PerfLens: A Data-Driven Performance Bug Detection and Fix Platform

Spandan Garg
spgarg@microsoft.com
Microsoft
Vancouver, BC, Canada

Roshanak Zilouchian Moghaddam
rozilouc@microsoft.com
Microsoft
Redmond, WA, USA

Neel Sundaresan
neels@microsoft.com
Microsoft
Redmond, WA, USA

Chen Wu
wuche@microsoft.com
Microsoft
Shanghai, China

Abstract

The wealth of open-source software development artifacts available online creates a great opportunity to learn the patterns of performance improvements from data. In this paper, we present a data-driven approach to software performance improvement in C#. We first compile a large dataset of hundreds of performance improvements made in open source projects. We then leverage this data to build a tool called PerfLens for performance improvement recommendations via code search. PerfLens indexes the performance improvements, takes a codebase as an input and searches a pool of performance improvements for similar code. We show that when our system is further augmented with profiler data information our recommendations are more accurate. Our experiments show that PerfLens can suggest performance improvements with 90% accuracy when profiler data is available and 55% accuracy when it analyzes source code only.

CCS Concepts: • **Computer systems organization**; • **Computing methodologies** → *Machine learning*; • **Software and its engineering** → **Software performance**;

Keywords: Software Performance, Machine Learning

ACM Reference Format:
Spandan Garg, Roshanak Zilouchian Moghaddam, Neel Sundaresan, and Chen Wu. 2021. PerfLens: A Data-Driven Performance Bug Detection and Fix Platform. In *Proceedings of the 10th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP '21), June 22, 2021, Virtual, Canada.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3460946.3464318

SOAP '21, June 22, 2021, Virtual, Canada

ACM ISBN 978-1-4503-8468-1/21/06…$15.00
https://doi.org/10.1145/3460946.3464318

1 Introduction

Inefficient code sequences can cause significant performance degradation and resource waste, referred to as performance bugs. Detecting and fixing performance bugs is important as these bugs can lead to poor user experience, reduced throughput, increased latency, and wasted resources. However, performance bugs are hard to detect as they don't cause system failure, are dependent on user input, and in some cases, only manifest with specific inputs. These bugs can be introduced even by expert developers in well known applications [6, 9, 10, 14] and can propagate quickly due to prevalence of code re-use. Even when performance bugs are detected, they are more difficult to fix than non-performance bugs [11, 15]. Therefore, better tool support is needed for fixing performance bugs.

In recent years, a variety of techniques have been proposed to detect performance issues, majority of which rely either on static code analysis or application profiling [3, 5]. Profiling is an important analysis technique, where an application is analyzed dynamically to determine its space and time complexities and usage of its instructions that reveal performance bottlenecks [16]. However, profiling is not always available and static analysis only detects a small subset of bugs based on pre-defined rules. With the recent widespread availability of open source repositories, it has become possible to use data-driven techniques to discover patterns of performance improvements. In this paper, we present techniques that use machine learning to collect performance improvements. We then describe PerfLens, a novel performance improvement recommendation system that leverages collected data to suggest improvements in C# code. PerfLens operates in two modes: (i) *basic source code analysis mode* that only analyzes source code and (ii) *combined profiler data and source code analysis mode* that uses both profiler data and source code to suggest improvements. Our evaluation on five open source and five proprietary projects shows that PerfLens can suggest fairly accurate performance improvements especially when used with profiler data.

2 Performance Improvement Recommendation Engine

In this section, we first describe how we leverage machine learning to collect performance PRs (Pull Requests) from open source projects. We then explain details of our performance improvement recommendation algorithm. Finally, we explain how we leverage profiler data to further prune and rank our recommendations.

2.1 Collecting Performance Improvement Data

Given the complexity and variation of performance issues, a large number of examples are necessary to make valid recommendations for different scenarios. We used a semi-supervised data gathering approach to gather relevant PRs from Github. We first gathered 6 million PRs from 4559 C# repositories with >100 stars. To identify performance PRs, similar to prior work [2, 11], we started by selecting 100 repositories from our data set. For these repositories, we found all the PRs that contained a performance-related keyword ("perf", "performance", "latency", "slow" etc.) in their title, description, etc. We then manually labeled these PRs to be performance and non-performance related, which yielded 60 performance PRs and 120 non-performance PRs. We used the set of labeled PRs to train a binary classifier to detect whether or not a PR is performance related. From each PR, we collected the title, description, commit texts, and the before and after code in each modified file. We then extracted two sets of features: text features and source code features.

