Butterfly Space: An Architectural Approach for Investigating Performance Issues

Yutong Zhao
School of Systems and Enterprises
Stevens Institute of Technology
Hoboken, United States
yzhao102@stevens.edu

Lu Xiao
School of Systems and Enterprises
Stevens Institute of Technology
Hoboken, United States
lxiao6@stevens.edu

Xiao Wang
School of Systems and Enterprises
Stevens Institute of Technology
Hoboken, United States
xwang97@stevens.edu

Zhifei Chen
State Key Lab for Novel Software Technology
Nanjing University
Nanjing, China
chenzhifei@smail.nju.edu.cn

Bihuan Chen
School of Computer Science
Fudan University
Shanghai, China
chenbihuan@gmail.com

Yang Liu
School of Computer Science and Engineering
Nanyang Technological University
Singapore
yangliu@ntu.edu.sg

***Abstract*—Performance issues widely exist in modern software systems. Existing performance optimization approaches, such as dynamic profiling, usually fail to consider the impacts of architectural connections among methods on performance issues.**

This paper contributes an architectural approach, *Butterfly Space* modeling, to investigate performance issues. Each *Butterfly Space* is composed of 1) a seed method; 2) methods in the "upper wing" that call the seed directly or transitively; and 3) methods in the "lower wing" that are called by the seed, directly or transitively. The rationale is that the performance of the seed method impacts and is impacted by all the other methods in the space because of the call relationship. As such, developers can more efficiently investigate groups of connected performance improvement opportunities in *Butterfly Spaces*.

We studied three real-world open source Java projects to evaluate such potential. Our findings are three-fold: 1) If the seed method of a *Butterfly Space* contains performance problems, up to 60% of the methods in the space also contain performance problems; 2) *Butterfly Spaces* can potentially help to non-trivially increase the precision/recall and reduce the costs in identifying performance improvement opportunities, compared to dynamic profiling; and 3) Visualizing dynamic profiling metrics with *Butterfly Spaces* simultaneously help to reveal two typical patterns, namely *Expensive Callee* and *Inefficient Caller*, that are responsible for performance problems and provide insights on where to improve next. We believe that *Butterfly Space* modeling has great potential for investigating performance issues.

***Index Terms*—software performance; software architecture; performance optimization;**

I. Introduction

Software performance is an indicator of how well a software system or component meets its requirements for timeliness [1]. Performance issues can result in long execution time, memory bloat, and even program crash [2]–[5]. Researchers found that users are more likely to switch to competitors' products due to performance bugs than due to other general bugs [4]. To improve software performance, the first and foremost task is to find the performance optimization opportunities. Currently, there are mainly two types of techniques to find performance optimization opportunities: profiling based techniques and root cause based techniques [6].

Profiling based techniques focus on collecting run-time metrics that measure the memory and time consumption when the program is running [7]–[12]. The high run-time metrics help locate code regions (or hot spots) that account for performance degradation. Profiling based approaches have two major limitations. First, it is expensive and sometimes impractical to collect high quality profiling data that reveal performance problems [13], [14]. Developers have been using various off-the-shelf tools to automatically generate test inputs for better results [15]. But many performance problems are triggered by special input, which is hard to obtain by using automated tools. Developers rely on their experience, expertise, and even intuition to discover such cases. In particular, creating high quality test inputs to comprehensively uncover all potential performance problems is almost impossible in large-scale, complex software systems. Second, based on our observation of three real-life software projects (shown in detail in Section VI), not all the methods revised for performance optimization are "hot-spots". In other words, the dynamic metrics of these methods look just like average methods. Relying on profiling-based approaches to locate such methods is almost like random or exhaustive searching.

The other approach is to locate performance issues based on code patterns that cause inefficiency [5], [16]–[20]. For example, the usage of inefficient data structure in the source code can cause performance problems [3], [5], [10], [16]–[18], [20]–[23]. Similarly, inefficient loop [10], [24]–[29], wasted calculation [4], [16], [19], [26], [30]–[34], and inefficient multi-thread synchronization [35]–[38] also cause performance problems. However, these approaches usually only focused on a specific type of localized performance problems that can be fixed by a few lines of simple code fixes. They fail to be applicable to a wider range of performance problems that may be caused by more complicated reasons [6]. For example, the

performance problem of a method could be the cumulative result of a chain of methods it calls directly and transitively. Developers cannot improve its performance without fixing all the other methods on the call chain that contribute to the performance problem.

The common limitation of prior approaches is that they failed to consider the architectural impacts on system performance. In any software system, there are usually more than thousands of methods. There is hardly a method isolated from other methods; instead the methods call each other to deliver more complicated functions. As such, the performance of a method is actually determined by its *callers* and/or *callees*. For example, if a method contains an inefficient loop, its *caller* will also be inefficient. Similarly, if a method has high execution time, it could be because of the frequent invocations from its *callers*. Therefore, in order to locate and fix performance problems more efficiently, developers should be aware of the architectural impacts on performance. Instead of fixing a method that shows high dynamic metrics or contains problematic code patterns in isolation, developers should also inspect and prioritize methods that architecturally connect to it.

In this paper, we propose a new architectural modeling approach based on the static analysis of method calls, named *Butterfly Space* modeling, to help detect and diagnose performance issues. Each butterfly space is a method-level static call graph, where the vertices are a subset of methods from the entire system and the edges are the method calls that are statically extracted. It contains three key parts: 1) a *seed* method, which is the center of the space because of its relationship with other methods in the space; 2) the *upper wing* which contains all the methods that call the *seed* directly or transitively; and 3) the *lower wing*, which contains all the methods called by the *seed* directly or transitively. Essentially, the performance of the methods in a *Butterfly Space* impacts or is impacted by the seed method. Therefore, developers can use *Butterfly Spaces* to locate a group of architecturally connected methods with performance optimization opportunities. Specifically, *Butterfly space* builds a bridge between dynamic profiling and static analysis, since developer can treat a "hot spot" method as the *seed* of a *Butterfly Space*, then trace to and examine the methods in the *lower wing* and the *upper wing*.

We showed the potential of *Butterfly Spaces* to investigate performance problems on three real-world open source Java projects, PDFBox, Avro, and Ivy. We found that if the *seed* method contains performance problems, up to 60% of the other methods in the space also contain performance optimization opportunities. This is much higher than *Butterfly Spaces* where the *seed* does not contain performance problems. Therefore, we gained two insights 1) developers should leverage the architectural connections to examine performance problems and 2) developers should prioritize spaces with a problematic seed over spaces with a normal seed.

In addition, we compared the efficiency of *Butterfly Spaces* modeling with the dynamic profiling in two hypothetical scenarios. First, when developers have limited time, using *Butterfly Spaces* can potentially achieve up to 10% better precision and up to 33% better recall in locating performance optimization opportunities with comparable or even smaller effort, compared to traditional dynamic profiling. Second, when the goal is to retrieve all performance problems, i.e. 100% recall, *Butterfly Spaces* can drastically improve (averagely double) the precision and thus reduce (averagely half) the effort.

Finally, we provided in-depth qualitative study by visualizing dynamic metrics together with *Butterfly Spaces* to show why *Butterfly Spaces* can help identify performance problems. Each *Butterfly Space* usually contains typical patterns, such as *expensive callee* and *inefficient caller*, that connects inefficient methods that should be treated together in performance optimization [6]. The awareness of such patterns provides insights for developers regarding where to fix next.

We believe *Butterfly Spaces* have promising potential to help developers better locate performance optimization opportunities by combining dynamic profiling and static structural analysis.

In summary, this paper makes the following contributions:

- We proposed a new architectural modeling approach, called *Butterfly Spaces* modeling, to bridge the gap between software architecture modeling and performance analysis.
- We conducted evaluation on three real-world projects to demonstrate the potential of using *Butterfly Spaces* to investigate performance improvement opportunities.
- We combine the visualization of dynamic metrics and static dependencies in *Butterfly Spaces* to provide qualitative understanding of two architectural patterns that are responsible for performance problems.

