Performance Test Selection Using Machine Learning and A Study of Binning Effect in Memory Allocators

by

Anderson Oliveira Sousa

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2019

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

This study contains reviews, suggestions and ideas from Professor Sebastian Fischmeister in Electrical and Computer Engineering department at University of Waterloo.

Chapter 1 was extended from [63], and had contributions from Petkovich, J. who aided in the machine learning approach, and Born, A. who assisted in feature engineering and the problem definition.

Chapter 2 was extended from [28], and includes contributions from Reza, A., who aided in detecting binning in Kernel Slab Allocator, and Petkovich, J. who assisted with suggestions.

Abstract

Performance testing is an essential part of the development life cycle that must be done in a timely fashion. However, checking for performance regressions in software can be time-consuming, especially for complex systems containing multiple lengthy tests cases. The first part of this thesis presents a technique to performance test selection using machine learning. In our approach, we build features using information extracted from the previous software versions to train classifiers that assist developers in deciding whether or not to execute a performance test on a new version. Our results show that the classifiers can be used as a mechanism that aids test selection and consequently avoids unnecessary testing.

The second part of this work investigates the binning effect on user-space memory allocators. First, we examine how binning events can be a source of performance outliers in Redis and CPython object allocators. Second, we implement a *Pintool* to detect the occurrence of binning on Python programs. The tool performs dynamic binary instrumentation on the interpreter and outputs information that helps developers in performing code optimizations. Finally, we use our tool to investigate the presence of binning in various widely used Python libraries.

Acknowledgements

I would like to express my deepest gratitude to my supervisor, Prof. Sebastian Fischmeister, for providing all the guidance, support and opportunities.

Thanks to Prof. Hiren Patel and Prof. Paul Ward for their time and effort to review my thesis.

Thanks to my family and my friends for all the love and kindness.

A special thanks to Angela for all the support during my Master's program.

Thanks to my friends in the Real-Time Embedded Software group, professors, staff members for all the assistance throughout these years. I would also like to thank Jean-Christophe for his suggestions and reviews.

Dedication

To my beloved parents, Virginia Almeida and Dionato Gomes.

To my precious sister, Sara Oliveira.

Table of Contents

List of Tables

List of Figures

Abbreviations

AUROC Area Under Receiving Operating Characteristic Curve 3, 7, 23, 26, 30, 32, 33, 35

DBI Dynamic Binary Instrumentation 59

FPR False Positive Rate 7, 25–27, 31

ICFG Interprocedural Control Flow Graph 4

IPC Inter-Process Communication 59

PRE Performance Risk Analysis 4

RFC Random Forest Classifier xi, 6–8, 25, 30, 36

ROC Receiving Operating Characteristic Curve xi, 3, 7, 20, 25, 26

RSD Relative Standard Deviation 43

RTP Regression Test Prioritization 4

SVM Support Vector Machine 6, 7, 25–27, 30, 31, 36

TNR True Negative Rate 32, 33

TPR True Positive Rate 7, 25–27, 31–33

VCS Version Control System 1, 3–5, 7, 9, 36

Chapter 1

Performance Test Selection Using Machine Learning

1.1 Introduction

Today's development methodologies aim to optimize the software release time and deliver high quality applications at lower costs. Agile [27] practices, for instance, try to tighten build test release cycle [75] and adopt an incremental and iterative development process to diminish project risks. DevOps practices seek to integrate development and operations [32], which reduces the gap between teams in a company and improve development efficiency.

Proper testing is fundamental for the adoption of these modern methodologies. However, there is a conflict between rapid development cycles and quality assurance processes since performance testing can be inherently a time-consuming phase [4, 70]. For instance, the non-functional tests of a complex system include load, stress, endurance and spike tests, which makes the software benchmark suite extensive. Bezemer et al. [4], showed that there is a problem in integrating performance testing and DevOps since most of the study participants did not spend significant time on performance engineering.

Furthermore, development teams typically make many source code changes daily. Each of these changes are recorded in a Version Control System (VCS) in the form of *commits*. It is challenging for developers to ensure whether their code modifications affect performance and, in many cases, running every single test in an associated benchmark suite is not a viable option. For this reason, some companies opt to skip test execution [44] or execute them only at the end of a project milestone [4].

D. Feitelson et. al [24] shows that release engineers at Facebook need to execute a battery of tens of thousands of regression tests for each new commit pushed to the repository. This lengthy phase means companies need an efficient methodology to select the exact set of performance tests to execute so practitioners would save several hours in the application deployment process. Therefore, in this study, we propose an approach to predict potential performance changes on new software versions and assist performance engineers in deciding which tests execute first [1].

Performance test selection is addressed in this thesis with an extended version of Perphecy [63]: a test selection tool that uses a greedy heuristic to train predictors of performance changes. The main threat to the validity of Perphecy is: how the predictors generalize to other software projects? Also, the author suggests that using better performance change indicators and machine learning algorithms could improve the tool effectiveness. Therefore, in this study we adopt different projects to verify the generality of Perphecy and machine learning to train classifiers and predict possible performance changes on commits recently recorded in the repository [2].

1.1.1 Problem Statement

We investigate a history-based performance test selection technique, which leverages information from previous versions of the program to solve the following problem: Given a new code commit and the software's benchmark suite, select all tests in the suite that are likely to be affected by the changes introduced in the commit.

1.1.2 Background

We extend the work of A. Oliveira et al. [63], who created the Perphecy tool and introduced the concept of performance change indicators. In his work, each performance indicator is a Boolean that is set to *true* whenever the difference between a new and an old commit is greater or equal to a threshold value, and *false* otherwise. The threshold value for each indicator is learned using a greedy heuristic that takes into account information from previous versions of the software. Then, the disjunction of multiple indicators is used to build a performance predictor.

