ABSTRACT
Performance problems widely exist in modern software systems. Existing performance optimization techniques, including profiling-based and pattern-based techniques, usually fail to consider the architectural impacts among methods that easily slow down the overall system performance. This paper contributes a new approach, named Speedoo, to identify groups of methods that should be treated together and reserve high priorities for performance optimization. The uniqueness of Speedoo is to measure and rank the performance optimization opportunities of a method based on 1) the architectural impact and 2) the optimization potential. For each highly ranked method, we locate a respective Optimization Space based on 5 performance patterns generalized from empirical observations. The top ranked optimization spaces are suggested to developers as potential optimization opportunities. Our evaluation on three real-life projects has demonstrated that 18.52% to 42.86% of methods in the top ranked optimization spaces indeed undertook performance optimization in the projects. This outperforms one of the state-of-the-art profiling tools YourKit by 2 to 3 times. An important implication of this study is that developers should treat methods in an optimization space together as a group rather than as individuals in performance optimization. The proposed approach can provide guidelines and reduce developers’ manual effort.

1 INTRODUCTION

Performance problems widely exist in modern software systems due to the high complexity of source code [7, 23, 29, 33, 39, 43, 59]. They can slow down production runs, hurt user experience, or even cause system failures. Therefore, the first task for software performance optimization is to find performance optimization opportunities and then conduct performance refactoring.

There mainly exist two types of techniques for finding performance optimization opportunities. Profiling-based techniques leverage instrumentation to collect profiles or execution traces. The purpose is to locate the code regions (or hot spots) that consume most resources (e.g., memory and time) [1, 3], to identify the frequently executed program paths (or hot paths) [6, 17, 27], or to fit a performance function [11, 14, 30, 60]. Such techniques use consumed resources or execution frequencies as the only performance indicator, and do not consider the performance impact due to architectural connections among software elements, e.g., modules, source files, methods, etc. Thus, they provide a narrow view for identifying performance problems, and developers have to spend a large amount of manual effort to locate the root causes that are not even in the ranked list of profilers [43, 44]. Moreover, the manifestation of the code with performance problems heavily relies on the quality of the test inputs. Despite advances on producing performance test inputs [21, 41], important performance problems still remain unrevealed.

Another type of performance problem detection techniques is pattern-based techniques. Such techniques leverage static and/or dynamic program analysis to identify code regions that match specific patterns; e.g., inefficient loops [16, 34, 44], inefficient containers [28, 53], inefficient synchronizations [36, 57], or redundant collection traversals [35]. While they are effective at identifying specific types of performance problems, they fail to be applicable...
to a wider range of performance problems. Moreover, they are not designed and thus will not provide any suggestions on the benefited code regions after the performance optimization.

In this paper, we propose a new approach, named Speedoo\(^2\), to prioritize performance optimization opportunities based on their impact on the overall system performance as well as their local potential of being optimized. Thus, the identified optimization opportunities have a high chance of being optimized; and once optimized, they can benefit to the overall system performance. A main insight is that the investigation of performance optimization should consider the architectural connections among methods, instead of treating each method in isolation, because the performance of a method can affect or be affected by the behaviors of other methods (e.g., the methods that it calls). Hence, each opportunity is a set of architecturally related methods with respect to both architecture and performance (e.g., the performance of a method is improved by optimizing the methods it calls). Different from profiling-based techniques, Speedoo further relies on the architectural knowledge to prioritize potential optimization opportunities more accurately. Different from pattern-based techniques, Speedoo is more generic, and considers the overall architecture of methods.

Our approach works in three steps to identify the optimization opportunities that should be given high priorities to improve the overall performance. First, we compute a set of metrics that measure the architectural impact and optimization potential of each method. The metrics include architectural metrics (e.g., the number of methods called by a method; the larger the number, the more impact the optimization has), dynamic execution metrics (e.g., the time a method consumes; the longer the time, the more potentially it can be optimized), and static complexity metrics (e.g., the complexity of the method; the more complex the code structure, the more potentially it can be optimized). Then, we compute an optimization priority score for each method to indicate the priority for optimization based on the metrics, and rank the methods. Finally, for each highly ranked method (i.e., candidate), we locate a respective Optimization Space composed of the methods that contribute to the dynamic performance of the candidate, which is based on five performance patterns summarized from real performance optimization cases. Each method in optimization space could be optimized to improve the overall system performance. Such optimization space is suggested to developers as potential optimization opportunities, which can provide guidelines and reduce developers’ manual effort.

We have implemented the proposed approach, and conducted an experimental study on three real-life Java projects to demonstrate the effectiveness and performance of our approach. The results show that 18.52% to 34.62% of methods in the top ranked optimization spaces calculated by Speedoo indeed undertook performance optimization during software evolution. We also compared Speedoo with the state-of-the-art profiler YourKit\(^3\) and discovered Speedoo covers 3%-16% more refactored methods. The results have demonstrated that Speedoo effectively prioritizes potential performance optimization opportunities, significantly outperforms YourKit, and scales well to large-scale projects.

In summary, our work makes the following contributions.