II. BACKGROUND

a) Mining Software Repository: Software repository provides rich data source to uncover interesting and actionable information about software systems and projects. Mining software repository helps to achieve different research goals.

To study performance problems, researchers mine the bug tracking database to obtain ground truth data-set of real-life performance issues in software systems. They usually first use keywords to match issues that are potentially related to performance problems [6], [33], [35], [39]. If the description of an issue report contains keywords such as *fast, slow, speed, too many times, lot of time, too much time*, the issue is related to performance problems. Since the issues containing these keywords are not necessarily truly related to performance problems, manual inspection, verification, and selection of high quality performance issues are conducted after the keyword matching [6]. The selected performance issues are usually used as ground truth data for new approach evaluation or for empirical studies of analyzing the root causes and solutions of real-life performance problems [33], [40]–[43].

Beside mining the bug tracking database, practitioners also mine the version control systems to build the connections between code revisions and the respective issues being solved by the revisions [44]–[47]. According to the prior research, it is a recognized convention that developers tag the issue report ID in a revision message to indicate that this revision is to

fix the issue. This convention has been leveraged to build the connections between issues and revisions. Practitioners leverage the linkage to study the solutions of specific problems, such as fixing performance bugs or security vulnerabilities [46]–[48].

In this paper, we mine the bug tracking database and code repository to obtain the ground truth data-set containing performance problems and optimization. As will be described in detail in Section IV, we select high quality performance issues, link the respective code revisions, and extract the list of revised methods for performance improvements. We consider the list of the methods as the de facto performance optimization opportunities in a system. They serve as the ground to helps us understand the architectural impacts on performance problems.

b) Architecture Root Detection Algorithm: Software architecture is recognized as the most important determinant of various quality attributes of a system [49]. Researchers have focused on modeling and analyzing the descriptive architecture of a system as it has been built to facilitate understanding and maintainability [40]–[43], [50]–[55].

Cai et. al [41] developed an *Architectural Root (ArchRoot)* detection algorithm. The basic idea is that the complexity of a software system cannot be captured by a single view picture. The authors proposed to model the static structural dependencies among files as multiple design spaces. Each design space is a group of architecturally connected files that depend on one leading file. The *ArchRoot* detection algorithm takes two inputs 1) the comprehensive set of design spaces, calculated by using each and every source file in the system as the leading file; and 2) a list of bug-prone files in a system. The output of the *ArchRoot* detection algorithm is a minimal set of design spaces that maximally aggregate the bug-prone files in a system. The output design spaces are called the architectural roots of bug-proneness since they reveal how bug-prone files are architecturally connected to each other.

This paper employs the *ArchRoot* detection algorithm to investigate how and to what extent methods with performance optimization opportunities are architecturally connected with each other. Therefore, we replace the two inputs to the *ArchRoot* detection algorithm by 1) a comprehensive set of *Butterfly Spaces* that are calculated by using each and every method as the seed; and 2) the ground truth list of methods undertaken performance optimization. The output is a minimal list of *Butterfly Spaces* that connects the performance optimization opportunities. Comparing to the original usage of the *ArchRoot* detection algorithm, we made two changes. The architecture modeling granularity changed from the file level to the method level, since the file level is too coarse for performance analysis. The second input, the ground-truth list of methods with performance optimization, helps us to focus on the architectural connections among performance optimization opportunities.

III. Butterfly Space Modeling

We propose a new modeling technique called *Butterfly Spaces* to capture the method-level dependencies in a system as multiple call graphs. Each *Butterfly Space* is a call graph formed by a subset of methods from the entire system and the call relationship among these methods. We will formally define *Butterfly Space* and use the example in Figure 1 to illustrate.

a) Definition: Formally, a *Butterfly Space* is composed of three key parts:

- A seed method: it is the method in the center of a space. All the other methods in the space either call or are called by this seed method, directly or transitively.
- The Upper Wing: it is the groups of methods that call, directly or transitively, the seed method. These methods are separated into *n* layers, depending on the distance to the seed method. The layer number indicates the number of transitive calls from a method to the seed. For example, methods in Layer 1 all directly call the seed method; while methods in Layer 2 call the seed method through methods in Layer 1.
- The Lower Wing: it is the group of methods that are called by, directly or transitively, the seed method. They are also separated into *m* layers based on the distance to the seed. For example, methods in Layer 2 are transitively called by the seed through methods in Layer 1.

Figure 1 is an example *Butterfly Space* calculated from open source project, Avro. The seed of the space is method *ColumnValues.nextValue()*. The *Upper Wing* contains nine methods in three layers. The *Lower Wing* contains fifteen methods in five layers.

In any real life software system, the number of methods is usually counted by thousands (if not more). Using each method as the seed, we can calculate a separate *Butterfly Space*. The motivation and rationale of *Butterfly Space* modeling is that the performance of the seed method is impacted by and also impacts the performance of the methods in the same space, but not methods in another space. For example, in Figure 1, the performance of method *ColumnValues.nextValue()* is determined by the performance of methods in the *Lower Wing*, and influence the methods in the *Upper Wing*. In particular, since *ColumnValues.nextValue()* only directly calls method *InputButter.getValue()*, the execution time of the former is the execution time of its own code plus the time of the latter.

b) Terms and Concepts: We envision that *Butterfly Space* modeling provides a way to analyze performance problems with architectural insights. Methods in a *Butterfly Space* are architecturally connected to the seed method and with each other. By examining a *Butterfly Space*, we can potentially identify a group of connected performance optimization opportunities. We propose to use two measurements, *precision* and *recall* borrowed from information retrieval field [56], to evaluate how effective it is to examine a particular *Butterfly Space* to capture performance optimization opportunities.

The *precision* and *recall* of a *Butterfly Space* is defined as:

- Precision: it is the number of methods with performance fixes in a *Butterfly Space* divided by the total number of methods in this space. The higher the *precision*, the less effort will be wasted in examining this *Butterfly Space* for locating performance optimization opportunities.
- Recall: it is the number of methods with performance fixes in a *Butterfly Space* divided by the total number of

methods with performance fixes in the system. The higher the value, the more comprehensively this *Butterfly Space* captures all the performance optimization opportunities.

The *precision* and recall together describes how relevant and effective it is to examine a particular *Butterfly Space* to capture performance optimization opportunities.

In addition, usually just checking one single *Butterfly Space* will not capture all the performance improvement opportunities. We assume that developers have to check a set of *Butterfly Spaces* to increase the coverage of performance issues. Therefore, we use *Acc. Precision* and *Acc. Recall* to describe the accumulative precision and recall of a set of *Butterfly Spaces*. They are calculated in the same way as the original precision and recall by combining different *Butterfly Spaces*.

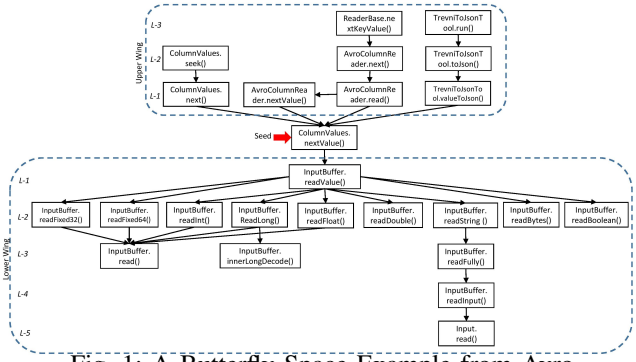

Fig. 1: A Butterfly Space Example from Avro

IV. Study Approach

This paper leverages *Butterfly Space* modeling to analyze performance problems from the architectural perspective. The approach overview is shown in Figure 2. It is composed of three parallel branches: 1) facts mining to extract methods with performance fixes, 2) static analysis to calculate the *Butterfly Spaces* of a system, and 3) dynamic analysis to collect dynamic profiling metrics; and last but not least 4) we combine the three aspects to gain in-depth, qualitative understanding of the relationship between architecture and performance problems.

a) Facts mining: The goal is to mine the de facto list of methods undertaken performance optimization, from the bug tracking database and code repository. The output serves as the ground-truth data for the following analysis.