[1] Benchmark and performance test are interchangeable throughout this thesis.

[2] A single commit is also considered a software version, even though it is not a tagged release version in the repository.

Besides adopting different case studies and machine learning algorithms, this study extends Perphecy in four ways. First, it divides the development timeline into several epochs such that a classifier is trained and tested in each epoch. Second, we designed additional performance features (indicators) to find possible performance changes on software. Third, a web framework was implemented to control experiment submission and speed up the data collection phase. Fourth, we evaluate the classifier's performance using well-known metrics in machine learning.

1.1.3 Related Work

Previous studies have also proposed approaches to regression test selection. However, most of them use static information only. Given the variety of techniques, Rothermel [71] defined a framework to analyze each approach. This framework is based on four main categories: inclusiveness, precision, efficiency and generality. We recognize the importance of these categories for the test selection problem; throughout this work we address all of them. For example, to evaluate the precision and efficiency of the predictors, we use the Area Under Receiving Operating Characteristic Curve (AUROC) metric and the ROC curve. Further, generality and inclusiveness are achieved by using multiple software projects and generic performance factors.

J. Alcocer *et al.* [1] proposed *Rizel*. It is a code execution profiler that runs benchmarks over software versions automatically and uses visualization tools to track performance, compare versions and point out the differences. In our work, we also developed a framework to run benchmarks on software automatically. However, unlike *Rizel*, running the profiler on the new version is not necessary since only static information from the binary is needed to predict the probable performance change. Also, we use information gathered from the executed tests to predict performance failures.

Luo [48] proposed *PerfImpact*, a tool that uses a combination of search-based input profiling and change-impact analysis methods. This tool finds input sets that cause regressions and recommends code changes that affect performance. Previous research works such as [62, 61] also focus on identifying performance regression root causes induced by code changes. In contrast, our work does not explore the program's input space to identify code changes that affect performance. Since this study works on a commit level, that is, the predictor analyses changes introduced by a single commit, determining the code change which caused the regression should be straight forward as long as developers use VCS correctly. The best practices for VCS suggests that code changes should be small and commits should occur early and frequently. Additionally, this work does not assume prior implementation of unit tests for the new version; the only requirement is performance tests that were already implemented for the software.

G. Rothermel *et al.* [72] targets this problem using a static analysis approach for object-oriented programs in C++. Rothermel constructs the Interprocedural Control Flow Graph (ICFG) representations for the software, in combination with branch trace information recorded from the performance test, to find the tests to execute. In opposition to this approach, we do not use the program's execution path to generate predictors but a combination of static and dynamic analysis (e.g. profile information) to infer about the relevant functions for a performance test.

P. Huang *et. al.* [34] and R. Saha *et. al.* [73] addresses the problem of Regression Test Prioritization (RTP) with *PerfScope* and *REPiR*, respectively. *PerfScope* tool conducts a Performance Risk Analysis (PRE) on the source code change to estimate the level at which it will affect performance. In combination with a risk matrix, the risk level obtained from the tool helps test engineers prioritize execution of performance tests on certain code patches. *REPiR* tries to reduce RTP to a standard Information Retrieval problem such that the program modifications form the query and the performance tests constitute the document collection. Our work contrasts with these approaches in two major aspects: we detect performance change via predictive modelling using dynamic and static information from past software versions, also we do not perform test prioritization. Therefore, no further analysis is needed once the prediction is made, however the features designed in this study could be used as a form of risk analysis. In Section 1.5 we describe the features designed for our data samples and their values could also be used as an indication of whether the child commit imposes performance risks with respect to the previous commit. Finally, R. Mukherjee*et al.* [58] discuss and catalogues of many approaches for RTP.

M. Gligoric *et al.* [29] also proposes a technique that accounts for the history of a project on a distributed VCS. His approach takes into consideration the complex history graphs from the repository, and the VCS commands performed to create the new software version, to generate the test selection technique. In contrast to Gligoric, a non-linear commit history is not necessary for training our predictors. Instead, parent and child commits can be selected from different branches, and only the difference between them is used as input.

1.1.4 Definitions

The next sections use the following definitions: B is the set of all performance tests for a software, and b_j is the j-th test in the suite. C is the set of commits in the software's

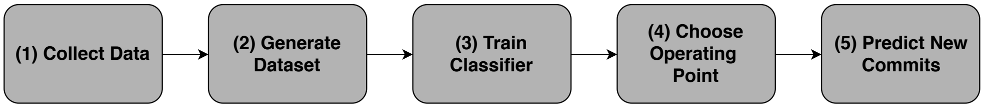


Figure 1.1: Performance test selection workflow.

VCS. Commit c_n represents a new software version, c_o is the closest parent of c_n and both $c_o, c_n \in C$. In our context, any commit in the repository is considered a new version. Additionally, c_o is the first ancestor of c_n whose profiling information is available. In other words, the parent is the latest commit which developers executed the performance tests [3]. Software profiling is referred to as *dynamic analysis*, and *static analysis* is the process of retrieving static information from c_n's binary.

1.2 Our Approach to Performance Test Selection

We investigate the performance test selection problem using a binary classifier. The classifier predict commit c_n as *positive* for performance test b_j, if the changes in c_n will likely affect the test's result, or *negative* otherwise. In a real-world scenario, classifying the new commit as positive means that c_n should be tested against benchmark b_j. A negative sample means a dismiss; that is, there is no need to execute b_j on c_n.