Source Code Features To get a represent the code changes in a PR, we tokenized the before and after code in each modified file. We then converted the extracted sequences of tokens to a vector representation by taking the differences in the before and after raw counts using bag-of-words [8].

Text Features To get a feature representation of the PR title, description and commit messages, we transformed the texts to lower-case, removed all special characters and common stop-words, and lemmatized the text. Unlike source code features, we used TF-IDF (term-frequency inverse-document-frequency) scores to vectorize text features.

We first trained a Random Forest classifier using the above features and our labeled data, to detect whether or not a PR is performance related. We ran the classifier on our large PR pool in multiple iterations to classify unlabeled PRs. On each iteration, we re-trained the classifier on the cumulative labeled data from past iterations. We then leveraged the classifier we trained to classify the rest of the unlabeled PRs, taking the high confidence results as positive samples and low confidence results as negative samples based on a threshold, incorporating them into our training data for future iterations. This allowed our model to learn new performance improvement code and text patterns and find new PRs on every iteration. After 10 iterations, we had collected 1350 performance PRs, which included 3022 commits.

2.2 Performance Improvement Recommendation

We used the PRs we gathered to build a performance improvement recommendation engine by building an index of recommendations based on the collected PRs. For every modified function in each PR, we parse the *before* and *after* version of the function and create a simplified parse tree and featurize the differences in the simplified parse trees. We then create an index out of all the differences in the features collected from the before and after snippets. Given a code snippet as input query, we return a set of recommended code improvements from the index. Below we explain the featurization and recommendation steps in detail.

2.2.1 Featurization. We first convert each parse tree into a simplified parse tree (SPT) as described in previous work in code search [7]. We then extract a set of features from each SPT. A key requirement of these features is that two similar code snippets, should yield a similar set of features.

For each SPT, we extract two kinds of features: Structural and Performance Features.

Structural Features Similar to [7], we define the following features for all non-keyword tokens in the SPT:

- *A Token Feature* which is the label representing a non-keyword token.
- *Parent Features* of the form (token, i_1, parent), (token, i_2, grand-parent), and (token, i_3, great-grand-parent), where i_1 is the index of token in its parent's list of children, i_2 is the index of parent in grand-parent's list of children and i_3 is the index of grand-parent in great-grand-parent's list of children.
- *Sibling Features* of the form (token, next token) and (previous token, token) where previous token and next token are the previous and next non-keyword tokens tokens relative to token with no intervening non-keyword tokens.
- *Variable Usage Features* is only defined for tokens representing local variables in the program and is of the form (Context(previous usage token), Context(token)) and (Context(token), Context(next usage token)). For a non-keyword token n, Context(n) is defined as:
  - Tuple (i, p), where p is parent of n and i is index of n in p's list of children, if label of p is not #.#.
  - First non-keyword token after n in program that is not a local variable.

Performance Features These are designed to capture performance related characteristics of a C# code snippet.

- *Chain Invocation Feature* is of the form (function token, next function token), where these tokens are ordered pairs of consecutive methods that are chained in a single statement. For instance, for the snippets in Figure 1, we would extract the feature `(Where, FirstOrDefault)` because they contain consecutive calls to these methods within a single statement.

```
foreach (var client in Clients) {
    var nodes = client.GetNodes();
    INode parent = nodes.Where(x => x.GetNodeId() == client.Id).FirstOrDefault();
    // ...
}
```

```
// Suggestion
private void GetCurrentFolder(bool create)
{
    foreach(var p in m_path.Split(new string[] {"/"}))
    {
        var el = PagedFileListResponse(id).Where(x => x.Name == p).FirstOrDefault();
        var el = PagedFileListResponse(id).FirstOrDefault(x => x.Name == p);
        // ...
    }
}
```

Figure 1. Sample code snippet followed by a performance improvement suggestion by PerfLens. The suggestion to directly call `FirstOrDefault` with the lambda function and omit the call to `Where`, which is more efficient. This is recommended because both code snippets are structurally similar and have a common Chain Invocation feature `(Where, FirstOrDefault)`. The suggestion itself was collected by our Performance PR mining model and was extracted from a pull-request[1] to the open source C# project, Duplicati.