\(^2\)The speed optimization opportunities that we are pursuing with our approach.

\(^3\)We proposed a new approach to prioritizing performance optimization opportunities based on their architectural impact on the overall performance and their potential of being optimized.

\(^4\)We implemented the proposed approach and conducted experiments on three real-life projects to demonstrate the effectiveness and performance of our approach.

2 Preliminaries on Architecture

To understand the architectural roles of different elements in a system, Design Rule Hierarchy (DRH) algorithm [10, 47] was proposed to cluster software elements into hierarchical layers. This algorithm is based on the design rule theory [5], which indicates that a modular structure is composed of design rules (i.e., architecturally important elements) and modules (i.e., elements depending on and decoupled by the design rule elements). The DRH captures the roles of software elements as design rules and modules in hierarchical layers based on the directed dependency graph of the software elements. Each layer contains a subset of elements. The layers manifest two key features: (1) the elements in the lower layers depend on the elements in the upper layers, but not vice versa; and (2) elements in the same layer are clustered into mutually independent modules.

In general, elements in the upper layers are architecturally more important than elements in the lower layers, because the latter depend on the former. For example, Fig. 1 illustrates a DRH calculated from the "extend" and "implement" dependencies in a math calculator program. The DRH is depicted as a Design Structure Matrix, a \(n \times n\) matrix. The rows and columns represent software elements, i.e. source files in this example. Each cell indicates the dependency from the file on the row to the file on the column. The internal rectangles in the DSM represent layers and modules. This DRH captures the key base classes or interfaces as the higher level design rules, and captures the concrete classes as two lower layers. Files 1 to 4 form layer 1 on the top, which are the key interfaces and parent classes. Files 5 to 6 form layer 2 in the middle, whose elements "Extend" the elements in the top layer. Files 7 to 20 form layer 3 in the bottom, which contains 5 mutually independent modules. Each module “extend” or “implement” a design rule element in the upper layers. For example, files 16 to 20 form a module as they all “Extend” the design rule: \texttt{mij.ast.Node}, which is a parent class.

To achieve different analysis goals, the elements could be at different granularities, e.g. method level. In this paper, we apply the DRH algorithm at method level to capture the architectural importance of each method and to understand the potential benefits of
Improving it for the overall system performance. We will introduce this in details in Section 3.1.

3 METHODOLOGY

Fig. 2 shows the overview of Speedoo. Basically, our approach works in three steps: computing metrics for each method in the target system (Section 3.1), prioritizing methods based on metric values to indicate their optimization priorities (Section 3.2), and locating the optimization space for each highly-ranked method (Section 3.3). The prioritized optimization spaces are reported to developers as potential optimization opportunities; and every method in an optimization space has the potential of being optimized to improve the overall system performance. We elaborate each step as follows.

3.1 Metric Computation

In the first step, we compute the values of metrics that measure each method’s architectural impact and optimization potential. Before diving into the details, we first explain the rationality of measuring architectural impact and optimization potential.

Rationality: Given a list of hot spot methods, developers usually treat them equally by considering whether any optimization could be applied to improve the system performance. However, the impact of a performance optimization varies depending on the architecture role of the optimized method in a system. For example, if a method is invoked by many methods, the performance optimization of this method will greatly improve the system performance. It is desired that such methods should be given higher priorities for performance optimization. Thus we analyze the architectural impact of each method to estimate the impact scope of performance optimization.

In addition, considering the functionality of each method, not every method has the same potential to be improved with respect to its performance. Intuitively, if a method has higher complexity or consumes more runtime, it has more potential to be optimized. For example, if a method frequently invokes a time-consuming method through a loop structure, we can either replace the time-consuming method or avoid unnecessary invocations. It is desired that such methods should obtain a higher priority for optimization. Hence, we analyze the optimization potential of each method to estimate the potential space of performance optimization.

Based on the previous understanding, we derive a set of metrics, as shown in Table 1, to represent and estimate the architectural impact and optimization potential of a method.

Architectural Impact. We propose an architectural-role-based metric and three caller-callee-scope-based metrics, whose values are computed from the output of Titan [48].

1. Role-based Metric: Layer. As briefly discussed in Section 2, we apply the DRH algorithm to the method level call graph of a system to capture the architectural importance of each method. Each method resides in a unique layer of the DRH, and the layer number reflects its architectural role. The top layers usually contain infrastructural level methods, which are called by many other methods in the system; while the bottom layers usually contain main methods that call many other methods. The methods in the middle layers gradually transit from the infrastructural roles to control roles. In addition, the naming conventions of the methods are generally consistent with the architectural roles suggested by the DRH Layer. For example, methods in the top layers are likely to be named with “util” and “commons” etc. In comparison, methods in the bottom layers are likely to be named with “execute” and “main” etc.

Hence, we use DRH Layer as a metric to describe the architectural importance of a method. A method in an upper layer is architecturally more important than a method in a lower layer, as the performance problems with methods in top layers tend to produce larger impacts on the whole system. The higher the DRH Layer, the smaller the metric value but the higher the architectural importance.