Firstly, We select performance issues using keywords matching from the bug tracking database. We used keywords including: *fast, slow, perform, latency, throughput, optimize, speed, heuristic, waste, efficient, unnecessary, redundant, too many times, lot of time, too much time*, which are a combination of the prior studies [6], [33], [35]. We manually verified and filtered the matched issues such that the selected issues are truly related to performance problems. Next, we developed a tool *Methods Extractor* to automatically extract the list of methods that were revised to fix the performance issues. The inputs to *Methods Extractor* include 1) performance issues and 2) the code repository. *Methods Extractor* first identifies code revisions that fix the performance issues through the linkage

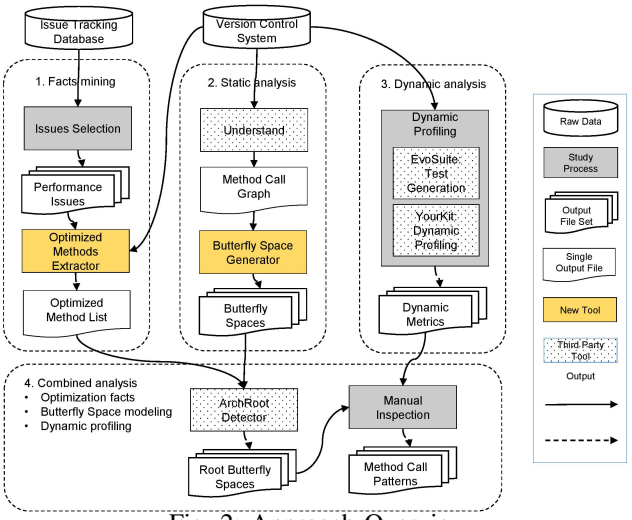

Fig. 2: Approach Overview

between issues and commits, as discussed in Section II. Then it scans the details of each code revision to summarize which methods were changed, which is not directly available from the code repository. The final output is a list of methods with performance fixes.

b) Static analysis: Next, we analyze the static code dependencies to calculate the *Butterfly Spaces*. Firstly, we use a commercial tool, called *Understand* to extract a *Method Call Graph*. The nodes are all the methods in a software system, the edges are the call relationship among the methods. Then, the *Butterfly Space Generator* uses the *Method Call Graph* as the input to generate a comprehensive set of *Butterfly Spaces*, using each method in the system as the *seed*. The *Butterfly Space Generator* is actually a graph traversal algorithm. Given any *seed* method, it performs the broad first traversal to include all the other methods that directly or transitively call or is called by it. Depending on the direction of call relationship, and the distance to the *seed*, the traversed methods are laid out in different layers of the *Upper Wing* and *Lower Wing*.

c) Dynamic analysis: This step collects the dynamic metrics of the methods at run-time. We used the off-the-shelf tools, including *EvoSuite* [15] and *YourKit* [57] to achieve this goal. To increase the code coverage in dynamic profiling, we run both the original testing cases in the project repository as well as the testing cases that are automatically generated by the unit testing tool, *EvoSuite*. Next, we use YourKit to supervise the dynamic execution metrics while running the test cases. We use *Time*, *OwnTime* and *Count* as the execution metric for profiling. *Time* is the entire execution time of a method. *OwnTime* is the execution time of a method excluding the invocation time to other methods. *Count* is the number of invocations to a method. These three metrics are widely used in the existing profiling tools [57]–[59] to identify hot spot methods. To increase the accuracy of the collected dynamic metrics, we ran each project three times and use the average of each metrics as the final dynamic profiling data. The dynamic profiling data were collected on two Intel i7, 4-core, 32 GB memory machines, running at 2.5 GHz.

d) Combined analysis: In this step, we combine the *Butterfly Space* modeling with the dynamic profiling metrics to bridge the gap between software architecture and performance. It is composed of two parts.

In the first part, we use the *ArchRoot* detection algorithm (introduced in Section II) to show that methods with performance optimization opportunities are architecturally connected to each other, instead of being isolated. As such, when developers are seeking performance optimization opportunities, they should not only look at the "hot spot" methods, they should investigate a group of connected methods. The input to the *ArchRoot* detection includes 1) the comprehensive set of *Butterfly Spaces* of a system and 2) the ground truth method list with performance fixes. It will identify a minimal number of *Butterfly Spaces* to aggregate the methods with performance optimization opportunities. The rational of the *ArchRoot* was introduced in Section II. In particular, we applied the *ArchRoot* detection algorithm in two hypothetical settings. In the first setting, we assume that developers only have limited time to examine a limited number of methods, by controlling the upper limit of the total number of methods to be included in the output root spaces. In the second setting, we assume that developers aim to identify all the performance optimization opportunities, by controlling the coverage to be 100%. In section VI, we will show that using *Butterfly Spaces* for analyzing performance issues can potentially beat the dynamic profiling approach by either reducing the cost or improving the precision.

In the second part, we conducted in-depth, qualitative study to investigate why *Butterfly Spaces* can help to analyze performance issues, and what and how typical architectural patterns contribute to performance problems. To achieve this goal, we combine the ground truth method list with performance fixes and the dynamic metrics in the visualization of *Butterfly Spaces*. This helps to immediately reveal the architectural connections among "hot spot" methods and two typical architectural patterns that are responsible for performance problems.

V. Research Questions

We aim at answering three research questions to evaluate the potential of *Butterfly Space* modeling for investigating performance problems.

- **RQ1: If the seed method of a *Butterfly Space* contains performance problems, how likely the other methods in the space also contain performance problems? In particular, how does this compare to butterfly spaces whose seeds are free of performance issues?** We assume that if the seed of a *Butterfly Space* contains performance problems, the other methods in the space are also likely to contain performance problems. The rationale is that the seed method calls or is called by all the other methods in the space, directly or transitively. Therefore, the performance of the seed impacts and is impacted by the other methods in the space. In comparison, if the seed of a *Butterfly Space* is free of performance problems, the rest of the space are less likely to contain performance improvement opportunities. To answer this question, we will examine the *precision* and *recall* of *Butterfly Spaces* whose seed contains performance problem, and compare these measurements with that of *Butterfly Spaces* whose seed does not contain any problems. If the *precision* and *recall* of the former is significantly higher than the latter, it implies that when developers are revising a method for improving performance, he/she should also prioritize and examine the methods in its *Butterfly Space* to capture more optimization opportunities at the same time. We envision this heuristic will help improve the efficiency of performance optimization localization for developers.
- **RQ2: What is the advantage of *Butterfly Space* modeling in identifying performance optimization opportunities compared to dynamic profiling?** Dynamic profiling suffers from two intrinsic limitations in identifying performance optimization opportunities. First, it is expensive and sometimes impractical to generate a large amount of high quality test data to reveal performance problems. Second, dynamic profiling relies on identifying the "hot spot" methods whose dynamic metrics are higher than average methods. However, sometimes performance optimization happens in methods whose dynamic metrics look average. To capture all possible performance optimization opportunities, dynamic profiling is close to random searching and thus is not applicable. This RQ investigates the potential advantages of using *Butterfly Spaces* compared to dynamic profiling approaches.
  We use the *ArchRoot* detection algorithm to identify a set of root *Butterfly Spaces* that aggregate methods undertaken performance optimization. We compare the accumulative *precision* and *recall* of the top root *Butterfly Spaces* with dynamic profiling in two hypothetical settings.
  First, suppose developers can only afford to review a limited number of methods whose dynamic metrics ranks in the top 30% in a project. We compare the accumulative *precision* and *recall* of the top few *Butterfly Spaces* containing comparable or smaller number of methods. If the *precision* and *recall* of *Butterfly Spaces* are higher, it implies that *Butterfly Spaces* have the potential to improve the efficiency of identifying performance improvement opportunities with comparable effort as dynamic profiling.
  Second, suppose developers aim at identifying all the methods that need performance optimization, the dynamic profiling becomes almost like random searching. We will investigate the number of root *Butterfly Spaces* that need to be examined to reach a cumulative recall of 100%. And we will compare the total number of methods included in these spaces, and the cumulative *precision*. If the *precision* is higher, while the number of methods is smaller than using dynamic profiling, it indicates that *Butterfly Spaces* have the potential to help comprehensively capture all the performance improvement opportunities with higher accuracy but less effort.
- **RQ3: What typical architectural patterns that are responsible for performance problems can be revealed when combining *Butterfly Space* with dynamic met-**

TABLE I
Study Subjects

Subject	# Issues				# Methods	
	Total	Keyword	Verified	Solved	Total	Perf
PDFBox	3855	135	93	78	7249	554
Avro	2151	135	113	53	3446	739
Ivy	1522	54	41	24	4755	278
Total	7528	324	247	155	15450	1571

rics? The goal of this RQ is to evaluate the potential of combining *Butterfly Space* with dynamic metrics to provide fundamental and qualitative understanding of whether exist and what are the typical architectural patterns in the root *Butterfly Spaces* that are responsible for the performance problems. This will provide more in-depth insights for developers when using *Butterfly Spaces* to identify architecturally connected performance improvement opportunities with limited dynamic data or historical fixing information.