- *Nested Function Call Feature* is of the form (function token, nested function token), where first token is a non-keyword token associated with a function and the second is the non-keyword token for a function that's called from the argument list of the first function.
- *Repeat Function Call Feature* is of the form (function token, invocation expression) repeated as often as the max number of times the same invocation i.e. function call with same parameters is found on any path through the Control Flow Graph within the method.
- *Definition & Use Feature* is of the form (function token, usage token), where the first token is the identifier of the function call in the most recent assignment to a variable and the second token is the identifier of the variable's member function/property that's accessed.
- *Member Access Feature* is only defined for variable tokens in the program. It is of the form (usage token, next usage token), where the two tokens are the identifiers associated with the variable members (function, properties, etc.) that are accessed in these two usages.

2.2.2 Recommendation Algorithm.

For each function f with before code f_b and after code f_a, let $F(f_b)$ and $F(f_a)$ represent the multi-set of features extracted from before and after code. Then the difference between them, $D(f)$ is defined as $D(f) = \{ x \in F(f_b) \mid m_b(x) < m_a(x) \}$, where $m_b(x)$, $m_a(x)$ represent the frequency of feature x in multi-sets $F(f_b)$ and $F(f_a)$ respectively. If feature x is not present in a multi-set, function m returns 0. $D(f)$ represents code that was modified by the developer in before code. We then create an index where each document is the pair (f, $D(f)$).

To make recommendations for a given query q and each function, f_q within the query, we extract features $F(f_q)$ using featurization explained above. We then filter out documents that don't have any performance features in common with the query function. This initial filtration helps us limit our search to only the PR's that fix issues that may be present in the query function. We then search the filtered index for the best possible recommendations with each document, d, is scored as follows: $Score(f_q, d) = \frac{|d \cap F(f_q)|}{|d|}$ If we find documents with a high similarity score(> 0.8), we report the function as buggy and return a ranked list of suggestions. Using this algorithm, Figure 1 shows a suggestion to simplify the LINQ [12] query in the boxed snippet using the transformation shown in the snippet below i.e. remove the call to `Where()` and move its contents to `FirstOrDefault()`. This index entry is selected because both the code snippet and the document associated with this suggestion have Chain Invocation feature `(Where, FirstOrDefault)` in common and are structurally similar to one another.

2.3 Pruning Recommendations Based on Profiler Data

Many commonly used software systems and services leverage performance profiling to monitor their service performance. When profiler data is available, we use it to further prune our list of performance recommendations. We first analyze profiler data to determine hot code paths and recommend changes only for functions on a hot code path.

2.3.1 Hot Code Path Function Detection.

We find hot code path functions at every depth of the stack trace individually by considering functions whose CPU or memory usage is above 2.5 standard deviations from the average at

[1] https://github.com/duplicati/duplicati/pull/3410

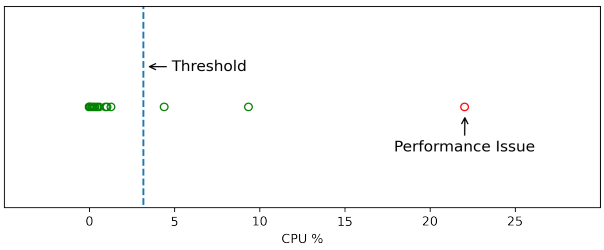

Figure 2. Scatter plot showing stack depth based outlier detection. Each point in the scatter plot corresponds to the CPU usage of a stacks at the same depth. A dotted line shows the outlier threshold of 2.5 standard deviations from mean. The outlier stack (in red) was later confirmed to be a performance bug by the project owners.

its depth. The majority of performance bugs are expected to be among these functions. Figure 2 shows CPU usage of various functions at one depth of a stack trace. The red dot represents a hot path function later confirmed to be a bug.