2. Scope-based Metrics: Size, Depth, and Width. We calculate two scopes related to each method $f$: 1) the caller-scope containing the methods directly or transitively calling $f$; and 2) the callee-scope containing the methods directly or transitively called by $f$. If $f$ has a performance problem, the methods in the caller-scope also perform poorly as their execution time contains $f$’s execution time. If $f$ has a performance problem, e.g., an inefficient loop, the methods called by it may also consume more time due to frequent invocations by $f$. The actual impact of $f$ on other methods may depend on the run-time environment, but the two scopes contain all the potentially impacted methods if $f$ has a performance problem.

The caller-scope and the callee-scope can be calculated as two sub-trees by transversing from $f$ through static “call” and “called by” relations respectively. We extract three metrics from the two sub-trees of $f$ to measure its importance:

1. Size: the total number of methods in the caller-scope and the callee-scope. The larger the value, the larger the impacts of $f$, and thus the more important is $f$.
2. Depth: the sum of the depth of the two sub-trees. The larger the value, the deeper the impact of $f$, and thus the more important is $f$.
3. Width: the sum of the width of the two sub-trees. The larger the value, the broader the impact of $f$, and thus the more important is $f$.

Optimization Potential. We measure both the static complexity and dynamic execution of a method to estimate the optimization potential. For the complexity of a method, we choose five metrics based on the internal structure of the method and the external relationship with other methods, as shown in the sixth to tenth rows of Table 1. CC captures the Cyclomatic Complexity of a method.
**Table 1: Metrics Used for Measuring Architectural Impact and Optimization Potential of a Method**

<table>
<thead>
<tr>
<th>Category</th>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Architectural Metrics</strong></td>
<td>Layer</td>
<td>The residing DRH layer number of a method</td>
</tr>
<tr>
<td></td>
<td>Size</td>
<td>The total number of methods in the caller-callee-scope of a method</td>
</tr>
<tr>
<td></td>
<td>Depth</td>
<td>The sum of the depth of the caller- and callee-scope of a method</td>
</tr>
<tr>
<td></td>
<td>Width</td>
<td>The sum of the width of the caller- and callee-scope of a method</td>
</tr>
<tr>
<td><strong>Static Complexity Metrics</strong></td>
<td>CC</td>
<td>Cyclomatic complexity, i.e., the number of linearly independent paths in a method</td>
</tr>
<tr>
<td></td>
<td>SLOC</td>
<td>The source lines of code in a method</td>
</tr>
<tr>
<td></td>
<td>FanIn</td>
<td>The number of methods directly calling a method</td>
</tr>
<tr>
<td></td>
<td>FanOut</td>
<td>The number of methods directly called by a method</td>
</tr>
<tr>
<td></td>
<td>Loop</td>
<td>The number of loops in a method</td>
</tr>
<tr>
<td><strong>Dynamic Execution Metrics</strong></td>
<td>Time</td>
<td>The CPU time consumed by a method</td>
</tr>
<tr>
<td></td>
<td>OwnTime</td>
<td>The CPU time consumed by a method itself without subcalls</td>
</tr>
<tr>
<td></td>
<td>Counts</td>
<td>The invocation counts of a method</td>
</tr>
</tbody>
</table>

**SLOC, FanIn and FanOut** characterize the size and complexity of the functional responsibility of a method. The higher these metrics, the more opportunities to optimize the structure or refactor the coupling relationships to improve performance. **Loop** is selected because inefficient or redundant loops are common performance problems [16, 32, 34, 44]. If a method contains many loops, there are potential optimization opportunities in it. Note that these metrics are all widely used in the literature as useful indicators for quality problems [56]. Here we leverage them to reflect the potential of performance optimization. We use Understand [2] in our implementation to compute these metrics.

Moreover, considering the dynamic execution of a method, we use **Time**, **OwnTime** and **Counts**, as listed in the last three rows of Table 1, to measure the performance level of a method during concrete executions. A higher value of these metrics indicates a severer performance problem of a method and thus a higher possibility to improve its performance. These metrics are also widely used in the existing profiling tools [1, 3] to identify hot spot methods. In our current implementation, we obtain these metrics by running profiling tools, i.e., YourKit [3], against the test cases.

As will be discussed in Section 5, our approach can be extended with additional metrics to prioritize optimization opportunities.

### 3.2 Method Prioritization

In the second step, based on the metric values, we compute an optimization priority score for each method \( f \in F \) (where \( F \) is the set of all methods in a system) to indicate its optimization priority, and rank all methods in \( F \) according to their optimization priority scores. This step is achieved in the following three sub-steps.

**Normalizing Metrics.** To allow a unified measurement of these metrics independent of their units and ranges, we normalize each metric into a value between 0 and 1 by comparing it with the maximum and minimum value of the metric, as formulated in Eq. 1 and 2.

\[
m'_f = (m_f - \min_{i \in F} m_i) / (\max_{i \in F} m_i - \min_{i \in F} m_i) \quad (1)
\]

\[
m''_f = (\max_{i \in F} m_i - m_f) / (\max_{i \in F} m_i - \min_{i \in F} m_i) \quad (2)
\]

\( m_f \) represents the value of a metric \( m \) of a method \( f \); and \( m'_f \) is the normalized value of \( m_f \). Except for **Layer**, all the metrics are positive metrics, i.e., the higher the metric value, the higher the optimization priority. Thus, **Layer** is normalized by Eq. 2, and the other metrics are normalized by Eq. 1.

