Speedoo: Prioritizing Performance Optimization Opportunities

Zhifei Chen
State Key Laboratory for Novel Software Technology
Nanjing University, China

Bihuan Chen
School of Computer Science, Shanghai Key Laboratory of Data Science, and Shanghai Institute of Intelligent Electronics & Systems
Fudan University, China

Lu Xiao
Xiao Wang
School of Systems and Enterprises
Stevens Institute of Technology
United States

Lin Chen*
State Key Laboratory for Novel Software Technology
Nanjing University, China

Yang Liu
School of Computer Science and Engineering
Nanyang Technological University
Singapore

Baowen Xu*
State Key Laboratory for Novel Software Technology
Nanjing University, China

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 deserve 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.

CCS CONCEPTS

• **Software and its engineering → Software performance**;

KEYWORDS

Performance, Metrics, Architecture

*Corresponding authors.

ICSE '18, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5638-1/18/05…$15.00
https://doi.org/10.1145/3180155.3180229

ACM Reference Format:
Zhifei Chen, Bihuan Chen, Lu Xiao, Xiao Wang, Lin Chen, Yang Liu, and Baowen Xu. 2018. Speedoo: Prioritizing Performance Optimization Opportunities. In *ICSE '18: 40th International Conference on Software Engineering, May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3180155.3180229

1 INTRODUCTION

Performance problems[1] 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

[1]Performance problems are sometimes also called performance bugs or performance issues. For consistency, we use performance problems throughout the paper.

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.

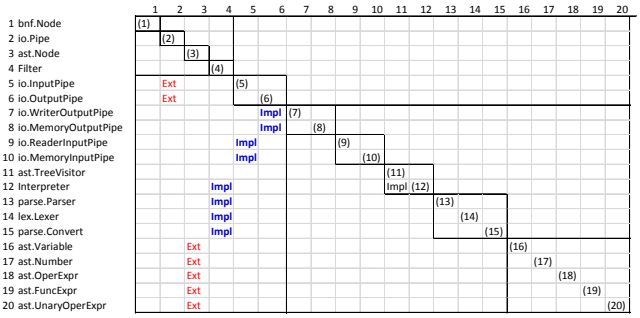


Figure 1: An Example of DRH

- 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.
- 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: *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

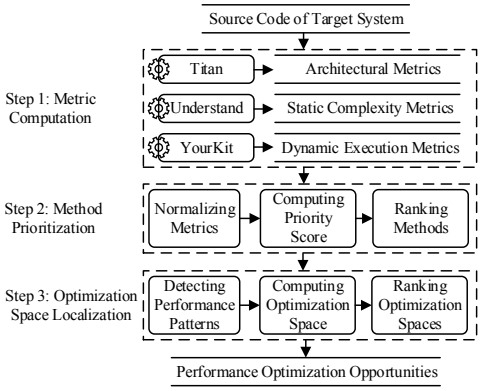


Figure 2: Approach Overview of Speedoo

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

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

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 OPS for each method f to estimate the optimization priority, as formulated in Eq. 3.

$$OPS_f = AI_f \times OP_f \quad (3)$$

The first term AI_f (see Eq. 4) represents the architectural impact of a method, and the second term 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.

$$AI_f = (Layer'_f + Size'_f + Depth'_f + Width'_f)/4 \quad (4)$$

$$OP_f = (SC_f + DE_f)/2 \quad (5)$$

Notice that 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.

Table 2: Summarized Performance Patterns from Four Apache Systems

Performance Pattern	Symptom Description	Optimization Strategies	# Issues
Cyclic Invocation	a slow or frequently executed method is in an invocation cycle	improve any method in the cycle	1
		change dependency relations to break the circle	2
Expensive Recursion	a slow method calls itself repeatedly	improve the method	2
		avoid or reduce the calls in its callers	3
Frequent Invocation	a method is frequently executed	improve the method	3
		cache the returned values to reduce calls	10
		add or update the call conditions in its callers	10
		avoid unnecessary or duplicated calls in the callers	4
Inefficient Method	a method is slow in executing its own code	improve the method	15
		reduce its calls in its callers	5
		replace its calls with cheaper callees in its callers	6
Expensive Callee	a method is slow in executing its callees' code	improve the expensive callees	15
		reduce the calls to expensive callees	5
		replace the calls to expensive callees with cheaper callees	6
Others	all the issues not belonging to the above	–	14