2.3.2 Performance Improvement Recommendations on Hot Code Paths.

Instead of using all the functions in the application as queries, we only provide recommendations for the hot code path functions. We use each detected function as a query to our recommendation algorithm explained above.

3 PerfLens

Based on the approach discussed, we built a software system called PerfLens. PerfLens is a performance bug detection and fix suggestion platform that suggests performance improvements when possible. When profiler data is available, PerfLens performs a combined source code and profiler data analysis. In absence of profiler data, the system uses source code analysis to detect performance bugs and fix suggestions. Details for each component are described below.

Source Code Analyzer Our source code analyzer takes the entire source code repository as the input. It parses every file and creates a simplified parse tree for each class found within the file. For each method within a class, the feature extraction component extracts features as explained in 2.2.1. Features for each method are then used as a query against the search index and if a matching document with a high score is found based on our search algorithm described in 2.2.2, we report the PR associated with the match as code suggestions to help developers fix the issue.

Profiler Data Analyzer When profiler data is available, PerfLens only analyzes functions on the hot-code path.

4 Evaluation

We conducted two experiments of PerfLens. The goal of the first experiment was to gauge the accuracy of our basic source code analysis capabilities. The second experiment focused on the effectiveness of our combined profiler data and source code analysis techniques.

4.1 Experiment 1: Basic Source Code Analysis

We ran PerfLens basic source code detection analysis on five popular C# open source projects. A performance expert analyzed and verified the correctness of our results. Table 1 show the number of issues that we found in each project and the accuracy of the detection.

On average the accuracy of PerfLens suggestions was 55%. The majority of false positives were caused by our assumption that performance changes are confined to individual methods, when in practice performance changes often span multiple methods or classes. When creating the index, we split performance PRs into methods and assume that the changes made in the before and after code can be used as self-contained recommendations. In some cases, however, these changes do not completely capture the intent of the performance changes made by the developer as they might be part of a larger code change such as a refactoring or a systematic change across the file. As a result, these documents may not contain a complete performance change, but might still be recommended for a given query. This was the cause for most of the false positives seen in *AngleSharp*.

Some false positives were due to mismatch in scope of a variable between the suggestion and the query function in cases where scoped control structures (if-statement, for-loop) were involved. We believe that this may be because some performance features such as *Member Access* or *Definition & Use* features lack information about the control structure or variable scope. Similar to prior work [7], we omitted certain details such as scope information from our features on purpose because a feature set that is too specific may cause PerfLens to only suggest changes when it finds identical code and can impact the variety of recommendations.

4.2 Experiment 2: Combined Profiler Data and Source Code Analysis

For the second experiment, we picked five active proprietary projects with profiler data. We then ran PerfLens analysis on these projects. Table 2 shows the number of issues found in each project alongside the accuracy of the detection.

Similar to experiment 1, many of the false positives observed in project E, were caused by the similarity of a hot code path function to a function in a PR that was part of a large performance change that spanned multiple functions but was not a valid performance change by itself.

4.3 Overall Results

In both experiments, PerfLens was able to detect and recommend fixes for a wide range of performance problems. For instance, some suggested changes addressed issues related to the use of .NET functionality. These included suggestions like using the `Count` property when checking whether a

Table 1. Number of performance suggestions in each project

Project Name	# of functions	# of suggestions	% of correct suggestions
Serilog	1325	6	83%
octokit.net	11441	4	50%
VsVim	8980	5	60%
FluentCassandra	3949	12	58%
AngleSharp	6858	9	22%

Table 2. Number of performance suggestions in each project

Project Name	# of functions	# of bottleneck functions	# of suggestions	% of correct suggestions
A	9522	3	1	100%
B	9631	21	2	100%
C	8196	34	1	100%
D	170179	20	2	100%
E	26359	15	8	50%

`List` is empty instead of `Any()` that has the added cost of enumeration, simplifying LINQ queries similar to suggestion in Figure 1, replacing multiple string concatenations with `StringBuilder` to reduce string allocations, replacing `ContainsKey()` with `TryGetValue()` to reduce the number of accesses to a `Dictionary`, merging `Any()` and `First()` calls to a single call to `FirstOrDefault()`, removing unnecessary `ToList()` calls to reduce `List` allocations, etc. PerfLens was also able to make more general performance suggestions for optimizations like caching the results of repeated method calls with the same parameters, hoisting allocations to an outer scope to prevent repeated allocations and specifying the size when allocating a `List` or a `Dictionary` when an exact count is available beforehand.