**Computing Priority Score.** Based on the normalized metric values, we compute an optimization priority score \( \text{OPS}_f \) for each method \( f \) to estimate the optimization priority, as formulated in Eq. 3.

\[
\text{OPS}_f = A_f \times \text{OP}_f
\]

The first term \( A_f \) (see Eq. 4) represents the architectural impact of a method, and the second term \( \text{OP}_f \) (see Eq. 5) indicates the optimization potential of a method. The multiplication represents the global potential performance improvement on the overall system by optimizing the method \( f \).

\[
A_f = (Layer'_f + Size'_f + Depth'_f + Width'_f)/4 \quad (4)
\]

\[
\text{OP}_f = (SC'_f + DE'_f)/2 \quad (5)
\]

Notice that \( \text{OP}_f \) considers both the static complexity (see Eq. 6) and the dynamic execution (see Eq. 7) of a method.

\[
SC'_f = (CC'_f + SLOC'_f + FanIn'_f + FanOut'_f + Loop'_f)/5 \quad (6)
\]

\[
DE'_f = (Time'_f + OwnTime'_f + Counts'_f)/3 \quad (7)
\]

Here we assign equal weights to the architectural metrics in Eq 4, to the static complexity and dynamic execution in Eq 5, to the static complexity metrics in Eq 6, and to the dynamic execution metrics in Eq 7, because we consider different factors to be equally important.

**Ranking Methods.** Once the priority scores of all the methods are computed, we rank these methods in decreasing order of their scores. Thus, the larger the optimization priority score of a method, the higher priority it should be given for performance optimization.

### 3.3 Optimization Space Localization

If a highly ranked method manifests performance problems, the optimization opportunities reside not only in the method itself, but also in the methods that contribute to its dynamic performance (e.g., its callees). However, such contributing methods may not rank high, even if they are potentially important for solving the performance problem. Hence, we locate a respective **Optimization Space** for each highly ranked method (i.e., _candidate_) by finding its contributing methods. Each method in the space could be an optimization opportunity to improve the performance of the candidate.

Based on our empirical analysis of performance problems fixed by developers, we find that the optimization space of a candidate is often determined by the performance patterns of the candidate.
Thus, we first present the performance patterns we summarized, and then introduce the steps to locate optimization spaces.

**Performance Patterns.** We manually analyzed real performance problems from four Apache systems (i.e., CXF, Avro, Ivy and PDFBox), which are all mature and industrial scale software projects. We searched the issue databases using a set of performance-related keywords, as was similarly done in the literature [23, 39, 59]. Finally, we randomly sampled 75 fixed performance issues, and analyzed how the problems were fixed (i.e., optimization strategies). Table 2 describes the summarized performance patterns with their symptom description, optimization strategies and the number of corresponding issues. These performance patterns appear nearly in all the systems. Notice that although performance patterns were also summarized in the literature (e.g., [23, 34]), their patterns focus on root causes of performance problems and are local to a method, and thus are often specific. Our patterns emphasize the optimization locations of performance problems from the perspective of symptoms and influence, and are hence global and generic. In the following, we illustrate each performance pattern.

1. **Cyclic Invocation.** The invocation of a set of methods forms a cycle. Any inefficient or frequently executed method in the circle would slow down the execution of all involved methods. For example, in Fig. 3, the method `startElement` was reported to be slow, because it’s in an invocation circle containing a slow method, `includeStarted`. This issue was solved by improving `includeStarted`. Note that the performance problem would worsen if the calling circle is repeatedly executed. An alternative solution is to break this cycle evidenced by 2 real performance issues.

2. **Expensive Recursion.** Recursion allows a method to repeatedly call itself. If the method is expensive, the performance problem will be dramatically amplified. For example, the method `parseNextToken` in Fig. 4 repeatedly calls itself by recursion. The reason was that the PDF parser converted the color spaces of images for every pixel. Finally, developers improved this method by performing color conversion in one operation for each image. Performance problems due to expensive recursion can be optimized by improving the recursion method itself or by reducing calling such methods.

3. **Frequent Invocation.** Frequently executed methods, especially the most influential methods in low DRH layers, can be optimized to produce a significant improvement of system performance, even with minor optimization. For example, in Fig. 5, the method `getDictionaryObject` is called frequently by `PDFResource` for obtaining resources. In commit 54037862, `PDFResources` was refactored to reduce the invocations of `getDictionaryObject`. This is the most common pattern in the studied issues. It can also be reduced by updating the call conditions, or avoiding unnecessary/duplicated invocations.