Figure 1.1 shows five steps that comprise our approach to performance test selection. First, we run all performance tests in B on all software versions in C to perform dynamic and static analysis and collect data. Details of this step are presented in Section 1.4. Second, we use this data to generate the features and class labels for the samples in the training and testing sets. A data sample in these sets represents the features computed from a tuple $t \in T, T = \{\langle c_n, c_o, b_j \rangle | b_j \in B, c_n, c_o \in C\}$. In other words, a data sample is a numeric vector of the features listed in Table 1.3 representing the comparison of c_o and c_n relative to test b_j. Also, in this step we label the data sample in the training and testing sets as *positive* if the performance change between c_n and c_o for test b_j is statistically significant, or *negative* otherwise. Third, we train a classifier through a machine learning algorithm for each $b \in B$. Fourth, to use the classifier the developer choose a probability threshold (operating point) which defines the performance of the classifier in terms of the number of true and false positives. Section 1.8 shows examples of how one can choose the operating point and Figure 1.1 illustrates every step mentioned above.

[3] In Git terminology, the parent commit is the predecessor(s) of a commit object, however, in this study the parent is an ancestor commit whose performance metrics was collected.

Finally, the performance test selection on new commits (fifth step) works as follows: when the developer records the recent code changes c_n in the repository, we perform a static analysis on c_n's binary and compare with profile information from its parent c_o. After, the classifier predict the data sample $\langle c_n, c_o, b_j \rangle$ as Positive or Negative for test b_j. c_n is positive to the performance test b_j if the probability outputted by the classifier is higher than the chosen probability threshold, or Negative otherwise. The computationally inexpensive static analysis on c_n makes this approach a fast technique to performance test selection.

1.2.1 The Binary Classifier

We evaluate two machine learning algorithms in our study: RFC [8] and Support Vector Machine (SVM) [26]. We recognize that there are multiple supervised classification methods such as Naive Bayes, K-means, Nearest Neighbours, Logistic Regression, Artificial Neural Networks, among others. However, we restricted the number of classifiers because the main focus of this study is to show whether performance test selection works with machine learning and if it generalizes for multiple software projects. Besides, training time is another reason why we narrow the number of machine learning algorithms evaluated. In total, we investigate twelve performance tests, and we train one classifier multiple times for every test in the benchmark suite. Therefore, finding the best model and model parameters would require longer training times.

RFC is an ensemble supervised machine learning algorithm which uses multiple weak learners to produce a better learner with a higher classification accuracy. The weak learners are decision trees, as shown in Figure 1.2. Each tree learns from a subset of the sample set and features, and the combination of their classification result generates a better overall classifier. Besides, RFC is robust to outliers in the dataset and does not require extensive hyper-parameter tuning.

We also adopted SVM classifier to compare with the classification results obtained with RFC and provide more confidence about our results. Support Vector Machine is a supervised learning technique that finds the hyperplanes that separate each class in the training dataset. Once the possible hyperplanes (also called boundary) are found, the algorithm runs an optimization routine to find the one with the maximum margin between classes.

1.2.2 Evalutation of the Classifier Performance

The prediction results are evaluated via the Receiver Operating Characteristic curve and its area [7]. The ROC is a probabilistic curve plotted of the True Positive Rate (TPR) versus the False Positive Rate (FPR) at different classification probability thresholds. The True Positive Rate is defined as the fraction of the commits that were correctly predicted as *positive* for a test b. False Positive Rate is the fraction of commits that were incorrectly classified as *positive*. Additionally, They are defined as follows:

$$\text{TPR} = T_p/(T_p + F_n) \tag{1.1}$$

$$\text{FPR} = F_p/(F_p + T_n) \tag{1.2}$$

$$\text{TNR} = T_n/(T_n + F_p) \tag{1.3}$$

T_p, T_n, F_n and F_p are the number of true positives, true negatives, false negatives and false positives, respectively.

Practitioners adopt the ROC curve to visualize the classifier's performance and select the best operating point (decision threshold). For instance, some practitioners might accept a high FPR as long as the classifier can predict the majority of the positive samples. Also, the ROC facilitates the comparison of RFC and SVM with a single figure.

Additionally, we provide the AUROC to evaluate the classifiers. We aim to maximize the area under the curve, which represent a ROC curve that is distant from the black diagonal line shown in Figure 1.5. AUROC gives the probability that a classifier will rank a randomly chosen positive observation higher than a randomly selected negative observation. Higher probabilities indicate that the classifier performs well at separating the two classes. Further, AUROC is one of the most used metrics when evaluating machine learning algorithms. Previous studies [33, 7] have shown that reporting the area is better than just the overall classification accuracy, especially when there is a significant class imbalance in the dataset.

1.3 Case Studies

The selection of the case study has a few challenging requirements. First, we target projects applied across a broad set of applications and whose routines exercise CPU, memory or file I/O subsystems. Second, we select open-source projects that use a VCS and has several commits recorded in the repository. We aim projects with one thousand commits