Commit: Ivy 6f8302fe
Candidate: XmlSettingsParser.startElement(String,String,String,Attributes)

```
private void includeStarted(Map attributes) throws IOException,
        ParseException {
  ... new XmlSettingsParser(ivy).parse(configurator, settingsURL); ...
}
private void parse(Configurator configurator, URL configuration) throws
        IOException, ParseException {
  ... doParse(configuration); ⋯
}
...
public void startElement(String uri, String localName, String qName,
        Attributes att) throws SAXException {
  ... includeStarted(attributes); ⋯
}
```

Optimization: improve includeStarted(Map) by removing unnecessary cast

Figure 3: A Case for Cyclic Invocation

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 the cycle evidenced by 2 real performance issues.

Commit: PDFBox 7929477d
Candidate: PDFStreamParser.parseNextToken()

```
private Object parseNextToken() throws IOException {
  ...
  while((nextToken = parseNextToken()) instanceof COSName ) {
    Object value = parseNextToken(); ...
  }
}
```

Optimization: improve this method by using per-image color conversion

Figure 4: A Case for Expensive Recursion

Commit: PDFBox 54037862
Candidate: COSDictionary.getDictionaryObject(COSName)

```
public Map<String, PDFont> getFonts() throws IOException {
  ... COSBase font = fontsDictionary.getDictionaryObject(fontName); ...
}
public Map<String, PDColorSpace> getColorSpaces() {
  ... COSDictionary csDictionary = (COSDictionary)
        resources.getDictionaryObject(COSName.COLORSPACE); ...
}
```

Optimization: reduce the calls to getDictionaryObject(COSName) in getFonts() and getColorSpaces() through caching

Figure 5: A Case for Frequent Invocation

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 *PDResource* for obtaining resources. In commit 54037862, *PDResources* 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

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

Commit: Avro e0966a1e
Candidate: GenericData.deepCopy(Schema, T)

```
public <T> T deepCopy(Schema schema, T value) {
  ...
  case FLOAT:
    return (T)new Float((Float)value);
  case INT:
    return (T)new Integer((Integer)value);
  case LONG:
    return (T)new Long((Long)value);
  ...
}
```

Optimization: improve this method by avoiding creating new instances

Figure 6: A Case for Inefficient Method

Commit: PDFBox 4746da78
Candidate: PreflightParser.parseObjectDynamically(long, int, boolean)

```
protected COSBase parseObjectDynamically(long objNr, int objGenNr,
          boolean requireExistingNotCompressedObj) throws IOException {
  ... final Set<Long> refObjNrs =
          xrefTrailerResolver.getContainedObjectNumbers(objstmObjNr); ...
}
```

Optimization: replace xrefTrailerResolver.getContainedObjectNumbers(int) with xrefTrailerResolver.getXrefTable()

Figure 7: A Case for Expensive Callee

method *deepCopy* created many new instances for primitives, which is unnecessary. This method was optimized in commit e0966a1e by removing unnecessary instance creations. If an inefficient method can hardly be optimized, the overall performance of a system can still be improved by reducing the invocations of this method.

5. Expensive Callee. The performance of a method is also determined by the performance of its callees. Expensive callee is a performance pattern where a method is slow in executing its callees' code. The optimization strategies include improving the expensive callees, and reducing or avoiding the invocations of expensive callees. The example in Fig. 7 illustrates the performance optimization of the method *parseObjectDynamically*. As reported, the method took quite some time which was mostly spent in *getContainedObjectNumbers*. The reason was that many full scans were performed in *getContainedObjectNumbers*. Developers fixed this issue by replacing the calls to *getContainedObjectNumbers* with the calls to *getXrefTable*.

Note that the optimization of an inefficient method can be treated either as the *Inefficient Method* pattern (i.e., to improve the inefficient method) or as the *Expensive Callee* pattern (i.e., to improve the caller of the inefficient method). Thus, we counted the number of corresponding issues equally for these two patterns. We separately define these two patterns for locating individual optimization opportunities for each target method. There were 14 issues that exhibit no specific patterns, e.g., moving a generic method from other classes to a target class, avoiding using type parameters, and package reorganization. Current patterns are defined among architecturally connected methods, which can be extended to support high-level structures like packages and modules.

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* and *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 a 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.