VI. Study Results

This section presents the subjects of this study, and provide answers to the research questions.

A. Subjects

We selected three real life software projects from the Apache open source community: PDFBox, Avro, and Ivy. The PDFbox library is a Java tool for working with PDF documents. Avro is a remote procedure call and data serialization framework. Ivy is a transitive package manager to resolve complex project dependencies. These subjects are selected due to the following considerations. First, they are in different domains. Performance plays a critical role in all these projects. The goal is to draw general observations across different problem domains. Second, these projects are all well-accepted, successful, and still active Apache open source projects. The source code, version control repository, and bug-tracking systems are all well organized and readily available. This provides high quality data for our study.

We collected a total of 7528 issues reports in the JIRA issues tracking system of the three selected projects up to 2018. There are 324 (4.3%) issues matching performance keywords (Column "Keyword"). The ratio is consistent with prior studies [6], [33], [35]. There are totally 247 manually verified performance issues (Column "Verified"). Totally 155 performance issues can be linked with an accepted fixing solutions in the version control system (Column "Solved"). Then we counted the number of methods that were modified due to performance issues (Column "Perf"), as shown in Table I.

B. Study Results

Next, we will answer the three research questions discussed in Section V.

RQ1: If the seed of a butterfly space contains performance issue, how likely the rest of the space will also contain performance issue? In particular, how does this compare to butterfly spaces whose seeds are free of performance issue?

To answer this question, we calculate the precision of each *Butterfly Space*. The total number of methods in a system determined the total number of *Butterfly Spaces*. We separate all the *Butterfly Spaces* into two mutually exclusive groups: 1) the seed method was fixed for performance improvement, thus contains performance problems; and 2) the seed method is free of performance problems, i.e. never fixed for performance improvement. For the sake of brevity, in the following context, we use PS (Perf-Seed) *Butterfly Space* to the first group; NPS (Non-Perf-Seed) *Butterfly Space* to refer to the second group. Then, we calculate the average and median precision of Perf-Seed *Butterfly Space* and that of Non-Perf-Seed *Butterfly Space* separately.

As shown in Table II, the average precision of PS *Butterfly Space* is from 33% (PDFBox) to 60% (Avro), comparing to the average precision of NPS *Butterfly Space* between 5% (Ivy) to 11% (Avro). The interpretation of the data is two-fold: 1) if the seed contains performance problems, at least one in every three methods in the space also can be improved for performance, thus if a method contains performance problems, the developers should prioritize and examine the methods in its *Butterfly Space*; 2) if the seed is free of performance problems, the other methods are 5 to 8 times less likely to contain performance problems compared to the seed contains performance problems. As such it is inefficient for developers to check NPS *Butterfly Space*. Consistent observations hold when looking at the median presison in the table.

As a particular note, as shown in Table II, the number of PS *Butterfly Space* is drastically smaller than the number of NPS *Butterfly Space*. Therefore, it is potentially biased to directly compare the average and median precision of two groups with unbalanced size. To address this concern, we conducted random sampling on the NPS *Butterfly Space*. In every sampling experiment, we randomly select the same number of NPS *Butterfly Spaces* that is equal to the number of PS *Butterfly Spaces* in each project, and compare the average precision and median. The results are listed in column "Sampled NPS Spaces", the results of the sampling remain consistent with the findings above.

Answer to RQ1: If the seed of a *Butterfly Space* contains performance problems, averagely 33% to 60% of the methods in the space can also be improved for performance. This is about 5 times higher comparing to a *Butterfly Space* with a seed free of any performance problems. The insight is that, when developers are fixing performance problems of a method, they should prioritize and examine other methods in the *Butterfly Space* to fix a group of connected methods all at once, instead of checking and fixing each method individually. In comparison, if a method is free of any performance problems, it is not efficient for the developers to examine its *Butterfly Space* for finding more improvement opportunities.

RQ2: What is the advantage of *Butterfly Space* modeling in identifying performance optimization opportunities compared to dynamic profiling? As discussed in Section V, we answer this question in two hypothetical settings.

TABLE II
Average and Median Precision of Perf-Seed and Non-Perf-Seed Butterfly Spaces

Subject	#PS-Spaces	#NPS-Spaces	PS Spaces		Sampled NPS Spaces		NPS Spaces	
			Avg Prec.	Med Prec.	Avg Prec.	Med Prec.	Avg Prec.	Med Prec.
PDFBox	554	6695	33%	20%	6%	0%	6%	0%
Avro	739	2707	60%	53%	10%	0%	11%	0%
Ivy	278	4477	39%	23%	5%	0%	5%	0%

Note: "PS-Spaces" means *Butterfly Spaces* with the seed method fixed for performance improvement.
"NPS-Spaces" means *Butterfly Spaces* with the seed method free of performance problems.
"Avg Prec." means the average precision of the *Butterfly Spaces*.
"Med Prec." means the median precision of the *Butterfly Spaces*.

Setting 1: Suppose that the developers can only afford the effort to examine the top "hot spot" methods based on dynamic profiling metrics. In real life, developers often face time and resource constraints, as such they can only examine a limited number of methods. This may very likely to happen when the developers have approaching release deadlines or limited number of people working on performance improvements.

In this setting, we want to investigate whether the top few root *Butterfly Spaces* containing fewer or smaller number of methods would achieve higher *precision* and *recall* compared to dynamic profiling. We experimented the top 10%, 20%, and 30% as the threshold for identifying "hot spot" methods. The results are all consistent, thus we will just present the detailed results of the top 30% threshold. Since there are three different types of dynamic metrics, namely, *Time*, *OwnTime*, and *Count* as introduced in Section IV, we selected the metric that achieves the highest *precision* and *recall* in each project to make a fair comparison. When running the *ArchRoot* detection algorithm, we specified a threshold on the precision of each selected *Butterfly Space*, i.e. *Butterfly Space* with precision less than the threshold will not be conducted to the set of root *Butterfly Spaces*. For each project, we selected a threshold that achieves the highest *precision* and *recall* of its set of root *Butterfly Spaces* to make a fair comparison. The goal is to show that there exists a set of root *Butterfly Spaces* that can beat dynamic profiling in this setting, rather than tell developers how to predict the locations of methods with performance optimization opportunities.

The results under this setting are shown in Table III. The first column shows the project name. The second and third columns show the precision and recall of examining the top 30% hot spot methods according to dynamic profiling metrics. Column four and five shows the minimal number of root *Butterfly Spaces* that contain a comparable amount of methods to the top 30% hot spot methods. The last two columns show the cumulative precision and recall of the top root spaces in capturing performance improvement opportunities.

We can make the following observations: First, the top 40 (PDFBox) to 60 (Avro) root spaces contains 20% to 29% of the total number of methods. If the amount of methods to be examined is the proxy of effort, the root *Butterfly Spaces* require smaller (20% in PDFBox and 22% in Ivy) or comparable (29% in Avro) effort compared to checking the top 30% hot spot methods based on the dynamic profiling metrics. Second, the precision of the root *Butterfly Spaces* is slight improved compared to the dynamic profiling approach, with 3% (25%-22% in PDFBox) to 11% (21%-10% in Ivy) improvement. Third, the recall of the root *Butterfly Spaces* is obviously improved compared to the dynamic profiling approach, with 33% (72%-39% in Avro) to 20% (67%-47% in PDFBox) improvement.