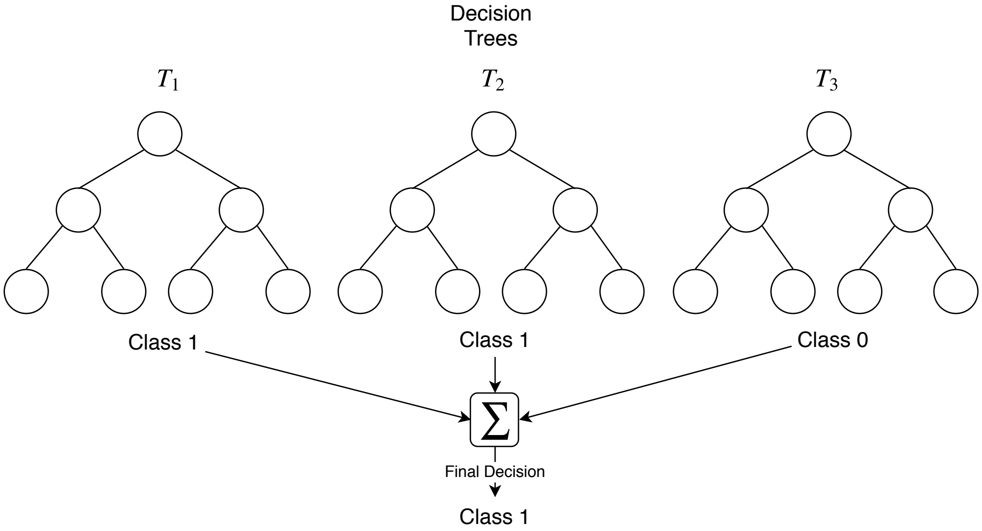


Figure 1.2: Decision trees in a RFC. This example uses simple majority voting to define the final decision.

or more because, among all versions, multiple builds are expected to fail. Third, we target applications developed in C or C++. Finally, the fourth requirement is to select projects that contain multiple statistically significant performance changes between parent and child commits throughout its history. For instance, we attempted studying FFmpeg which is a software that converts audio and video files among several formats. Unfortunately, this application is unpractical for this study because there were no relevant variations in performance in any of the commits we tested. The case studies adopted in this work are described below. Also, for the projects which we collect execution time, we measure the wall-clock time starting from the moment the process is spawned until its completion.

Redis

Redis [42] is a well-known in-memory data structure store applied in cache systems and, among others, message brokers. The project includes a benchmark tool called *redis-benchmark* which is used to generate real-world workloads and measure the server throughput. In total, there are approximately 225 commands available for the Redis client to manage the data structures and we choose four commands that are commonly adopted by any in-memory database. Also, the chosen commands are compatible with most of the previous software versions and are less likely to cause failures during the data collection phase. The chosen commands are the following:

Table 1.1: Number of commits available for each performance test.

Study Case	**Performance Test**	**Number of Commits**
Brotli	Compress	234
	Decompress	359
Git	Add	288
	Init	114
	Diff	278
	Clone	125
Jq	Contains	766
	Compare	759
Redis	Set, Get, Lpop, Lpush	154

- SET: set a key in the database.
- GET: get the value of a key.
- LPOP: Pop the first item from a list data structure.
- LPUSH: Push an item to a list data structure.

For each command, we designed a performance test that submits 500 thousand pipelined requests of 256 bytes to the server and measures the number of processed requests per second. This workload follows the guidelines shown in [20] and allows us exercise a realistic server behavior. For instance, for the SET test, each request sets a random key to a string value of 256 bytes in size.

Git

Git [76] is a popular distributed VCS used in most software projects. Among the several Git commands available, we choose the most popular ones for our performance tests and, for each command, we measure its execution time. The tests and workloads are the following:

- Add: Add ten files of 10 MB to the staging area.

- Init: Initialize a new repository.
- Diff: Compute the difference between two 100 MB files.
- Clone: Clone a repository.

The sizes chosen for this case study are large enough to exercise the core functions of Git.

Brotli

Brotli [16] is a data compression software developed by Google which uses a variant of the LZ77 algorithm. For this case study, we designed the following tests:

- Compress: Compress a large text file.
- Decompress: Decompress a file compressed with Brotli.

The file used in the Compression test is the enwik8, the first 10^8 bytes of the English Wikipedia dump. The Decompression test uses the compressed enwik9 file, which is the first 10^8 bytes of the English Wikipedia dump. These are the same files used in the Large Text Compression Benchmark [49]. For the Compress benchmark we measure the execution time and compression ratio, and for the decompression test we measure the execution time.

Jq

Jq [51] is a command-line JSON processor, capable of filtering, extracting and converting JSON formatted text. The two tests designed for Jq collects the execution time of a query with the following command-line arguments:

- Contains: Query entries whose field contains a string.
- Compare: Query entries whose field value is greater than an integer value.

We measure the execution time of both tests when processing a 54 MB JSON. The workload for this case study is large contains multiple fields which allows Jq exercise its querying routines which simulates a real-world workload. The *Contains* test first exercises querying, string parsing and sub-string comparison routines. The *Compare* test exercises querying, integer parsing and integer comparison routines.

1.4 Data Collection Phase

We collect data from each project listed in Section 1.3 and extract information that will constitute the sample features explained in Section 1.5. In this phase, we run every performance test on each software version to collect profiling information (dynamic analysis). Also, we perform static analysis in the compiled binary of each software version. Section 1.5 presents how we design the sample features with the data collected. Static and dynamic analysis are described below.

Dynamic analysis

The dynamic analysis aims to gather profiling data from the software. Profiling occurs at run time, while the program exercises the performance test. We perform dynamic analysis using *Perf Linux Profiler* [54] and *Pin* [47] and the information collected is shown in Table 1.2.

Perf is a low-overhead profiler that allows users instrument programs, count kernel events, trace memory usage, measure scheduler latencies, among others. Nowadays, it is an essential tool for engineers since it offers several features that facilitate performance evaluation. We collect two events using Perf: CPU cycles and instructions count (hardware events).

Pin is a dynamic binary instrumentation tool that provides a useful API which makes possible users create their instrumentation tools called Pintools. We use an adapted version of *proccount* pintool to count the number of times a function is called while the program is under test (more details about how *Pin* works is presented in Chapter 2 Section 2.6). To speedup profiling with Pin, the *proccount* pintool was modified such that only the main image is instrumented. Also, we measure the number of instructions each function executed while running the performance test.

