, as well as direct feedback.

A. Setting

As a preparation of each experiment, a performance test—if not already available—was created for each of the investigated programs. This test was repeatedly run throughout the experiment for collecting current performance information and served as a benchmark for improving the program. In particular, the performance test was executed at the beginning as well as at the end of each experiment, in each case once with deactivated profiling, once with activated profiling. As reported in Table II, the initial tests consumed between 3 and 20 seconds without profiling (3 and 27 seconds with profiling), which is fast enough for frequent repetition. To further increase their reliability, they were repeated three times each, and only the median result (the second among the three) is reported. All experiments were conducted on a Windows 7 PC with 6 GB of RAM and an Intel Core i7 M 620 processor having 2 cores (4 threads); the IDE was shown on a 24” full HD monitor.

In the main part of the experiment, the users tried to optimize the performance of their software with help of our approach. They were neither restricted in the type of changes they could apply nor in the use of other resources such as functionality of the IDE, documentation, or web content. The experiment was led by an experimenter (one of the authors), who introduced the tool at the beginning, took notes, and occasionally helped the users. The users were asked to speak out their thoughts aloud and to comment their actions (thinking-aloud experiment); voice and screen were recorded. The experiment ended when the users were satisfied with their changes or did not see room for further performance improvements that could be implemented with reasonable effort.

Table II lists the time savings contrasting the performance at the beginning of the study (pre) to the performance at the end (post).

| Software: information visualization tool for graph and hierarchy comparison (parallelized) |  |  |
| Benchmark: drawing the visualization 100 times for a sample dataset |  |  |
| Strategy: started with main method, followed invocations, directly jumped to interesting locations |  |  |
| Bottlenecks: unnecessary consistency checks removed (small improvements); assemble lists of graph edges switching to non-smooth drawing (considerable improvements) |

After the main part, participant and experimenter discussed whether it could have been possible to find the same bottlenecks and implementing equivalent improvements only with the help of a traditional profiling approach providing a list of hot methods (high self time). As a foundation for the discussion, the respective performance benchmark was profiled using VisualVM, which is the standard profiling tool shipped with the Java Development Kit (JDK) and outputs a sortable list of hot methods.

B. Usage Stories

Fitting the explorative approach of the conducted evaluation, the results of the experiments are documented in form of usage stories noted in shorthand. First, we provide the length of the main part of the experiment, which ranges from 82 to 126 minutes. Further, we briefly describe the software the participants optimized, provide some information on the respective benchmarks, and report the strategies the users applied working with our in situ approach. But most importantly, the identified bottlenecks are described and how the users further tried to fix them; we also report whether the fix was successful whereas success is determined by improvement of runtime. Additional to the usage stories, Table II contrasts the runtime of benchmarks before the experiment to results after optimizing the programs (with and without profiling)—in three of four cases, performance could be improved reasonably.

User 1 (82 min, co-author) – Software: information visualization tool for graph and hierarchy comparison (parallelized) – Benchmark: drawing the visualization 100 times for a sample dataset – Strategy: started with main method, followed invocations, directly jumped to interesting locations – Bottlenecks: unnecessary consistency checks removed (small improvements); assemble lists of graph edges switching to non-smooth drawing (considerable improvements).

User 2 (112 min, co-author) – Software: ray tracing engine for education (non-parallelized) – Benchmark: rendering a sample scene – Strategy: started with main method, browsed through the editor searching for relevant methods, followed invocations – Bottlenecks: expensive debug message though debugging was deactivated removed debugging message (considerable improvement); hitObject() consumes nearly all relevant runtime -> tried to optimize hitObject(), e.g., by testing different data structures (no improvements); parallelized ray tracing (considerable improvements after several tries).
**User 3** (92 min) – **Software**: test case generator (non-parallelized) – **Benchmark**: generate a collection of test cases – **Strategy**: started with main method, systematically followed invocations, bookmarked bottlenecks for later improvement – **Bottlenecks**: most runtime is consumed by library method selecting nodes in an XML tree → introduce caching mechanism (not implemented because too much effort for the experiment); delete directories on disk → replaced own implementation by library method (no improvements); parsing a complete XML file for retrieving a single string → implemented a lightweight special-purpose parser (no improvements).

**User 4** (126 min) – **Software**: code clone detection (parallelized) – **Benchmark**: code clone detection in the JFreeChart project (587 classes and interfaces) – **Strategy**: started with main method, followed invocations, directly jumped to interesting locations – **Bottlenecks**: misconfiguration of parallelization → fixed configuration (no improvements); waiting unnecessary long for the completion of threads → notification upon completion (small improvements); inefficient loop iterator → replaced iterator by direct access (small improvements); parser consumes much runtime → tried to optimize parser configuration (no improvements); concurrently adding elements to a large set → tried different implementations of sets and optimized synchronization (small improvements).