Table 4: Subject Projects

Project	Analyzed Release				Refactored Release	
	Ver.	Date	LOC (#)	Meth. (#)	Ver.	Date
Avro	1.3.0	2010-05-10	10,637	1,238	1.8.1	2016-05-14
Ivy	2.0.0	2009-01-18	41,939	4,673	2.4.0	2014-12-13
PDFBox	1.8.4	2014-01-27	88,435	8,352	2.0.4	2016-12-12

4 EVALUATION

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 should improve. Consequently, the optimization priority score offered by our approach should capture such improvement to show its effectiveness. Otherwise, it implies that the proposed optimization priority score may not be effective (the another possibility is that the optimization is not successful).
  To answer this question, we use Wilcoxon test [46] to compare the optimization priority scores of the actually optimized methods and of the actually optimized optimization spaces in two releases. Wilcoxon test is a non-parametric statistical hypothesis test, which is used to compare two groups of independent samples to assess whether their population mean ranks differ. Without assuming that the data has a normal distribution, we test at the significance level of 0.001 to investigate whether the optimization

Table 5: Distribution of Detected Performance Patterns

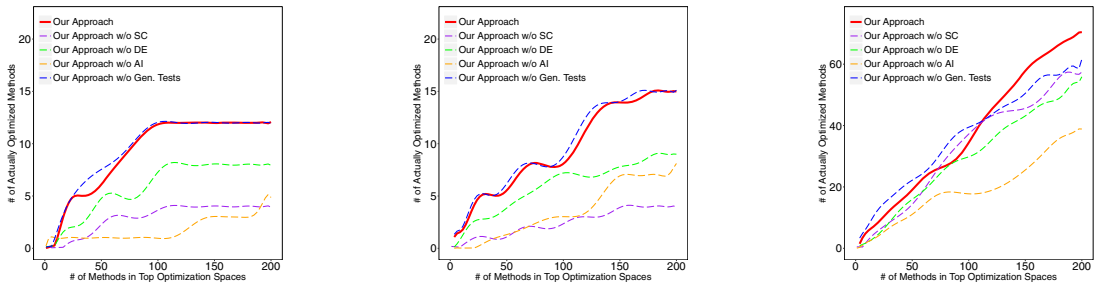
Project	Cyclic Invocation		Expensive Recursion		Frequent Invocation		Inefficient Method		Expensive Callee		Others
	Count	Avg. Size	Count	Avg. Size	Count	Avg. Size	Count	Avg. Size	Count	Avg. Size	Count
Avro	5	4	2	2	76	4	3	2	22	2	555
Ivy	32	19	2	3	270	5	12	2	118	2	1759
PDFBox	7	7	2	3	460	16	68	1	40	2	3584

Table 6: Effectiveness of Optimization Space Ranks

Project	Whole System			Top 25 Spaces			Top 50 Spaces			Top 100 Spaces			YourKit Hot Spots		
	Opt.	Total	Den.	Opt.	Total	Den.	Opt.	Total	Den.	Opt.	Total	Den.	Opt.	Total	Den.
Avro	43	1,238	3.47%	5	26	19.23%	7	60	11.67%	12	110	10.91%	4	110	3.63%
Ivy	61	4,673	1.31%	5	27	18.52%	8	72	11.11%	14	166	8.43%	9	166	5.42%
PDFBox	621	8,352	7.44%	12	28	42.86%	24	61	39.34%	48	126	38.10%	16	96	16.67%

Table 7: Change of Optimization Priority Scores after Performance Optimization

Project	Optimized Methods					Methods in Optimized Optimization Spaces				
	Total	Higher Rank	Lower Rank	*p*-value	*d*	Total	Higher Rank	Lower Rank	*p*-value	*d*
Avro	36	18	18	0.7462	-	48	14	34	0.0878	-
Ivy	40	8	32	0.0035	-	176	17	159	**3.431e-21**	**0.1288**
PDFBox	428	250	178	0.9999	-	539	169	370	**1.940e-05**	**0.1031**



(a) Avro **(b) Ivy** **(c) PDFBox**

Figure 8: Metric Sensitivity in Subject Projects

priority scores of the methods and the optimization spaces significantly become lower after performance optimization. Furthermore, we use Cliff's Delta effect size [13] to measure the magnitude of the difference if Wilcoxon test indicates a significant difference. Cliff's Delta (i.e., d) measures how often the values in a distribution are larger than the values in another distribution. The effect size is *negligible* for $|d| < 0.147$, *small* for $0.147 \leq |d| < 0.33$, *medium* for $0.33 \leq |d| < 0.474$, and *large* for $|d| \geq 0.474$.