Therefore, *Butterfly Spaces* have the potential to help developers to find performance improvement opportunities with higher precision and recall, but with less or comparable effort, compared to dynamic profiling approach.

TABLE III
Top 30% "Hot Spots" vs. Butterfly Spaces

Subject	Top 30% Hot Spots		Top Root Butterfly Spaces			
	Prec.	Rec.	# Spaces	% Methods	Prec.	Rec.
PDFBox	22%	47%	40	20%	25%	67%
Avro	46%	39%	60	29%	52%	72%
Ivy	10%	51%	43	22%	21%	77%

Note: "Prec." means precision.
"Rec." means recall.

Setting 2: Suppose that developers aim at identifying all the methods that can be optimized for maximal performance improvements. A small performance issue can cause significant slowness and even system crash. When developers have enough time and resource, they would like to capture all possible optimization opportunities, i.e. achieving 100% recall. However, developers would always prefer an approach that costs less effort to find all the performance improvement opportunities.

Before answering this question, we first point out that dynamic profiling is close to exhaustive searching when the goal is 100% recall. For example, as shown in Figure 3, the x-axis is the top x% ranking based on the dynamic metrics of different methods, the y-axis is the recall achieved by checking the methods whose dynamic metrics rank in the top x%. The curve shows that dynamic profiling is most efficient when focusing on the higher ranking. For example, when checking the top 30% hot spot methods, developers can achieve almost 40% recall; however, to achieve 100% recall, the developers have to check all the methods. This figure is calculated from Avro, but the other two projects are consistent.

Therefore, we actually compare the effort (i.e. the amount of methods) and the precision of using *Butterfly Spaces* with exhaustive searching. We run the *ArchRoot* detection by setting the coverage to be 100%. As such, the result root *Butterfly Spaces* together can achieve 100% recall. The comparison results are shown in Table IV. The first column is the project name. The second column shows the precision of exhaustive searching, i.e. the percentage of methods revised for performance improvements. The last three columns show the number of root *Butterfly Spaces*, the percentage of methods they contain, and the precision of examining these methods.

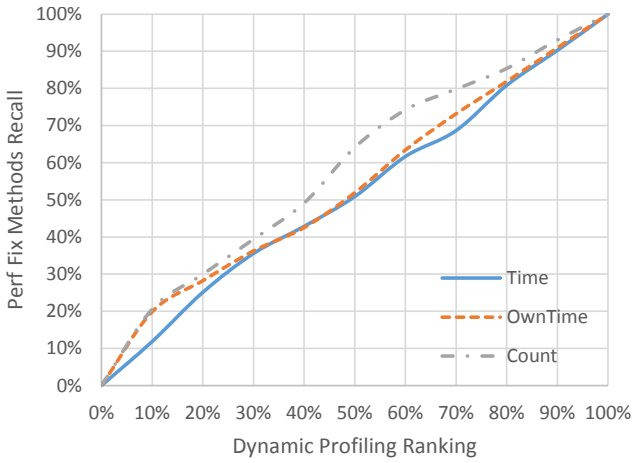

Fig. 3: Avro - Dynamic Metrics Ranking vs. Recall

We can make the following observations. First, developers just need to examine about half (43% in Ivy to 51% in Avro) of the methods, levering *Butterfly Spaces*, to achieve the 100% recall. Although not listed in the table, but exhaustive searching, as suggested by the name, required checking all the methods in a system to achieve 100% recall. Second, the precision of using *Butterfly Spaces* (Column "Prec.") can be significantly improved, about twice, compared to exhaustive searching (Column "ExS Prec."). This improvement is correlated with the reduction of the review effort.

Answer to RQ2: *Butterfly Space* modeling has the potential to improve the efficiency and reduce the costs of finding performance optimization opportunities compared to dynamic profiling. Specifically, the advantage can be reflected in two settings: 1) given limited time and resource, *Butterfly Space* modeling has the potential to increase the precision and recall by up to 11% and 33% respectively, with smaller or comparable effort; 2) *Butterfly Space* modeling has the potential to reach 100% recall in finding performance improvement opportunities with half of the cost and double the precision compared to dynamic profiling.

RQ3: What typical architectural patterns are responsible for performance problems can be revealed when combining *Butterfly Space* with dynamic metrics? In this RQ, we aim to draw in-depth, qualitative understanding of the architectural connections in the root *Butterfly Spaces* that aggregate methods with performance optimization. We found that, by combining the *Butterfly Spaces* visualization with the dynamic metrics, it helps us to facilitate the understanding of the results of RQ1 and RQ2 by seeing the connections among the dynamic metrics of optimized methods. In addition, it helps to reveal the typical architectural patterns that are responsible for the performance problems.

Figure 4 shows the *Butterfly Space* from Avro, the same example as Figure 1, with the dynamic profiling metrics added. In this figure, each rectangle represents a method in the space, the edges represent their call relationship. In side of the rectangle, we included the file name, where the method is

TABLE IV
Profiling vs. Butterfly Spaces for 100% Recall

Subject	ExS Prec.	# Spaces	% Methods	Prec.
PDFBox	8%	32	46%	17%
Avro	21%	61	51%	42%
Ivy	6%	38	43%	14%

Note: "ExS Prec." means the precision using exhaustive searching .

defined, the method name, as well as the three dynamic metrics. The dash lines "- - -" means that the dynamic data cannot be obtained. We use different background in the rectangle to differentiate the characteristics of methods. Slash pattern background indicates that this method is never revised to fix performance problems. The shaded background indicates methods whose dynamic metrics ranked in the top 10% and the top 30% in the system. The methods in this space are used to process a "column-major" file format, which allows better compression and efficient skipping of fields that are not of interest, as to save calculating time.

This visualization helps us to notice the two most expensive paths of call relationship going through the seed method. We emphasized these two paths with thicker edges, and added numbering labels to the edges. Methods along the two paths all have very high *Count*, indicating that they are very frequently called. We conjecture that a small improvement in any of the methods along these paths can result in augmented performance optimization due to the high invocation. In addition, in this space, the most distant methods from the seed are not revised for performance improvements at all.

In addition, we have observed that these two "hot" paths actually reveal typical architectural patterns that are potentially responsible for the performance problems, including the *expensive callee* pattern and the *Inefficient caller* pattern.

a) Expensive callee: A method takes a long time to execute because of the method it calls. Such methods are featured by long total execution time but relatively low own time. The root cause of the slowness in it is due to the invocation to other inefficient methods. In Figure 4, the methods connected by edges with label number (1) actually form a chain of *Expensive Callee*. As shown in Figure 4, the own time of the methods are all much lower compared to the total execution time. And, the execution time and invocation count add up reversing the call chain. Therefore, when developers fix performance problems, they should trace along the call chain of *expensive callee* to reap cumulative performance improvement, instead of fixing an individual method.

b) Inefficient caller: The performance problems of a method is caused by an inefficient caller that frequently calls it, while the method itself does not have much room to improve. Such methods are featured by similar total execution time and own time, indicating the lack of accountability to its callees. As highlighted by label (2), the method on the bottom, *InputBuffer.innerLongDecode()* has the same time and own time, indicating there is no way to improve it by improving its callee chain. While the benefits of fixing the code in itself could be extremly limited, note that the per-invocation time is close to zero already (85ms/833041). The promise to reduce

the execution time of *InputBuffer.innerLongDecode()* is to the caller, who frequently calls it and the frequent invocation is not efficient. Similarly, the method *InputBuffer.readLong()* also has relatively high own time (195 ms out of the total 281 ms in 833091 invocations). Thus, it has limited room to improve just by fixing its own code. To reap greater performance improvement, the developers should trace back the its caller, *InputBuffer.readValue()* to reduce the number of inefficient invocation. Therefore, the three methods connected by edges with label (2) form a chain of *Inefficient Caller* and should be treated together to reap greater improvements.