The discussion comparing our approach to a traditional approach further showed that a list of hot methods often also contains the relevant information, but context is completely missing: making sense of the information is much harder because the necessary code and flow of control first has to be manually retrieved. For instance, the crucial performance issue fixed by User 1 was also listed as a hot method, but would have looked unsuspicous because of missing context information in form of source code, which was needed for finding that smooth drawing was applied unnecessarily. Moreover, methods from different parts of the program and libraries are mixed together in a list of hot methods. In case of User 3, the critical runtime consumption was hidden deeply in a library—just having a list of hot method does not tell why the particular library method is called. In one case (User 4), however, analyzing the list of hot methods revealed a further bottleneck, that was not observed using the in situ approach: a specific `hashCode()` method (required for comparing objects) consumed reasonable runtime; it was not noticed before because the method is called from the Java API and not from user-developed code—in situ visualizations can only be depicted for available source code.

### C. Discussion

The usage stories show that the approach supports identifying and understanding bottlenecks in various applications: Each participant located several performance issues and was able to retrieve how they arise, why they consume as much runtime, and whether this consumption is necessary. While all participants tried to optimize the code of the program, three participants managed to tune their software considerably. Some of the diagnosed issues probably would have also been found with a traditional profiling approach, but the comparison to VisualVM suggests that, in lists or trees of methods, it is harder to discern necessarily consumed runtime from wasted runtime. Moreover, the in situ approach seems to help navigating through the methods since the code always provides supportive context and methods are not represented multiple times as in call tree representations. Nevertheless, in some situations such as analyzing library calls, a traditional overview representation could have been helpful.

Analyzing the usage stories in detail, we identify high-level usage concepts. In general, three types of activities can be discerned as discussed in the following. While one step has to follow the other in the order listed, these are not clear phases of the experiment, but were applied in iteration by the participants, often in quick repetition.

1) **Identifying Bottlenecks**: The first activity is, of course, searching for runtime issues in the program. All participants started with the main method (or another central method) and followed the invocation of methods, either in order of execution or primarily focusing on those invocations that consume most runtime. Two of the four participants also occasionally jumped to locations that they suspect as being problematic based on prior knowledge. One participant used the bookmarking functionality of the IDE to mark suspicious locations in the code for later analysis—many developers, however, does not seem to be aware of this functionality or at least rarely use it [16]. All participants, no matter which specific strategy they applied, quickly found performance issues in their code; the in situ visualization guided the search.

2) **Understanding Bottlenecks**: When a performance bottleneck is identified, it is required to analyze whether the issue is a necessary computation that can hardly be optimized, whether it is non-optimized but useful code, or whether it is unnecessary complex or even totally superfluous. Although the participants worked with their own software, they needed to scrutinize the source code in nearly all cases for answering this question. While some of the issues were expected to consume reasonable runtime and could be easily explained, others were unexpected for the participants and caused further detailed analyses. Especially for unexpected bottlenecks, the close connection of visualization and code seems to be beneficial: users were able to quickly jump between methods tracking down the specific problem while always having the necessary performance information available. In case of recursive calls or methods called by several other methods, the listing of callees often helped understanding the concrete flow of control.

3) **Fixing Bottlenecks**: In some cases, usually applying quick fixes, participants directly tried to resolve an identified and analyzed issue. By using bookmarks or taking (mental) notes, this step can be postponed alternatively until sufficient overview of performance issues is gained. In either case, the visualization provided immediate feedback whether the modifications were successful and how they changed the runtime behavior (without difference profiling, the user, however, has to remember the results of the previous run). Often, the applied changes were not successful in the first instance—in these cases, the visualization helped detecting the freshly introduced problems. For instance, User 4 experimented with different set implementations and found that improving the runtime of
the add operation equally increases the cost of the also used remove operation.

D. Threats to Validity

Reflecting the explorative and formative character of the user study, collected evidence can only be considered preliminary. In particular, the internal validity might be affected: Since no control group was provided, it cannot be conclusively rejected that similar or even better performance improvements could have been reached without applying our approach; using a traditional approach was only discussed hypothetically. Further, the small number of users does not sufficiently absorb random effects; two of the users being co-authors of the paper might have biased the results.

Despite the limited number of users, however, a certain external validity is provided: First, the tested application scenarios are realistic as users tried to optimize their own software and were not considerably restricted in the use of other resources. Second, the approach is successfully applied to very different programs suggesting a certain generalizability. Nevertheless, the external validity is limited by the still somewhat artificial scenarios, the small set of tested scenarios, the small sizes of the optimized programs, and the focus on Java systems.