4. **Inefficient Method.** A method manifests poor performance due to the inefficiency in itself. Fig. 6 shows an example in Avro. The
### Table 3: Detection Conditions and Optimization Spaces of Performance Patterns

<table>
<thead>
<tr>
<th>Performance Pattern</th>
<th>Detection Condition</th>
<th>Optimization Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cyclic Invocation</td>
<td>isCallCycle and isHigh(Time) and isHigh(Counts)</td>
<td>OS(f) = all the methods in the invocation cycle</td>
</tr>
<tr>
<td>Expensive Recursion</td>
<td>isRecursion and isHigh(Time) and isHigh(Counts)</td>
<td>OS(f) = f + OS(each method in getCallers(f) that satisfies isHigh(Time))</td>
</tr>
<tr>
<td>Frequent Invocation</td>
<td>isHigh(Counts)</td>
<td>OS(f) = f + OS(each method in getCallers(f) that satisfies isHigh(Invocations))</td>
</tr>
<tr>
<td>Inefficient Method</td>
<td>isHigh(Time) and isHigh(OwnTime)</td>
<td>OS(f) = f + OS(each method in getCallers(f) that satisfies isHigh(Time))</td>
</tr>
<tr>
<td>Expensive Callee</td>
<td>isHigh(Time) and Not isHigh(OwnTime)</td>
<td>OS(f) = f + OS(each method in getCallees(f) that satisfies isHigh(Time))</td>
</tr>
<tr>
<td>Others</td>
<td>all the conditions not belonging to the above</td>
<td>OS(f) = f</td>
</tr>
</tbody>
</table>

---

**Detecting Performance Patterns.** As shown above, the optimization opportunities of a method reside in all the methods that contribute to its performance, and the scope is determined by the performance patterns it belongs to. Therefore, before computing the optimization space of each candidate, we first detect its performance patterns, using the dynamic execution metrics \((Time, OwnTime, Counts)\) as well as its call relations with other methods.

Generally, the methods in different DRH layers are expected to exhibit different level of performance. The dynamic execution metrics of methods are influenced by the architectural roles suggested by the DRH layer. Intuitively, since the infrastructural methods provide basic functions to support other methods, they usually have lower method execution time but higher invocation counts. The control methods often provide higher level functions by calling many other methods, thus they usually have lower invocation counts but higher execution time. In that sense, the performance of a method is only comparable to the methods within the same DRH layer. Hence, our general idea of detecting performance patterns is to check whether the dynamic performance of a method acts as an outlier among all the methods in the same DRH layer.

Specifically, the conditions for detecting performance patterns of a candidate \(f\) are defined in the second column of Table 3 based on their symptoms in Table 2. Here isHigh returns whether the metric value of \(f\) is a statistical outlier among the methods in the same DRH layer. For call relations, isCallCycle returns whether \(f\) is involved in a cyclic invocation and isRecursion returns whether \(f\) invokes itself.

Following the detection conditions, a method can be matched in more than one performance patterns. For example, a method matches both Frequent Invocation and Expensive Callee, which means that the optimization opportunities reside in the two optimization spaces.

**Computing Optimization Space.** Based on the performance patterns of a candidate \(f\), we compute the optimization space \(OS(f)\) to locate the potential optimization opportunities. Guided by the optimization strategies in Table 2, we locate the optimization space for each pattern in the third column of Table 3. getCallers returns the list of callers of \(f\), and getCallees returns the list of callees.

**Ranking Optimization Spaces.** The optimization space consists of all the contributing methods to a certain performance problem, and should be investigated by the developers as a whole. Thus we compute the optimization priority score of an optimization space as the average priority score of all the methods in the space to represent its optimization priority. Finally, the optimization spaces are ranked based on their priority scores. Only those highly-ranked optimization spaces are recommended to the developers as potential optimization opportunities.
To evaluate the effectiveness and performance of our approach, we conducted an experimental study on three real-life Java projects.

### 4.1 Evaluation Setup

**Subject Projects.** Table 4 describes the subject projects used in our evaluation. Three well-known Apache projects are selected: **Avro** is a data serialization system; **Ivy** is a popular dependency manager; and **PDFBox** is a tool for working with PDF documents. They are selected due to the following reasons. First, we can collect sufficient performance problems from the formal issue reports documented in JIRA and the entire commits history traced through GitHub (see below). Second, they are non-trivial projects that contain thousands of methods. Third, these projects belong to different domains, and have high performance requirements. Notice that **CXF**, used in Section 3.3, is not used in the evaluation as we failed to run its original tests. Two releases are selected from each project, namely analyzed release (column 2-5) and refactored release (column 6-7). We apply **Speedoo** on the analyzed release, and evaluate whether the methods in highly-ranked optimization spaces are actually optimized in the refactored release. Generally, the studied time frame should be long enough to allow plenty of performance problems reported and fixed. Considering the projects’ update frequencies, the studied interval is approximately five years in **Avro** and **Ivy**, and three years in **PDFBox**. Then two releases in Table 4 were selected to cover most performance problems within this time frame.

**Ranked Methods to Optimize.** We apply our approach on the analyzed release of each project to identify highly-ranked optimization spaces. The methods in these optimization spaces are suggested for optimization to developers. The suggested methods all suffer from performance problems (running slowly or frequently called).