Answer to RQ3: By combining the *Butterfly Spaces* with the dynamic profiling data, developers can gain in-depth understanding of the architectural impacts on performance issues. We observed two typical architectural patterns, namely *Expensive Callee* and *Inefficient Caller*, that are responsible for performance problems, and also provide insights for developers to identify architecturally connected performance improvement opportunities to reap greater benefits.

VII. Related Work

This section discussed related work in software performance and software architecture.

a) Performance Analysis: A rich body of prior studies have been conducted to understand and classify performance bugs from different perspectives [4], [19], [29], [33], [34], [60], [61]. Some work dedicated to profiling performance specifications from running systems [10], [12], [62], [63]. Profiling tools [57], [58] are widely used to locate hot spot methods that consumes most resources, such as memory and execution time. However, these works have the limitation of heavily relying on test inputs. Thus, some recent works inspected profiling on special test inputs, which are usually either empty or extremely large [12], [19]. This work is most relevant to prior work that focused on tracing execution paths of hot spot methods for root cause analysis of performance issues [7]–[9], [59]. The uniqueness of our approach is that we evaluated the potential of locating performance issues by reasoning based on the static call graph without the dynamic data. In addition, we compared our approach with using dynamic data to identify hot-spots as performance optimization candidates.

Some work focused on analyzing the root causes of performance issues at the fine-grained code-level [64]–[67]. For example, loop-related performance issues have been well-studied [24], [25], [27], [28]. Besides, inefficient data structure [5], [16]–[18] and inefficient synchronization [35]–[38] were also revealed as common root causes of performance issues. In addition, anti-patterns are also used to detect, both statically and dynamically, performance problems [35]–[39], [68]. For example, *Kieker* [69] and *WESSBAS* [70] leverage software testing to automatically capture performance code anti-patterns, such as circuitous treasure hunt and extensive processing. *PerOpteryx* supports the systematic process of evaluating and optimizing component-based software architecture models [71]. These prior work are effective at detecting specific types of code patterns that cause inefficiency. However, they fail to analyze the various performance issues from the big picture of software architecture.

b) Software Architecture: Software architecture refers to the high-level design of the key elements and their interconnections of a system [49]. Numerous prior work focused on the describing, modeling, and analyzing of software architecture. Some work focused on recovering the high-level structure of software architecture based on different criteria and using different techniques [54], [55], [72]–[74]. Some work aimed at analyzing the relationship between software architecture and maintenance quality [43], [50], [75]–[78], aiming at identifying architectural problems that contribute to high maintenance costs.

Despite the large amount of prior work in architecture modeling and analysis, little effort has been invested to study the relationship between software architecture and performance. This study bridges the gap between software architecture modeling and performance analysis. To the best of the knowledge, it is the first work to directly model and visualize software architecture and combine it with dynamic profiling metrics.

VIII. Discussion

In this section, we discuss the limitations and threats to validity of this work, followed by our plan of future work.

a) Limitations: There are three limitations in this work. First, this paper mined the project repository, including bug tracking database and version control systems, to extract the de facto list of methods with performance fixes. Therefore, the root *Butterfly Spaces* that aggregate methods with performance issues currently only works in retrospective, not predictive. But the results indeed showed the potential of leveraging *Butterfly Spaces* for investigating performance problems more efficiently than traditional dynamic profiling. Second, in answering RQ3, we reported two typical architectural patterns, namely, *Expensive Callee* and *Inefficient Caller*, that recur in many *Butterfly Spaces*. Although they provide insights for the architectural influences on performance problems, we are not sure if these two patterns comprehensively cover all possible architectural patterns that are responsible for performance problems. Lastly, this study is based upon 155 performance issues from three open sourced Apache projects, all implemented in Java. We cannot guarantee that the same results will hold for other projects with different characteristics, such as projects in other communities or in other languages. But we envision that consistent observations should still hold for other projects.

b) Threats to Validity: There are two threats to validity we are aware of. First, we used keyword matching for selecting performance issues. We cannot guarantee that all the performance issues are included, because a performance issue may not contain any keywords. And we acknowledge that the selected issues may contain a small amount of false positive due to the limited knowledge about the projects. To

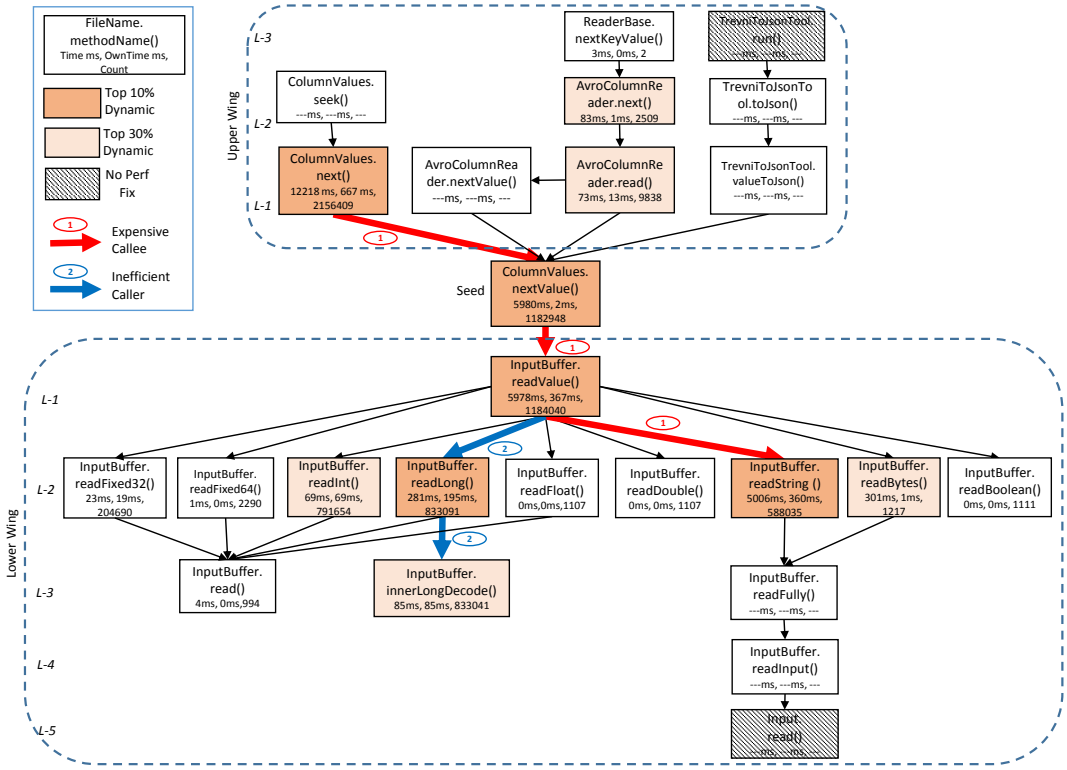

Fig. 4: Butterfly Space with Dynamic Profiling Metrics

best avoid bias, we used a combination of all the keywords used in prior studies, and conducted rigorous manual review to best avoid bias in the selection. Second, we consider all the revised methods in commits for fixing performance issues as containing performance problems. We acknowledge that this may include false positives. Some methods could involve in performance fixes due to accidental reasons. There is no practical way to verify whether each revised method is a root cause to performance problems. We acknowledge that this is an internal threat to validity. However, we believe that the noise introduced due to this treatment should not change the overall conclusions of this paper. Third, due to the difficulties in running all the test cases, we were not able to collect the dynamic metrics of all the methods. To best resolve this, we run both original testing cases in the project repository, as well as the testing cases that are automatically generated by the unit testing tool. Fourth, collecting dynamic execution metrics relies on the running environment and the input of test cases. As these metrics were collected while running test cases on our local computer, the dynamic profiling data will be different when running the same test cases on a different computer or remote server. To cover this limitation, we run the test cases on two computers, each for 3 times, and calculated the average for each metric.

c) Future Work: In our future work, we plan to further exploit the potential of using *Butterfly Space* modeling to deepen the understanding of the relationship between architecture and performance. We are interested in 1) conducting more systematic and comprehensive investigation of possible architectural patterns that are responsible for performance degradation; 2) using *Butterfly Space* modeling predicatively, instead of retrospectively, in locating and analyzing performance problems and optimization opportunities.