The information in Table 1.2 shows how relevant a function is for a performance test. The profile information allows pinpointing the procedures which have the highest contribution to the overall performance. Therefore, the data gives practitioners hints of a possible significant variations in performance in case the function is modified in the next program version.

Table 1.2: Data obtained for dynamic analysis.

Information Collected	Description	Tool
Dynamic instruction count	Count the number of dynamic instructions executed by each function reached by the test.	Pin
CPU cycles count	Count the number of cycles spent on each function reached by the test.	Perf
Number of function calls	Count the number of times each function is called.	Pin

Static Analysis

We perform static analysis in the program's binary to count the number of assembly instructions in each function [4]. The analysis is done three steps: (i) build the program's executable, (ii) parse the contents of the disassembled executable and (iii) count the number of instructions in each function. The static instruction count is relevant because it allows a quick comparison of function changes in different software versions. Furthermore, combined with dynamic analysis, it provides more hints of how likely the performance of the new software version will be affected.

1.4.1 Infrastructure Used for Data Collection

The software profiling phase described above involves three steps: compilation, performance evaluation and profiling. In this study, we prioritize the best practices of performance evaluation, which states that test runs must be stable. However, this imposes another challenge for us: *complete data collection of all commits in a reasonable time given that the program needs to be tested multiple times.* Profiling one thousand commits for each case study could take several months to complete. For instance, considering that each commit takes at least 10 minutes to compile and test, it would take us approximately 138 days to finish data collection of all case studies including repetitions.

To overcome the challenge above, we developed a client-server infrastructure to perform a controlled and rigorous experimentation [64]. The infrastructure has of one master and

[4] Binaries where statically compiled and linked.

two worker nodes, the former orchestrate the distribution of tests and the latter execute the tasks previously assigned to them. The workers pull their pending tasks via a web-framework available in the master and execute them. In our context, a task represents a compilation, profiling or benchmark routine.

Master Node Setup

The master node uses Django [50] and the PostgreSQL database. Django is a fast, secure and scalable Python Web Framework that facilitates the development of web applications. The main goal for this application is to provide a simple interface for users to create experiments, schedule tasks and expose a web API to the workers. Once the experiment is defined in the master, workers fetch their tasks via *get_task* API endpoint on the server. When the execution finishes, they respond the server via *submit_result* endpoint. The PostgreSQL database [5] was the central storage for experiments metadata. Since the volume of data generated from experiments is enormous, reaching hundreds of gigabytes, we did not use the database to store the compilation and profiling results.

In the context of our study, an experiment represents a project which we want to collect data. Using the web interface, users define the experiment setup, such as its tasks (compiling and profiling), and the worker nodes that will execute them. In the Task creation page, users set the script which the worker node will run, the task type and any dependencies they may have. For instance, the dependency of a benchmark task is the successful completion of a compilation task. Further, the application engine contains a scheduler to facilitate task assignment. Since we can have multiple prioritized experiments, workers and tasks (with dependencies), a scheduler application was essential for overall coordination of the experiments. Additionally, the scheduler randomizes the task assignment to enhance experimentation robustness.

Worker Node Setup

Workers were designed to be near plug-and-play and lightweight, requiring minimal setup configurations to execute tasks. The nodes are an Intel(R) Xeon(R) E5-2620 with 26GB RAM and 24 Cores (hyperthreading enabled). The system is comprised of a minimal Debian distribution version 9.3 and Linux kernel version 4.9.

Rigorous software experimentation is not an easy task [59, 36]. The experimenter should be aware of most controlled, uncontrolled and random variables to reduce the variance of

[5] PostgreSQL version 9.6.

performance measurements. T. Kalibera [38, 39] shows that benchmarks are inherently and unavoidably a non-deterministic process because of the effects of random initial states and mutating system states. Also, it is impossible to eliminate all sources of non-determinism and the experimenter has the responsibility to modify the experimental procedure to mitigate it. For this reason, the worker's system needs to configured correctly to obtain high quality performance measurements [77, 60].

One of the advantages of benchmarking software in computers with multiple CPUs is that the machine will rarely be overloaded by the system's background processes and threads. Also, in our context, it allows the rapid completion of a compilation task since it can be done in parallel. However, even though 24 cores are available, only a subset was used for profiling software. Profiling tasks were isolated on a different set of cores to decrease the interference of system processes, reduce the number of CPU migrations and diminish context switch overhead [6].

Furthermore, we disable the processor frequency scaling and Intel Turbo boost capabilities. Additionally, to avoid thermal throttling [55], we fixed the CPU frequency to its minimum value of 1.2 GHz because the frequency driver cannot decrease the clock speed below this value when the temperature of the cores increase. Further, the tool *vmtouch* [7] was used for some tests that depend on data stored on disk such as Brotli. This tool reduces the overhead of *read* syscall and decrease disk I/O by leveraging file system caching [30]. Therefore, we eliminate a critical source of variances which is unrelated to the application under test [8]. Finally, we clean the system caches and perform a test warm-up before profiling begins. Each test is repeated at least five times to collect performance metrics, and a performance change between two versions c_a and c_b is found when the Mann-Whitney U test ($\alpha = 0.05$) shows a statistically significant difference in the benchmark result (b_j) of each version. This is a non-parametric test that is as efficient as the t-test for normal distributions.

1.5 Feature Engineering

We use the data obtained in the collection phase to design the performance features shown in Table 1.3. The features form a vector of values that indicate how likely there will be a difference in performance between the parent, c_n, and the child, c_o, commits relative to a

[6] This is known as CPU pinning and shielding. NUMA awareness was preserved when deciding which cores to isolate the profiling processes.