We found that, although our basic source code detection technique is promising, the combined approach performs much better. While the average accuracy of the basic source code approach is 55%, the average accuracy of our profiler data and source code analysis approach was 90%. However, since profile data is not always readily available, we built PerfLens to be able to operate in both modes. This will allow any C# application to benefit from our recommendations.

5 Discussion

Our experiment results validate the quality of the PR data we gathered. Similar to [2, 13], our dataset can enable future studies on performance bugs and training of various machine learning models for performance bug detection. Similarly, our semi-supervised data collection methodology is improving the state of the art in performance bug fix data collection [2, 11], which are based on simple keyword search. With our methodology, we were able to collect four times more data (1350 PRs that have 3022 commits) in comparison to prior work [2] (733 commits). Our methodology can also inform future data collections targeting other types of high impact bugs such as security or reliability bugs.

For the purpose of this paper, we decided to focus on recommending function level performance improvements. This enables us to capture a wide variety of performance improvements as described in the section 4.3. Therefore, we are contributing to the state of the art in performance improvements that are focused on single type of performance improvements [1, 4, 11, 17–20]. However, we cannot yet understand or suggest improvements that span multiple functions. Future work needs to explore ways of capturing more complex multi-method/class improvements.

Our combined profiler and source code detection approach relies on profile information to select functions for source code detection to focus on. However, previous work has shown that often the root-cause function may not get ranked by the profiler at all [15]. Even if the functions appear in the profile report, finding the root-cause function from the call stack is not a trivial task. Therefore, to ensure that the recommended improvements are targeting underlying performance issues, future studies should be performed to explore alternative ways of selecting target functions.

Our experiment results were verified by a performance expert, however previous studies have shown that the decision of whether or not to fix a bug can often be project specific and ultimately rests with the developer. Therefore, further studies are required to investigate whether the project owners will leverage our recommendations in action.

6 Related Work

Majority of performance detection tools focus on detecting a specific type of performance bug. For instance, tools have been developed to detect runtime bloat [4, 18, 20], low-utility data structures [19], database related anti-patterns [1], and

inefficient loops [11, 17]. Our tool extends prior work on performance bug detection by developing a system that focuses on diagnosing general performance problems considering both runtime symptoms and source code features.

Prior work has explored various profiling techniques. For instance, Xu et al [18] introduced copy profiling, which summarizes the program's runtime activity in terms of chains of data copies. Similarly, Yan et al [21] presented a profiling method based on reference propagation graph, which represent creation, assignment, and operations on object references. Coppa et al presented a profiler that automatically measures the performance of routines as a function of input size [3]. Research have also explored various ways of focusing the analysis of profiling data. For instance, Han et al [5] used a clustering mechanism based on domain-specific characteristics of program-execution traces to help performance analysts focus their analysis when dealing with millions of profile stack traces. Our system is original relative to prior work because it leverages both profiler data and source code features to suggest performance improvements.

7 Conclusions

Detecting and fixing performance bugs is both important and challenging. Our work makes three contributes to improve the state of the art of detecting and diagnosing performance bugs. First, we presented a machine learning based approach to automatically collect performance PRs from open source software. This dataset can inform future studies and applications in this space. Second, we built a novel system that uses source code and profiler data to suggest performance improvements for C# applications. Finally, we reported results from two evaluations showing that combining profiler data with source code analysis can improve the accuracy of performance improvement suggestions in practice.

References

[1] 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 *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 1001–1012.

[2] Yiqun Chen, Stefan Winter, and Neeraj Suri. 2019. Inferring Performance Bug Patterns from Developer Commits. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 70–81.