IX. Conclusion

To conclude, this paper proposed and implemented a new architectural modeling approach, based on the static analysis of method calls, named *Butterfly Space*, to help detect and diagnose performance issues. We evaluated the potential of *Butterfly Space* to investigate performance issues on three real-world open source Java projects. The results showed that the methods with performance fixes are strongly connected, instead of being isolated. Compared to the current widely-used dynamic profiling ranking technique, *Butterfly Space* modeling has the potential to improve the efficiency and reduce the costs of finding performance optimization opportunities. By combining the *Butterfly Space* with dynamic profiling data, developers can gain in-depth understanding of the architectural impacts on performance issues. Two typical architectural patterns, *Expensive Callee* and *Inefficient caller* are responsible for performance problems, and provide guidance on where to improve next. In summary, *Butterfly Spaces* not only help locate performance problems, but also provide insights for developers to understand the architectural impacts of performance problems.

Acknowledgements

This work was supported in part by the National Science Foundation of the US under grants CCF-1823074.

REFERENCES

[1] Connie U Smith and Lloyd G Williams. *Performance solutions: a practical guide to creating responsive, scalable software*, volume 1. Addison-Wesley Reading, 2002.

[2] Simonetta Balsamo, Antinisca Di Marco, Paola Inverardi, and Marta Simeoni. Model-based performance prediction in software development: A survey. *IEEE Transactions on Software Engineering*, 30(5):295–310, 2004.

[3] Guoqing (Harry) Xu and Atanas Rountev. Precise memory leak detection for java software using container profiling. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008* [3], pages 151–160.

[4] Shahed Zaman, Bram Adams, and Ahmed E Hassan. A qualitative study on performance bugs. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, pages 199–208. IEEE Press, 2012.

[5] Guoqing Xu, Dacong Yan, and Atanas Rountev. Static detection of loop-invariant data structures. In *European Conference on Object-Oriented Programming*, pages 738–763. Springer, 2012.

[6] Zhifei Chen, Bihuan Chen, Lu Xiao, Xiao Wang, Lin Chen, Yang Liu, and Baowen Xu. Speedoo: prioritizing performance optimization opportunities. In *Proceedings of the 40th International Conference on Software Engineering*, pages 811–821. ACM, 2018.

[7] Thomas Ball and James R Larus. Efficient path profiling. In *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 46–57. IEEE Computer Society, 1996.

[8] Evelyn Duesterwald and Vasanth Bala. Software profiling for hot path prediction: Less is more. *ACM SIGOPS Operating Systems Review*, 34(5):202–211, 2000.

[9] James R Larus. Whole program paths. In *ACM SIGPLAN Notices*, volume 34, pages 259–269. ACM, 1999.

[10] Bihuan Chen, Yang Liu, and Wei Le. Generating performance distributions via probabilistic symbolic execution. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016* [10], pages 49–60.

[11] Emilio Coppa, Camil Demetrescu, and Irene Finocchi. Input-sensitive profiling. *ACM SIGPLAN Notices*, 47(6):89–98, 2012.

[12] Rashmi Mudduluru and Murali Krishna Ramanathan. Efficient flow profiling for detecting performance bugs. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016* [12], pages 413–424.

[13] Mark Grechanik, Chen Fu, and Qing Xie. Automatically finding performance problems with feedback-directed learning software testing. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 156–166. IEEE, 2012.

[14] Du Shen, Qi Luo, Denys Poshyvanyk, and Mark Grechanik. Automating performance bottleneck detection using search-based application profiling. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 270–281. ACM, 2015.

[15] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 416–419. ACM, 2011.

[16] Guoqing Xu. Finding reusable data structures. In *ACM SIGPLAN Notices*, volume 47, pages 1017–1034. ACM, 2012.

[17] Changhee Jung, Silvius Rus, Brian P Railing, Nathan Clark, and Santosh Pande. Brainy: effective selection of data structures. In *ACM SIGPLAN Notices*, volume 46, pages 86–97. ACM, 2011.

[18] Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, Edith Schonberg, and Gary Sevitsky. Finding low-utility data structures. In *ACM Sigplan Notices*, volume 45, pages 174–186. ACM, 2010.

[19] Marija Selakovic and Michael Pradel. Performance issues and optimizations in javascript: an empirical study. In *Proceedings of the 38th International Conference on Software Engineering*, pages 61–72. ACM, 2016.

[20] Lixia Liu and Silvius Rus. Perflint: A context sensitive performance advisor for c++ programs. In *Code Generation and Optimization, 2009. CGO 2009. International Symposium on*, pages 265–274. IEEE, 2009.

[21] Guoqing (Harry) Xu. Coco: Sound and adaptive replacement of java collections. In *ECOOP 2013 - Object-Oriented Programming - 27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings*, pages 1–26, 2013.

[22] Ohad Shacham, Martin Vechev, and Eran Yahav. Chameleon: adaptive selection of collections. In *ACM Sigplan Notices*, volume 44, pages 408–418. ACM, 2009.

[23] Guoqing Xu and Atanas Rountev. Detecting inefficiently-used containers to avoid bloat. In *ACM Sigplan Notices*, volume 45, pages 160–173. ACM, 2010.

[24] Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. Toddler: Detecting performance problems via similar memory-access patterns. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 562–571. IEEE Press, 2013.

[25] Adrian Nistor, Po-Chun Chang, Cosmin Radoi, and Shan Lu. Caramel: Detecting and fixing performance problems that have non-intrusive fixes. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, volume 1, pages 902–912. IEEE, 2015.

[26] Linhai Song and Shan Lu. Performance diagnosis for inefficient loops. In *Proceedings of the 39th International Conference on Software Engineering*, pages 370–380. IEEE Press, 2017.

[27] Oswaldo Olivo, Isil Dillig, and Calvin Lin. Static detection of asymptotic performance bugs in collection traversals. In *ACM SIGPLAN Notices*, volume 50, pages 369–378. ACM, 2015.

[28] Monika Dhok and Murali Krishna Ramanathan. Directed test generation to detect loop inefficiencies. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 895–907. ACM, 2016.

[29] Linhai Song and Shan Lu. Statistical debugging for real-world performance problems. In *ACM SIGPLAN Notices*, volume 49, pages 561–578. ACM, 2014.

[30] Luca Della Toffola, Michael Pradel, and Thomas R Gross. Performance problems you can fix: A dynamic analysis of memoization opportunities. In *ACM SIGPLAN Notices*, volume 50, pages 607–622. ACM, 2015.

[31] Khanh Nguyen and Guoqing Xu. Cachetor: Detecting cacheable data to remove bloat. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 268–278. ACM, 2013.

[32] Ajeet Shankar, Matthew Arnold, and Rastislav Bodík. Jolt: lightweight dynamic analysis and removal of object churn. In *Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-23, 2008, Nashville, TN, USA* [32], pages 127–142.

[33] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. *ACM SIGPLAN Notices*, 47(6):77–88, 2012.

[34] Adrian Nistor, Tian Jiang, and Lin Tan. Discovering, reporting, and fixing performance bugs. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 237–246. IEEE Press, 2013.

[35] Michael Pradel, Markus Huggler, and Thomas R Gross. Performance regression testing of concurrent classes. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 13–25. ACM, 2014.

[36] Tingting Yu and Michael Pradel. Syncprof: Detecting, localizing, and optimizing synchronization bottlenecks. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 389–400. ACM, 2016.

[37] Jeff Bogda and Urs Hölzle. Removing unnecessary synchronization in java. In *ACM SIGPLAN Notices*, volume 34, pages 35–46. ACM, 1999.

[38] Erik Ruf. Effective synchronization removal for java. In *ACM SIGPLAN Notices*, volume 35, pages 208–218. ACM, 2000.

[39] Juan Pablo Sandoval Alcocer, Alexandre Bergel, and Marco Tulio Valente. Learning from source code history to identify performance failures. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, pages 37–48, 2016.