Answer RQ1-1. Table 6 compares five groups of methods: the whole methods in a system (column 2-4), the methods in top 25 (column 5-7), 50 (column 8-10) and 100 (column 11-13) optimization spaces, and the hot spots reported by YourKit (column 14-16). For each group of methods, we report the number of actually optimized methods (column *Opt.*), the total number of methods (column *Total*), and the density (i.e., *Opt.*/*Total*) in this group (column *Den.*).

The table shows 19% to 43% of methods in top 25 spaces are actually optimized. When the investigated optimization spaces adjust from top 25 to top 100 ones, the number of methods increases drastically and the density of actually optimized methods decreases slightly. It suggests developers to focus on the top few optimization spaces to narrow down the methods as candidates to optimize.

In comparison, the density of actually optimized methods in the whole system ranges from 1.31% to 7.44%. The density by our approach is several orders of magnitude higher than random search (when developers have no clue). YourKit reports a list of hot spots and suggests each one as a potential optimization opportunity. In order to compare Speedoo with YourKit, we selected the same number of hot spots (excluding library methods) with the number of methods in top 100 optimization spaces. Noting that YourKit only reported 96 hot spots in PDFBox. The density shown in column 13 and column 16 indicate that our approach outperforms YourKit by 2-3 times. We can conclude that our approach provides a significantly more effective prioritization of optimization opportunities, compared with random search and YourKit, and thus is worthwhile.

Answer RQ1-2. Table 7 reports the change of optimization priority scores in the actually optimized methods and the methods in optimized optimization spaces. For each group of methods, it reports the total number of target methods (column *Total*), the number of methods whose priority scores increase (column *Higher Rank*), the number of methods whose priority scores decrease (column *Lower Rank*), the p-value of Wilcoxon test, and Cliff's Delta d if Wilcoxon test shows significant lower values (p-value < 0.001).

The table reveals that the optimization priorities of actually optimized methods are not significantly lower after performance optimization; in contrast, the optimization priorities of the methods in actually optimized optimization spaces are significantly lower.

Table 8: Performance Overhead of Speedoo

Project	Metric Computation			Method Prioritization	Optimization Space Localization
	Architectural Metrics	Static Complexity Metrics	Dynamic Execution Metrics		
Avro	0.4 s	2.8 s	1.5 h + 2.5 h	0.3 s	3.0 s
Ivy	2.2 s	22.0 s	1.0 h + 4.5 h	3.1 s	79.7 s
PDFBox	3.7 s	30.6 s	3.0 h + 5.5 h	10.3 s	195.8 s

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 impact 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 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. Generally, 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 guidances 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.