[7] *vmtouch* locks files stored on disk in the file system cache.

[8] During data collection, the operating system executes only the essential processes.

benchmark b_j. For instance, Chen, J. et al. [11] have shown that the simple addition of new functionalities to a function and changes in the algorithm is one of the main causes of performance regressions. Therefore, we added features *Changed functions* and *Instruction difference* to cover this root-cause of regression. The description of every feature is shown in Table 1.3.

1.6 Dataset Preprocessing

The training and testing datasets must be transformed before fitting the classifiers to avoid bias in the model towards features with bigger values. Figure 1.3a shows a histogram of the feature *New Functions* gathered from all Jq data samples. In the figure, no functions were added for the majority of the samples in the dataset. The sparse distribution skewed to the right is explained by the fact that most child commits do not differ too much between their parent in terms of the number of new functions and it is likely that only distant commits would have more than ten new functions. The skew is unavoidable because of software development nature, where developers tend to commit small changes to the source code, especially when the project is mature.

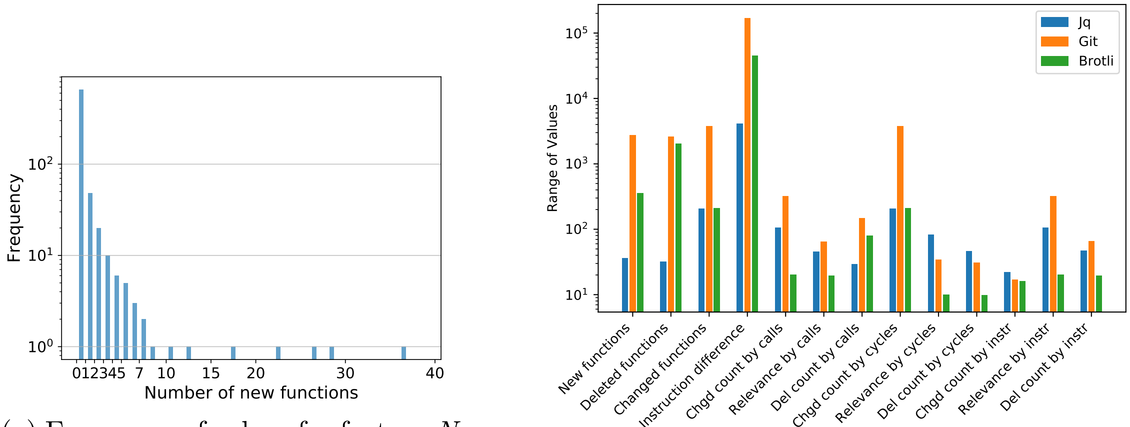
Furthermore, the range of features values is very distinct. The bar plot in Figure 1.3b shows the difference between the maximum and minimum values for each feature in Table 1.3. For instance, the ranges for features *Chgd count by instr* and *Instruction difference* contrast significantly. Therefore, the features are linearly scaled by its maximum absolute value to overcome these issues in the sample dataset. As a result, the values lie in between the $(0, 1)$ interval and preserves the sparsity of the data.

We expect that the execution of tasks fail due to various reasons [9] and not all commits selected for data collection will be compiled or profiled successfully. One of the consequences of this problem is not only a reduced number of data samples to train our models, but also the presence of data samples with longer commit distances. Also, the performance test selection is preferable when predicting close commits relative to the time since distant commits are more likely to have differences in performance. Therefore, in addition to scaling features to a specific range, we weight for each sample such that the penalty of mispredicting commits chronologically close to each other is higher than mispredicting distant ones.

[9] Automating compilation and profile of software projects with a significant amount of commits is challenging since we must customize scripts for specific software versions.

Table 1.3: List of Features for a tuple $\langle c_n, c_o, b_j \rangle$ extended from [63].

Feature	**Description and Rationale**
New functions Deleted functions	Overall number of new and deleted functions. It indicates new functionality or refactoring which can lead to overall performance variation.
Instruction difference	Difference in number of static instructions of functions. The execution of more instructions can affect performance.
Changed functions	Overall number of changed functions. Function change indicates bug fixes or code improvements.
Relevance by cycles Relevance by call Relevance by instr	Profile the program and find the hottest functions (highest overhead) relative to three categories: number of CPU cycles (cycles), number function calls (call) and number of dynamic instructions executed (instr). The rank of the function in the profile tells its contribution to the overall performance (its relevance). We summarize the relevance for all changed functions for each category. Changing a function with high impact on performance, e.g. the one which had the highest count for dynamic instructions, is likely to cause performance changes.
Chgd count by CPU cycles Del count by CPU cycles Chgd count by call Del count by call Chgd count by instr Del count by instr	For the hottest functions, count how many were changed and deleted. Changing hot functions, e.g. five most called, is likely to affect performance.



(a) Frequency of values for feature *New functions* for Jq samples.

(b) Range of values for each feature computed for Jq, Git and Brotli samples.

Figure 1.3: Statistics for features listed in Table 1.3.

1.7 Commit Selection for Training and Testing Sets

In Section 1.2 we describe that a sample in both the training and testing sets represents a tuple $\langle c_n, c_o, b_j \rangle$, where c_o is a parent commit of c_n whose profile information was collected for performance test b_j. The simplified algorithm to train and test a classifier is shown in Algorithm 1 [10]. In the algorithm, functions *get_train_tuples* and *get_test_tuples* are the routines that generates the data samples, and T_{tr} and T_{te} are the set of tuples in training and testing sets, respectively. S_{tr} and S_{te} are the set of data samples in training and testing sets, respectively, after computing the features. Procedure $compute\_features(t, b)$ generates the features for a tuple t according to Section 1.5. Finally, we train a classifier m through the machine learning algorithm in procedure *train*, and output the predicted class probabilities P after testing the samples in function *test*.