VI. RELATED WORKS AND TOOLS

In general, a variety of profiling techniques exists, which can be divided into static techniques just observing the program and dynamic techniques instrumenting the executable code [17]—the sampling technique applied in this paper is static. The influential profiling tool gprof [1] collects similar profiling information on method level as our approach: self times and time propagation through method calls; in contrast, however, recursive calls are collapsed. The outputs of gprof are two lists, one providing self times, the other showing time propagation through calls for each method. The authors describe the latter as "a window into the call graph". Our approach combines both representations and visualizes the information in the code view.

Profiling program performance seems to be established task among software developers as there exists a multitude of tools for supporting this task. In context of Java, the VisualVM, which is integrated into the Java Development Kit (JDK), supports the profiling of method calls: on the one hand, the developers can inspect a list of methods sortable by self time; on the other hand, method times and time propagation is accessible through call trees, which can be expanded interactively. Similar views are integrated in other profiling tools such as JetBrains dotTrace, SmartBear AQTime, or Intel VTune. The representations provide overview and list all relevant information. But even if they are connected interactively to a source code view, they do not integrated code and profiling information leading to split-attention effects. Moreover, the context the developer currently is in due to analyzing a specific piece of code is not considered by the additional views—the users always have to reapply it by searching.

But some of the listed tools already provide a source code view enriched with basic profiling information in a sidebar. Moreover, Waddell and Ashley [14] present an advanced code view encoding execution frequencies in the background color of lines. Recently, heatmap-based approaches like this have been implemented for IDEs: With their 2012 release of Visual Studio, Microsoft introduced a performance analyzer that visualizes method times and time propagation similar to our approach; the code is overlaid with a heatmap showing the runtime consumption, a side bar provides the precise values, and an extra view visualizes the time propagation (callers and callees) for a currently selected method. Also the Performance Source Viewer integrated into IBM Rational Developer provides color-coded runtime consumption information in a sidebar for each line of code. Röthlisberger et al. [18] extend an IDE to showing invocation frequencies in a sidebar and further dynamic callee information in tooltips. While these tools head into the same direction as our approach, they are not going as far: though not only showing lists of methods, they still require additional views and sidebars for presenting the performance information; the in situ visualization that they use is restricted to heatmaps.

Generally speaking, the source code editors of IDEs have become more and more sophisticated over the last years. Specifically, the code is getting visually augmented with additional information like errors, warnings, breakpoints, search results, etc. The idea of displaying additional useful information in the code goes back to the Seesoft approach [19] from the early 90s, which encodes various software metrics in the line color of the code, among them execution counts of lines [20]. Since then, specialized tools have been developed based on this idea, for instance, for code fault localization [21] or program comprehension [22]. Further, Harward et al. [3] propose an IDE framework for in mapping software metrics to different visual features of source code, such as the colors of the line background or margin. IDE editors have also been enriched with further information like caller and callee relations, for example, for improving code navigation [23]. New IDE concepts like Code Bubbles [24] or Code Canvas [25] have been proposed, which also integrate diverse visual cues.

The execution of a program can as well be visualized as a dynamic graph, each weighted edge of the graph encoding the call of a method and the required time the execution takes [26]. Depicting those call graphs aggregated over time for a selection of methods [27] or a selection of time [28] is somewhat similar to our approach as the in situ visualization also shows a part of this graph. Further, call trees can be visualized for exploring runtime behavior, for instance, as radial icicle plots [29], [30]. For enriching these graph and hierarchy visualizations with profiling information, nodes representing methods are scaled according to the runtime they consume [27], [30]. When using a timeline instead of aggregating time, method calls or other messages between software artifacts create a form of sequence diagram [31], [32]. On a higher level of abstraction, thread activity or CPU core utilization is also visualized on timelines [33], [34].
VII. CONCLUSION

We presented an in situ visualization approach augmenting the source code of a program with rich profiling information. While similar techniques were already implemented into IDEs, our approach it particularly lightweight as it does not require any additional views or sidebars. By integrating all necessary information into the source code view, it promises to prevent the split-attention effect, to support navigating the code and finding performance bottlenecks quickly, and to ease the understanding of those bottlenecks. Special attention was paid on handling and comprehending multi-threaded software. We formalized the approach by providing a precise call tree model and metrics measuring the runtime consumption of methods, also considering recursion. Moreover, the approach is open for being combined with traditional representations of profiling data such as lists of hot methods. A small user study confirmed for several realistic scenarios that the approach is suitable for finding, understanding, and fixing performance issues; in some cases, however, traditional overview representations are still somewhat more convenient despite the discussed disadvantages. For practical application, we hence recommend using our approach as an extension to a traditional one.

REFERENCES