REFERENCES

[1] 2017. JPROFILER. https://www.ej-technologies.com/products/jprofiler/overview.html. (2017).
[2] 2017. Understand. https://scitools.com/. (2017).
[3] 2017. YourKit. https://www.yourkit.com/. (2017).
[4] Glenn Ammons, Jong-Deok Choi, Manish Gupta, and Nikhil Swamy. 2004. Finding and Removing Performance Bottlenecks in Large Systems. In *ECOOP*. 172–196.
[5] Carliss Y. Baldwin and Kim B. Clark. 1999. *Design Rules: The Power of Modularity Volume 1*. MIT Press, Cambridge, MA, USA.
[6] Thomas Ball and James R. Larus. 1996. Efficient Path Profiling. In *MICRO*. 46–57.
[7] S. Baltes, O. Moseler, F. Beck, and S. Diehl. 2015. Navigate, Understand, Communicate: How Developers Locate Performance Bugs. In *ESEM*. 1–10.
[8] Suparna Bhattacharya, Mangala Gowri Nanda, K. Gopinath, and Manish Gupta. 2011. Reuse, Recycle to De-bloat Software. In *ECOOP*. 408–432.
[9] Marc Brünink and David S. Rosenblum. 2016. Mining Performance Specifications. In *FSE*. 39–49.
[10] Yuanfang Cai and Kevin J. Sullivan. 2006. Modularity Analysis of Logical Design Models. In *ASE*. 91–102.
[11] Bihuan Chen, Yang Liu, and Wei Le. 2016. Generating Performance Distributions via Probabilistic Symbolic Execution. In *ICSE*. 49–60.
[12] Tse-Hsun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. 2014. Detecting Performance Anti-patterns for Applications Developed Using Object-relational Mapping. In *ICSE*. 1001–1012.
[13] Norman Cliff. 1993. Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological Bulletin* 114, 3 (1993), 494.
[14] Emilio Coppa, Camil Demetrescu, and Irene Finocchi. 2012. Input-sensitive Profiling. In *PLDI*. 89–98.
[15] Luca Della Toffola, Michael Pradel, and Thomas R. Gross. 2015. Performance Problems You Can Fix: A Dynamic Analysis of Memoization Opportunities. In *OOPSLA*. 607–622.
[16] Monika Dhok and Murali Krishna Ramanathan. 2016. Directed Test Generation to Detect Loop Inefficiencies. In *FSE*. 895–907.
[17] Evelyn Duesterwald and Vasanth Bala. 2000. Software Profiling for Hot Path Prediction: Less is More. In *ASPLOS*. 202–211.
[18] Bruno Dufour, Barbara G. Ryder, and Gary Sevitsky. 2008. A Scalable Technique for Characterizing the Usage of Temporaries in Framework-intensive Java Applications. In *FSE*. 59–70.
[19] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: Automatic Test Suite Generation for Object-Oriented Software. In *ESEC/FSE*. 416–419.
[20] Simon F. Goldsmith, Alex S. Aiken, and Daniel S. Wilkerson. 2007. Measuring Empirical Computational Complexity. In *ESEC-FSE*. 395–404.
[21] Mark Grechanik, Chen Fu, and Qing Xie. 2012. Automatically Finding Performance Problems with Feedback-directed Learning Software Testing. In *ICSE*. 156–166.
[22] Shi Han, Yingnong Dang, Song Ge, Dongmei Zhang, and Tao Xie. 2012. Performance Debugging in the Large via Mining Millions of Stack Traces. In *ICSE*. 145–155.
[23] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and Detecting Real-world Performance Bugs. In *PLDI*. 77–88.
[24] Milan Jovic, Andrea Adamoli, and Matthias Hauswirth. 2011. Catch Me if You Can: Performance Bug Detection in the Wild. In *OOPSLA*. 155–170.
[25] Changhee Jung, Silvius Rus, Brian P. Railing, Nathan Clark, and Santosh Pande. 2011. Brainy: Effective Selection of Data Structures. In *PLDI*. 86–97.
[26] Charles Killian, Karthik Nagaraj, Salman Pervez, Ryan Braud, James W. Anderson, and Ranjit Jhala. 2010. Finding Latent Performance Bugs in Systems Implementations. In *FSE*. 17–26.
[27] James R. Larus. 1999. Whole Program Paths. In *PLDI*. 259–269.
[28] Lixia Liu and Silvius Rus. 2009. Perflint: A Context Sensitive Performance Advisor for C++ Programs. 265–274.
[29] Yepang Liu, Chang Xu, and Shing-Chi Cheung. 2014. Characterizing and Detecting Performance Bugs for Smartphone Applications. In *ICSE*. 1013–1024.
[30] Rashmi Mudduluru and Murali Krishna Ramanathan. 2016. Efficient Flow Profiling for Detecting Performance Bugs. In *ISSTA*. 413–424.
[31] Khanh Nguyen and Guoqing Xu. 2013. Cachetor: Detecting Cacheable Data to Remove Bloat. In *ESEC/FSE*. 268–278.
[32] Adrian Nistor, Po-Chun Chang, Cosmin Radoi, and Shan Lu. 2015. CARAMEL: Detecting and Fixing Performance Problems That Have Non-Intrusive Fixes. In *ICSE*. 902–912.