[40] Lu Xiao, Yuanfang Cai, and Rick Kazman. Titan: A toolset that connects software architecture with quality analysis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 763–766. ACM, 2014.

[41] Yuanfang Cai, Lu Xiao, Rick Kazman, Ran Mo, and Qiong Feng. Design rule spaces: A new model for representing and analyzing software architecture. *IEEE Transactions on Software Engineering*, 2018.

[42] Lu Xiao. *Bridging the Gap between Software Architecture and Maintenance Quality*. Drexel University, 2016.

[43] Lu Xiao, Yuanfang Cai, and Rick Kazman. Design rule spaces: A new form of architecture insight. In *Proceedings of the 36th International Conference on Software Engineering*, pages 967–977. ACM, 2014.

[44] Iman Keivanloo, Christopher Forbes, Aseel Hmood, Mostafa Erfani, Christopher Neal, George Peristerakis, and Juergen Rilling. A linked data platform for mining software repositories. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, pages 32–35. IEEE Press, 2012.

[45] Adrian Bachmann, Christian Bird, Foyzur Rahman, Premkumar Devanbu, and Abraham Bernstein. The missing links: bugs and bug-fix commits. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 97–106. ACM, 2010.

[46] Rongxin Wu, Hongyu Zhang, Sunghun Kim, and Shing-Chi Cheung. Relink: recovering links between bugs and changes. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 15–25. ACM, 2011.

[47] Andrea De Lucia, Fausto Fasano, Rocco Oliveto, and Genoveffa Tortora. Recovering traceability links in software artifact management systems using information retrieval methods. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 16(4):13, 2007.

[48] Christopher S Corley, Nicholas A Kraft, Letha H Etzkorn, and Stacy K Lukins. Recovering traceability links between source code and fixed bugs via patch analysis. In *Proceedings of the 6th International Workshop on Traceability in Emerging Forms of Software Engineering*, pages 31–37. ACM, 2011.

[49] Len Bass, Paul Clements, and Rick Kazman. *Software architecture in practice*. Addison-Wesley Professional, 2003.

[50] Ran Mo, Yuanfang Cai, Rick Kazman, and Lu Xiao. Hotspot patterns: The formal definition and automatic detection of architecture smells. In *Software Architecture (WICSA), 2015 12th Working IEEE/IFIP Conference on*, pages 51–60. IEEE, 2015.

[51] Allan Terry, Frederick Hayes-Roth, Lee Erman, Norman Coleman, Mary Devito, George Papanagopoulos, and Barbara Hayes-Roth. Overview of teknowledge's domain-specific software architecture program. *SIGSOFT Softw. Eng. Notes*, 19(4):68–76, October 1994.

[52] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, and Gregory Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Trans. Softw. Eng.*, 21(4):314–335, April 1995.

[53] Jonathan Aldrich, Craig Chambers, and David Notkin. ArchJava: Connecting software architecture to implementation. In *Proc. 24th*, pages 187–197, May 2002.

[54] Kenichi Kobayashi, Manabu Kamimura, Koki Kato, Keisuke Yano, and Akihiko Matsuo. Feature-gathering dependency-based software clustering using dedication and modularity. In *Proceedings of the 2012 IEEE International Conference on Software Maintenance (ICSM)*, ICSM '12, pages 462–471, Washington, DC, USA, 2012. IEEE Computer Society.

[55] J. Misra, K. M. Annervaz, V. Kaulgud, S. Sengupta, and G. Titus. Software clustering: Unifying syntactic and semantic features. In *2012 19th Working Conference on Reverse Engineering*, pages 113–122, Oct 2012.

[56] Gerard Salton and Michael J McGill. Introduction to modern information retrieval. 1986.

[57] LLC Yourkit. Yourkit profiler, 2003.

[58] Eun-young Cho. Jprofiler: Code coverage analysis tool for omp project. Technical report, Technical Report: CMU 17-654 & 17, 2006.

[59] André Van Hoorn, Jan Waller, and Wilhelm Hasselbring. Kieker: A framework for application performance monitoring and dynamic software analysis. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, pages 247–248, 2012.

[60] Sebastian Baltes, Oliver Moseler, Fabian Beck, and Stephan Diehl. Navigate, understand, communicate: How developers locate performance bugs. In *Empirical Software Engineering and Measurement (ESEM), 2015 ACM/IEEE International Symposium on*, pages 1–10. IEEE, 2015.

[61] Yepang Liu, Chang Xu, and Shing-Chi Cheung. Characterizing and detecting performance bugs for smartphone applications. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1013–1024. ACM, 2014.

[62] Marc Brünink and David S Rosenblum. Mining performance specifications. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 39–49. ACM, 2016.

[63] Charlie Curtsinger and Emery D Berger. C oz: finding code that counts with causal profiling. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 184–197. ACM, 2015.

[64] Lu Fang, Liang Dou, and Guoqing (Harry) Xu. Perfblower: Quickly detecting memory-related performance problems via amplification. In *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic* [64], pages 296–320.

[65] Adriana E Chis, Nick Mitchell, Edith Schonberg, Gary Sevitsky, Patrick OSullivan, Trevor Parsons, and John Murphy. Patterns of memory inefficiency. In *European Conference on Object-Oriented Programming*, pages 383–407. Springer, 2011.

[66] Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, and Gary Sevitsky. Software bloat analysis: finding, removing, and preventing performance problems in modern large-scale object-oriented applications. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pages 421–426. ACM, 2010.

[67] Erik Altman, Matthew Arnold, Stephen Fink, and Nick Mitchell. Performance analysis of idle programs. In *ACM Sigplan Notices*, volume 45, pages 739–753. ACM, 2010.

[68] Catia Trubiani, Achraf Ghabi, and Alexander Egyed. Exploiting traceability uncertainty between software architectural models and performance analysis results. In *European Conference on Software Architecture*, pages 305–321. Springer, 2015.

[69] Catia Trubiani, Alexander Bran, André van Hoorn, Alberto Avritzer, and Holger Knoche. Exploiting load testing and profiling for performance antipattern detection. *Information and Software Technology*, 95:329–345, 2018.

[70] Christian Vögele, André van Hoorn, Eike Schulz, Wilhelm Hasselbring, and Helmut Krcmar. Wessbas: extraction of probabilistic workload specifications for load testing and performance predictiona model-driven approach for session-based application systems. *Software & Systems Modeling*, 17(2):443–477, 2018.

[71] Axel Busch, Dominik Fuchß, and Anne Koziolek. Peropteryx: Automated improvement of software architectures. In *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*, pages 162–165. IEEE, 2019.

[72] Kata Praditwong, Mark Harman, and Xin Yao. Software module clustering as a multi-objective search problem. *IEEE Trans. Softw. Eng.*, 37(2):264–282, March 2011.

[73] A. Corazza, S. Di Martino, V. Maggio, and G. Scanniello. Investigating the use of lexical information for software system clustering. In *2011 15th European Conference on Software Maintenance and Reengineering*, pages 35–44, March 2011.

[74] U. Erdemir, U. Tekin, and F. Buzluca. Object oriented software clustering based on community structure. In *2011 18th Asia-Pacific Software Engineering Conference*, pages 315–321, Dec 2011.

[75] Lu Xiao, Yuanfang Cai, Rick Kazman, Ran Mo, and Qiong Feng. Identifying and quantifying architectural debt. In *Proceedings of the 38th International Conference on Software Engineering*, pages 488–498. ACM, 2016.

[76] Sunny Wong, Yuanfang Cai, Miryung Kim, and Michael Dalton. Detecting software modularity violations. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 411–420. IEEE, 2011.

[77] Rick Kazman, Yuanfang Cai, Ran Mo, Qiong Feng, Lu Xiao, Serge Haziyev, Volodymyr Fedak, and Andriy Shapochka. A case study in locating the architectural roots of technical debt. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, volume 2, pages 179–188. IEEE, 2015.

[78] Nico Zazworka, Clemente Izurieta, Sunny Wong, Yuanfang Cai, Carolyn Seaman, Forrest Shull, et al. Comparing four approaches for technical debt identification. *Software Quality Journal*, 22(3):403–426, 2014.