Table 5 reports the distribution of performance patterns aggregating the identified methods. For each pattern, we list the number of pattern instances detected in the project under column Count as well as the average size (i.e. number of methods) in the optimization spaces under column Avg. Size. We can observe that:

- Frequent Invocation and Expensive Callee are the most prevalent performance patterns, implying developers should pay specific attention to frequently executed and slow methods.
- The impact scope of Cyclic Invocation and Frequent Invocation is larger than other performance patterns, with the average size of optimization spaces ranging from 4 to 19. Therefore, it may take more time for developers to solve these performance problems.

**Actually Optimized Methods.** We collect the performance problems fixed during the two selected releases as the ground truth.

For each project, we first identified all the performance issues documented in JIRA by matching performance-related keywords (e.g., “performance” and “optimization”) in the issue descriptions, as was similarly done in the literature [23, 39, 59]. Second, we scanned the commits between the analyzed and refactored releases to extract the matched commits that mention performance issue IDs in the commit messages. Further, to avoid missing undocumented performance issues, we also scanned those unmatched commits to extract the commits that mention performance-related keywords in the commit messages. Finally, the resulting performance issues and the corresponding linked commits were manually analyzed to filter out non-performance optimization, and the methods participating in performance optimization were identified as the ground truth.

**Research Questions.** We designed the experiments to answer the following three research questions.

- **RQ1:** How is the effectiveness of our approach in prioritizing performance optimization opportunities?
- **RQ2:** How is the sensitivity of the metrics on the effectiveness of our approach?
- **RQ3:** How is the performance overhead of our approach?

### 4.2 Effectiveness Evaluation (RQ1)

To evaluate the effectiveness of the proposed approach, we further breakdown the evaluation into 2 sub-questions below:

- **RQ1-1:** How many of the ranked methods to optimize from the analyzed release are actually optimized in the refactored release? How does our approach compare to random search (assuming developers have no knowledge what to do) and YourKit? This question evaluates whether Speedoo can truly identify worthwhile optimization opportunities.

To answer this question, we use the density, i.e., the percentage of actually optimized methods in highly-ranked optimization spaces to evaluate the effectiveness of Speedoo. As comparison, we also calculate the density of actually optimized methods in the whole system, which approximates the effectiveness of a random search, and the density of actually optimized methods in the hot spots reported by YourKit. We do not compare Speedoo with pattern-based techniques because each pattern-based technique often detects only one kind of performance problems and most of them are not publicly available.

- **RQ1-2:** How the before/after optimization priority of the actually optimized method and of the actually optimized optimization space change? Presumably, after performance optimization, the performance of the method and/or methods in its optimization space change? Therefore, it may take more time for developers to solve these performance problems.

**Actually Optimized Methods.** We collect the performance problems fixed during the two selected releases as the ground truth.

For each project, we first identified all the performance issues documented in JIRA by matching performance-related keywords (e.g., “performance” and “optimization”) in the issue descriptions, as was similarly done in the literature [23, 39, 59]. Second, we scanned the
Table 5: Distribution of Detected Performance Patterns

<table>
<thead>
<tr>
<th>Project</th>
<th>Cyclic Invocation Count</th>
<th>Expensive Recursion Count</th>
<th>Frequent Invocation Count</th>
<th>Inefficient Method Count</th>
<th>Expensive Callee Count</th>
<th>Others Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avro</td>
<td>5</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>555</td>
</tr>
<tr>
<td>Ivy</td>
<td>32</td>
<td>19</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>1759</td>
</tr>
<tr>
<td>PDFBox</td>
<td>7</td>
<td>7</td>
<td>2</td>
<td>3</td>
<td>460</td>
<td>3584</td>
</tr>
</tbody>
</table>

Table 6: Effectiveness of Optimization Space Ranks

<table>
<thead>
<tr>
<th>Project</th>
<th>Whole System</th>
<th>Top 25 Spaces</th>
<th>Top 50 Spaces</th>
<th>Top 100 Spaces</th>
<th>YourKit Hot Spots</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avro</td>
<td>43</td>
<td>1,238.00</td>
<td>5.00</td>
<td>26.00</td>
<td>19.23</td>
</tr>
<tr>
<td>Ivy</td>
<td>61</td>
<td>4,675.00</td>
<td>5.00</td>
<td>27.00</td>
<td>18.52</td>
</tr>
<tr>
<td>PDFBox</td>
<td>621</td>
<td>8,352.00</td>
<td>12.00</td>
<td>28.00</td>
<td>42.86</td>
</tr>
</tbody>
</table>

Table 7: Change of Optimization Priority Scores after Performance Optimization

<table>
<thead>
<tr>
<th>Project</th>
<th>Optimized Methods</th>
<th>Methods in Optimized Optimization Spaces</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total Higher Rank</td>
<td>Lower Rank</td>
</tr>
<tr>
<td>Avro</td>
<td>36</td>
<td>18</td>
</tr>
<tr>
<td>Ivy</td>
<td>40</td>
<td>8</td>
</tr>
<tr>
<td>PDFBox</td>
<td>428</td>
<td>250</td>
</tr>
</tbody>
</table>

Figure 8: Metric Sensitivity in Subject Projects
Wilcoxon test does not indicate a significant difference in Avro, because the number of target methods is small (i.e., only 48 methods), but most of the methods in actually optimized optimization spaces (i.e., 34 out of 48 methods) are ranked lower in the refactored release. These results indicate that even if the performance of optimized methods does not show significant improvements, the performance of the methods in their spaces would benefit from the optimization. This also demonstrates the rationality of our prioritization on the level of optimization spaces but not on the level of methods.