The method to select commits from Set C in the Algorithm 1 (lines 2 and 3) to form the data samples in the training and testing sets deserves attention since it can influence the classifier's performance. Before presenting our approach to generate the data samples, first, we evaluated three intuitive methods that practitioners might use to implement functions *get_train_tuples* and *get_test_tuples*. Also, one could implement the first three methods in case their dataset size is very small. However, in Section 1.7.1 we discuss that although

[10] We added the simplified version for clarity. The optimized algorithm of Algorithm 1 does not compute features at run-time.

intuitive, the Methods 1, 2 and 3 below can generate negative results. Thus our evaluation saves the petitioner's time when choosing alternatives to generate samples in the training and testing sets.

For Methods 1 to 3, *we remove the condition that c_o is the closest ancestor of c_n*, therefore, c_o can be any parent of c_n. Also, we evaluate each method using Jq *Compare* test and the results are shown in Figure 1.4 Table 1.4. Also, we use an example of development history whose commit graph that does not contain branches, however, the methods below are also extended to graphs which contain several branches and merge commit.

Algorithm 1 Simplified routine to train a classifier for benchmark b.

```
Input: $C$, the ordered set of commits; $b$, the target performance test.
Output: Classifier $m$; The predicted class probabilities $P$.
$S_{tr} \leftarrow \emptyset$, $S_{te} \leftarrow \emptyset$
$T_{tr} \leftarrow get\_train\_tuples(C)$
$T_{te} \leftarrow get\_test\_tuples(C)$
for all $t$ in $T_{tr}$ do
    $S_{tr} \leftarrow S_{tr} \cup compute\_features(t, b)$
end for
for all $t$ in $T_{te}$ do
    $S_{te} \leftarrow S_{te} \cup compute\_features(t, b)$
end for
$m \leftarrow train(S_{tr})$
$P \leftarrow test(m, S_{te})$
return $m, P$
```

Method 1

A set T_{tr} is formed from all combinations of two commits in C, which results in $\binom{n}{2}$ tuples. For the methods that adopt this kind of combination, c_o can be *any* ancestor of c_n. After, tuples in this set are randomly selected to form the training and testing set (e.g. 70% selected for training and 30% for testing). For instance, suppose a development history with the commits in shown Figure 1.4. Three possible pairs of commits among all are $t_1 = \langle c_2, c_1 \rangle$, $t_2 = \langle c_3, c_2 \rangle$ and $t_3 = \langle c_3, c_1 \rangle$. Then, the practitioner could choose t_1 and t_2 for training and t_3 for testing.

We trained a classifier for Jq Compare test using this method. Table 1.4 shows that it has the best score among the first three methods. The smooth curve in Figure 1.5 is

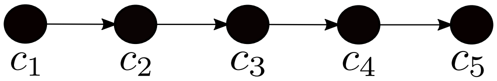


Figure 1.4: Example of commit graph with five commits.

a consequence of the increased amount of testing samples because of the combination of commits (86,299 in total).

Method 2

We select the training and testing commits from C at random, without replacement. Then, the commits in training set are combined, $\binom{n}{2}$, to form tuples in T_{tr}. The same approach applies to generate T_{te}. This strategy is similar to Method 1, except that commits in each set are selected from C before they are combined. In this method, *no commit will ever be present* in both testing and training sets.

Using the commit graph shown in Figure 1.4 as example, one could select commits c_1 and c_5 to training, and c_3 and c_4 to testing. Then, the tuple in the training set is $\langle c_5, c_1 \rangle$ and the tuple in the testing set is $\langle c_4, c_3 \rangle$. As seen in Table 1.4, the training set for this method has fewer testing samples compared to Method 1 and lower score.

Method 3

The first n earliest commits in C (e.g. 70% of total commits) are selected for training, and the remaining are chosen for testing. Commits selected for training are combined similar to Method 1, $\binom{n}{2}$, to generate tuples in the training set. For the testing tuples, the parent c_o is the latest commit selected for training (newest commit in the training set). After, the test commits are paired with their parent. Therefore, all tuples in this set will have the same parent c_o.

Using the commit graph shown in Figure 1.4 as example, one could pick the earliest commits c_1 c_2 and c_3 for training, and c_4, c_5 for testing. Then, The tuple in the training set are $\langle c_2, c_1 \rangle$, $\langle c_3, c_2 \rangle$, $\langle c_3, c_1 \rangle$. Since c_3 is the latest commit in the training set, it will be the parent for both c_4 and c_5. Therefore the testing set has the tuples $\langle c_4, c_3 \rangle$ and $\langle c_5, c_3 \rangle$.

In Table 1.4 we notice that Method 3 has less commits and lower score than the previous two methods.

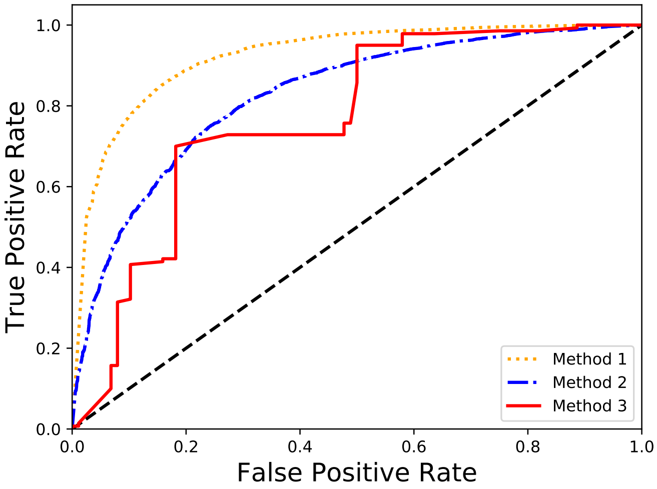


Figure 1.5: ROC curves for Methods 1, 2 and 3 using Jq study case.