[33] Adrian Nistor, Tian Jiang, and Lin Tan. 2013. Discovering, Reporting, and Fixing Performance Bugs. In *MSR*. 237–246.
[34] Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. 2013. Toddler: Detecting Performance Problems via Similar Memory-access Patterns. In *ICSE*. 562–571.
[35] Oswaldo Olivo, Isil Dillig, and Calvin Lin. 2015. Static Detection of Asymptotic Performance Bugs in Collection Traversals. In *PLDI*. 369–378.
[36] Michael Pradel, Markus Huggler, and Thomas R. Gross. 2014. Performance Regression Testing of Concurrent Classes. In *ISSTA*. 13–25.
[37] Michael Pradel, Parker Schuh, George Necula, and Koushik Sen. 2014. EventBreak: Analyzing the Responsiveness of User Interfaces Through Performance-guided Test Generation. In *OOPSLA*. 33–47.
[38] Marija Selakovic, Thomas Glaser, and Michael Pradel. 2017. An Actionable Performance Profiler for Optimizing the Order of Evaluations. In *ISSTA*. 170–180.
[39] Marija Selakovic and Michael Pradel. 2016. Performance Issues and Optimizations in JavaScript: An Empirical Study. In *ICSE*. 61–72.
[40] Ohad Shacham, Martin Vechev, and Eran Yahav. 2009. Chameleon: Adaptive Selection of Collections. In *PLDI*. 408–418.
[41] Du Shen, Qi Luo, Denys Poshyvanyk, and Mark Grechanik. 2015. Automating Performance Bottleneck Detection Using Search-based Application Profiling. In *ISSTA*. 270–281.
[42] Connie Smith and Lloyd G. Williams. 2002. New Software Performance AntiPatterns: More Ways to Shoot Yourself in the Foot. In *CMG*. 667–674.
[43] Linhai Song and Shan Lu. 2014. Statistical Debugging for Real-world Performance Problems. In *OOPSLA*. 561–578.
[44] Linhai Song and Shan Lu. 2017. Performance Diagnosis for Inefficient Loops. In *ICSE*. 370–380.
[45] Alexander Wert, Jens Happe, and Lucia Happe. 2013. Supporting Swift Reaction: Automatically Uncovering Performance Problems by Systematic Experiments. In *ICSE*. 552–561.
[46] Frank Wilcoxon. 1992. *Individual Comparisons by Ranking Methods", bookTitle="Breakthroughs in Statistics: Methodology and Distribution*. 196–202.
[47] Sunny Wong, Yuanfang Cai, Giuseppe Valetto, Georgi Simeonov, and Kanwarpreet Sethi. 2009. Design Rule Hierarchies and Parallelism in Software Development Tasks. In *ASE*. 197–208.
[48] Lu Xiao, Yuanfang Cai, and Rick Kazman. 2014. Titan: A Toolset That Connects Software Architecture With Quality Analysis. In *FSE*. 763–766.
[49] Xusheng Xiao, Shi Han, Dongmei Zhang, and Tao Xie. 2013. Context-sensitive Delta Inference for Identifying Workload-dependent Performance Bottlenecks. In *ISSTA*. 90–100.
[50] Guoqing Xu. 2012. Finding Reusable Data Structures. In *OOPSLA*. 1017–1034.
[51] Guoqing Xu, Matthew Arnold, Nick Mitchell, Atanas Rountev, and Gary Sevitsky. 2009. Go with the Flow: Profiling Copies to Find Runtime Bloat. In *PLDI*. 419–430.
[52] Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, Edith Schonberg, and Gary Sevitsky. 2010. Finding Low-utility Data Structures. In *PLDI*. 174–186.
[53] Guoqing Xu and Atanas Rountev. 2010. Detecting Inefficiently-used Containers to Avoid Bloat. In *PLDI*. 160–173.
[54] Guoqing Xu, Dacong Yan, and Atanas Rountev. 2012. Static Detection of Loop-invariant Data Structures. In *ECOOP*. 738–763.
[55] Dacong Yan, Guoqing Xu, and Atanas Rountev. 2012. Uncovering Performance Problems in Java Applications with Reference Propagation Profiling. In *ICSE*. 134–144.
[56] Yibiao Yang, Mark Harman, Jens Krinke, Syed Islam, David Binkley, Yuming Zhou, and Baowen Xu. 2016. An Empirical Study on Dependence Clusters for Effort-Aware Fault-Proneness Prediction. In *ASE*. 296–307.
[57] Tingting Yu and Michael Pradel. 2016. SyncProf: Detecting, Localizing, and Optimizing Synchronization Bottlenecks. In *ISSTA*. 389–400.
[58] Xiao Yu, Shi Han, Dongmei Zhang, and Tao Xie. 2014. Comprehending Performance from Real-world Execution Traces: A Device-driver Case. In *ASPLOS*. 193–206.
[59] Shahed Zaman, Bram Adams, and Ahmed E. Hassan. 2012. A Qualitative Study on Performance Bugs. In *MSR*. 199–208.
[60] Dmitrijs Zaparanuks and Matthias Hauswirth. 2012. Algorithmic Profiling. In *PLDI*. 67–76.