The small effect sizes in Table 7 indicate that the performance improvements are limited in most cases. In fact, we find performance optimization performed by developers is usually minor, e.g., removing an unnecessary type cast. We also find some highly-ranked methods were optimized many times in history. It implies that performance optimization is a long way in software evolution.

**Summary.** Speedoo is effective in prioritizing potential performance optimization opportunities, and outperforms a state-of-the-art profiling tool YourKit; and the prioritization is advisable as the priorities of the methods in actually optimized optimization spaces become significantly lower after performance optimization.

### 4.3 Sensitivity Evaluation (RQ2)

Three groups of metrics are used in our approach (i.e., architectural metrics, static complexity metrics, and dynamic execution metrics), the question is: what is the sensitivity of these metrics on the effectiveness of our approach? To answer it, we remove each group of metrics from our approach, and then estimate the metric sensitivity by comparing the effectiveness of different approaches.

**Answer RQ2.** The results are visually illustrated in Fig. 8, where the x-axis represents the number of methods in top optimization spaces, and the y-axis represents the number of actually optimized methods. Generally, the curves of our approach without architectural impact metrics (AI), static complexity metrics (SC), and dynamic execution metrics (DE) are lower than the curve of our approach with all the metrics, especially significant for Avro and Ivy. The results indicate that each group of metrics contributes to the effectiveness of our approach, and thus a combination of them is reasonable.

Similarly, we also evaluate the sensitivity of each metric. After removing any metric in Table 1 from our approach, the density in top 25 spaces decreases to 14.81%-19.23%, 10.34%-18.52%, and 36.36%-44.44% for three projects respectively. The density is lower than or same with Speedoo, with only one exception coming from removing CC for PDFBox. It indicates that each used metric is vital to Speedoo. Besides, we tried to assign different weights to these metrics and found that, using the same weights performed best in general.

In order to cover more methods when collecting dynamic execution metrics, we run both the original tests in systems as well as the tests automatically generated by the unit testing tool EvoSuite [19]. In comparison, we show the effectiveness of our approach without generated tests in Fig. 8 (*Our Approach w/o Gen. Tests*). The approach without generated tests turns out to be as effective as the whole approach. This is because the execution behaviors in automatically generated tests are different from the real behaviors in manually written tests. This suggests that the effectiveness of our approach is also affected by whether the collected dynamic metrics can reflect real executions that manifest performance problems.

**Summary.** All the metrics used in this study have contribution to the effectiveness of Speedoo. Besides, dynamic execution metrics should be obtained through real execution behaviors.

### 4.4 Performance Evaluation (RQ3)

This research question addresses how the performance overhead of Speedoo is. To answer this question, we collect the time spent in each step over the three projects, consisting of metric computation, method prioritization, and optimization space localization.

**Answer RQ3.** Table 8 presents the time consumed in each step of the proposed approach. We can see that computing dynamic execution metrics takes the most time. Note that this step includes the time for running original tests with YourKit, as well as the time for generating tests by EvoSuite and running them with YourKit, which are respectively reported in Table 8. As the result of RQ2 indicates that our approach without generated tests is as effective as our approach with generated tests, the performance overhead would be significantly improved when applying Speedoo without generated tests.

Similarly, our approach scales to large-scale project like PDFBox.

**Summary.** Speedoo scales to large-scale projects; and computing dynamic metrics takes most of the time as it needs to run tests.

### 5 DISCUSSION

**Threats to Validity.** First, the values of dynamic execution metrics are affected by the tests that produce them. Basically, Speedoo can be improved through the tests that cover more suspect methods and reflect real execution behaviors. Hence, in this study, the dynamic execution metrics were obtained by executing not only the original tests but also the automatically generated tests, although it turns out that automatically generated tests are not very helpful as shown in Section 4.3. In the future, we plan to further investigate how to generate advanced tests [21, 41], and integrate it into our approach.

Second, the ground truth was collected from the documented performance issues and commit logs of subject projects. In some cases, developers did not document optimization operations in issue reports when committing changes. We mitigated this problem by also matching performance keywords in commit messages in case of lacking issue records. The second problem is the informal description of performance problems in issues and commit logs. To mitigate it, we adopted a list of performance keywords that is widely used in literature, and manually checked the matched commits and issues.
Analysis of Top Optimization Spaces. We manually checked some really optimized methods in top optimization spaces. We find that performance optimization tends to be conducted on the methods in higher DRH layers. For example, `getDictionaryObject` in Fig. 5 is in the highest DRH layer and its optimization space is ranked 2. We also find that the methods were optimized mostly by minor changes. This indicates that time is not everything in motivating performance optimization, and architectural considerations are also important. This also explains why Speedoo outperforms YourKit.