[3] Emilio Coppa, Camil Demetrescu, and Irene Finocchi. 2012. Input-Sensitive Profiling. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. Association for Computing Machinery, New York, NY, USA, 89–98. https://doi.org/10.1145/2254064.2254076

[4] Bruno Dufour, Barbara G. Ryder, and Gary Sevitsky. 2008. A Scalable Technique for Characterizing the Usage of Temporaries in Framework-Intensive Java Applications. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Atlanta, Georgia) *(SIGSOFT '08/FSE-16)*. Association for Computing Machinery, New York, NY, USA, 59–70. https://doi.org/10.1145/1453101.1453111

[5] Shi Han, Yingnong Dang, Song Ge, Dongmei Zhang, and Tao Xie. 2012. Performance Debugging in the Large via Mining Millions of Stack Traces. In *Proceedings of the 34th International Conference on Software Engineering* (Zurich, Switzerland) *(ICSE '12)*. IEEE Press, 145–155.

[6] Paul Kallender. 2005. Trend Micro will pay for PC repair costs.

[7] Sifei Luan, Di Yang, Celeste Barnaby, Koushik Sen, and Satish Chandra. 2019. Aroma: Code recommendation via structural code search. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–28.

[8] Christopher Manning and Hinrich Schutze. 1999. *Foundations of statistical natural language processing*. MIT press.

[9] Domas Mituzas. 2009. Embarrassment. http://dom.as/2009/06/26/embarrassment. Accessed: 2021-03-10.

[10] Glen Emerson Morris. 2004. Lessons from the Colorado benefits management system disaster. *Advertising and Marketing Review* (2004).

[11] Adrian Nistor, Tian Jiang, and Lin Tan. 2013. Discovering, Reporting, and Fixing Performance Bugs. In *Proceedings of the 10th Working Conference on Mining Software Repositories* (San Francisco, CA, USA) *(MSR '13)*. IEEE Press, 237–246.

[12] Paolo Pialorsi and Marco Russo. 2007. *Introducing microsoft® linq*. Microsoft Press.

[13] Serena Elisa Ponta, Henrik Plate, Antonino Sabetta, Michele Bezzi, and Cédric Dangremont. 2019. A manually-curated dataset of fixes to vulnerabilities of open-source software. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 383–387.

[14] T. Richardson. 2002. 1901 census site still down after six months. http://www.theregister.co.uk/2002/07/03/1901censussitestilldown/

[15] Linhai Song and Shan Lu. 2014. Statistical debugging for real-world performance problems. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*. 561–578.

[16] Amitabh Srivastava and Alan Eustace. 1994. ATOM: A system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*. 196–205.

[17] Xusheng Xiao, Shi Han, Dongmei Zhang, and Tao Xie. 2013. Context-Sensitive Delta Inference for Identifying Workload-Dependent Performance Bottlenecks. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis* (Lugano, Switzerland) *(ISSTA 2013)*. Association for Computing Machinery, New York, NY, USA, 90–100. https://doi.org/10.1145/2483760.2483784

[18] Guoqing Xu, Matthew Arnold, Nick Mitchell, Atanas Rountev, and Gary Sevitsky. 2009. Go with the Flow: Profiling Copies to Find Runtime Bloat. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland) *(PLDI '09)*. Association for Computing Machinery, New York, NY, USA, 419–430. https://doi.org/10.1145/1542476.1542523

[19] Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, Edith Schonberg, and Gary Sevitsky. 2010. Finding Low-Utility Data Structures. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation* (Toronto, Ontario, Canada) *(PLDI '10)*. Association for Computing Machinery, New York, NY, USA, 174–186. https://doi.org/10.1145/1806596.1806617

[20] Guoqing Xu and Atanas Rountev. 2010. Detecting Inefficiently-Used Containers to Avoid Bloat. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation* (Toronto, Ontario, Canada) *(PLDI '10)*. Association for Computing Machinery, New York, NY, USA, 160–173. https://doi.org/10.1145/1806596.1806616

[21] D. Yan, G. Xu, and A. Rountev. 2012. Uncovering performance problems in Java applications with reference propagation profiling. In *2012 34th International Conference on Software Engineering (ICSE)*. 134–144. https://doi.org/10.1109/ICSE.2012.6227199