We also checked those highly-ranked methods which were not optimized. We can discover that, some methods were not optimized, although users reported a performance issue, because it would incur high maintenance cost (e.g., issue AVRO-1809) or the code is kept remained for future extensions (e.g., issue AVRO-464); and some methods have been optimized before the analyzed release, but the previous optimization only reduces the performance problem and cannot totally fix it (e.g., issue AVRO-911).

Extensions. Speedoo provides a ranked list of optimization opportunities to developers. One potential extension is to integrate pattern-based techniques into Speedoo. Once pattern-based techniques locate a pattern in the system, we can extend Speedoo to determine optimization spaces of the methods in the pattern, which indicates the impact of the pattern on other methods as well as the potential improvement that can be achieved by performance optimization. Another possible extension is to integrate more metrics into Speedoo to better reveal performance problems. For example, we can give higher priorities to the methods with more expensive API calls. Furthermore, this study assigns equal weights to the factors in Eq. 5, Eq. 6, and Eq. 7. If additional metrics are integrated into our approach, the factors can be assigned individual weights which are tuned with different combinations of weights.

6 RELATED WORK

Performance Understanding. Several empirical studies have been conducted to understand the characteristics of performance problems from different perspectives [7, 23, 29, 33, 39, 43, 59]. They investigated root causes of performance problems as well as how performance problems are introduced, discovered, reported, and fixed, which provides valuable insights and guidelines for designing performance profiling and performance problem detection approaches.

Profiling-Based Performance Analysis. Profiling tools [1, 3] are widely used to locate hot spot methods that consume most resources (e.g., memory and time). Besides, several path-sensitive profiling techniques (e.g., [6, 17, 27]) have been proposed to analyze execution traces of an instrumented program for predicting execution frequency of program paths and identifying hot paths. Further, there has been some recent work on input-sensitive profiling techniques (e.g., [14, 20, 60]). They execute an instrumented program with a set of different input sizes, measure the performance of each basic block, and fit a performance function with respect to input sizes. Moreover, Mudduluru and Ramanathan [30] proposed an efficient flow profiling technique for detecting memory-related performance problems. Chen et al. [11] applied probabilistic symbolic execution to generate performance distributions, while Brünink and Rosenblum [9] used in-field data to extract performance specifications. These profiling techniques are more helpful to comprehend performance than to pinpoint performance problems, because they use resources or execution frequencies as the only performance indicator and do not consider the performance impact among architecturally connected code. Thus, developers have to waste many manual effort to locate root causes. Differently, our approach tries to relieve such burden from developers by prioritizing optimization opportunities.

Besides, several advances [4, 21, 22, 41, 58] have been made to further analyze the profiles or execution traces for performance problem detection. Specifically, Ammons et al. [4] find expensive call sequences in call-tree profiles, and the call sequences that are significantly more expensive in one call-tree profile than in another call-tree profile. Han et al. [22] and Yu et al. [58] mine performance behavioral patterns from stack traces and execution traces. These approaches heavily rely on the test inputs that generate the profiles; i.e., performance problems might stay unrevealed as they are not manifested by those test inputs. Instead, our approach also considers architectural impact and static complexity to avoid solely depending on dynamic executions.

Pattern-Based Performance Problem Detection. A large body of work has been done on the detection of specific performance problems. One line of work focuses on loop-related performance problems; e.g., inefficient loops [16, 34], redundant loops [32, 34] and redundant collection traversals [35]. Another line of work focuses on memory-related performance problems; e.g., under-utilized or over-utilized containers [25, 28, 40, 53], inefficient usage of temporary structures or objects [18, 51, 52, 55] and reusable or cacheable data [8, 15, 31, 50, 54]. Besides these two main lines of work, researchers have also investigated performance problems caused by inefficient or incorrect synchronization [36, 57], slow responsiveness of user interfaces [24, 37], heavy input workloads [49], inefficient order of evaluating subexpressions [38] and anomalous executions [26]. In addition, performance anti-patterns [42] are used to detect performance problems satisfying the anti-patterns [12, 45]. While they are effective at detecting specific types of performance problems, they fail to be applicable to a wider range of performance problem types, as these patterns focus on the root causes of specific performance problems. Differently, the patterns in this study consider the optimization locations from execution symptoms, and are more generic.

7 CONCLUSIONS

We proposed and implemented a novel approach, named Speedoo, to identify the optimization opportunities that should be given high priorities to performance-critical methods to improve the overall system performance. Our evaluation on real-life projects has indicated that our approach effectively prioritizes potential performance optimization opportunities to reduce the manual effort of developers, and significantly outperforms random search and a state-of-the-art profiling tool YourKit.

ACKNOWLEDGMENTS

The work is supported by the National Key Basic Research and Development Program of China (2014CB340702), the National Natural Science Foundation of China (61472175, 61472178, 61772263), the Natural Science Foundation of Jiangsu Province of China (BK20140611), and the program B for Outstanding PhD candidate of Nanjing University.