Table 1.4: Results for Methods 1, 2 and 3 using Jq study case.

	Dataset Size		
Method	**Training**	**Testing**	**AUROC**
1	201,362	86,299	0.90
2	140,715	25,878	0.83
3	140,715	228	0.77

1.7.1 Applicability of Methods 1 to 3

Although methods 1 to 2 demonstrated to have better scores, they are not suitable for this study. Their most significant advantage is that by generating synthetic commits they have an increased amount of data samples which might also overcome the class imbalance in the datasets. However, these methods are not applicable for us because of the following reasons: first, pairing very distant commits relative to time is not meaningful for practitioners because the bigger distance between commits more likely the execution of benchmark is necessary. Second, augmenting the train set proved to be unsuitable because the software performance along its history is not always monotonic (increasing or decreasing). For example, the parent commit c_1, representing an older version of the software, could have the same performance of a child c_{100} that is several commits ahead of it even if their source code differences are enormous. Training a model with this pair could generate a predictor with low sensitivity to small code changes.

One of the consequences of selecting commits randomly (Methods 1 and 2) is information being leaked from the training set to the test set. Especially for Method 2, it could cause a misconception that estimators are tested with completely unseen data or no information in the training set is passed to the testing set. For example, this method allows the occurrence of the following scenario: tuple $\langle c_3, c_1, b_j \rangle$ in train set and tuple $\langle c_4, c_2, b_j \rangle$ in the test set. If commits c_1 to c_4 did not have significant changes in their source code, they would represent almost the same sample. As a consequence, predictors tested with them would have higher scores but fail to perform well on unseen data. Finally, Method 3 does not evaluate the classifier completely because only the latest commits of the development history are tested.

These observations guided us to develop a method which enforces the non-existence of data leaks from the training set to the testing set and allows practitioners to evaluate the predictor's performance thoughtfully. The approach we propose in this study divides the development history into segments, called *epochs*, such that an predictor is trained and tested for each epoch.

1.7.2 Method 4: Backtesting Cross Validation

Given the disadvantages presented above, now we evaluate the classifier for the performance test using backtesting cross validation [13] and and propose an intuitive approach to select the training and testing commits. In our approach, we segment the development history in *epochs* such that each epoch contains all commits *starting from the first commit until a*

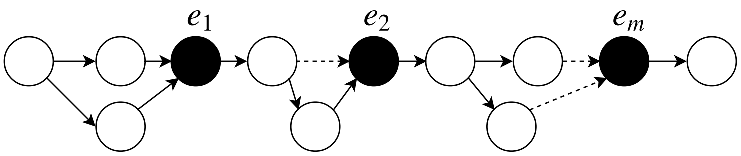


Figure 1.6: Example of commit graph that was segmented in m epochs.

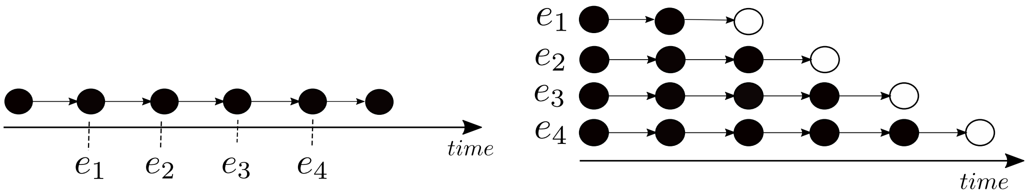
chosen commit date. The chosen date defines the epoch, and it could represent, for instance, a *milestone* in the project development where developers released a new major version and ran all performance tests on the commit. Figure 1.6 illustrate a development graph that is segmented in m epochs (black commit nodes). These nodes represent a commit which dynamic analysis was made and profiling information is available. Also, the commit graph could be segmented in different ways to train the classifier of a distinct performance test, because the number of data samples available for a performance test might diverge. This divergence is a consequence of scripts failing during the data collection phase.

Finally, we train and test a classifier for the performance test for each epoch. Then, we use all the predicted class probabilities to obtain the overall classifier performance. The approach to define the samples in training and testing sets of each epoch is explained below, and Algorithm 2 has the pseudo-code for evaluating a classifier using this Method 4 where every loop iteration in line 2 represents an epoch.

Defining the Training and Testing Samples Per Classifier and Epoch

We select all the commits available for the epoch to train a classifier for performance test b_j. Then, we combine the training commits to form the tuples $\langle c_n, c_o, b_j \rangle$ such that c_o is the closest ancestor of c_n whose profiling information is available. The developer chooses the commit which represents the epoch according to the availability of profiling information.

Further, we select k child commits ahead of the epoch to form the tuples in the testing set. The test commits represent, for instance, new code changes that developers added after the latest version was released and dynamic analysis was done. Then, the tuples $\langle c_n, c_o, b_j \rangle$ in the testing set are formed by pairing the testing commits with their parent c_o. Commit c_o represents the epoch and whose profiling information is available. The number of testing commits, k, ahead of the epoch is a design decision which the practitioner evaluates the impact on the classifier's performance. Large values for k is not ideal because, as code changes are added to the software, more likely that there will be a performance variation between parent and child commits.



(a) Example of commit graph with six commits, divided in four epochs.

(b) Example of four epochs which $k = 1$. Black nodes are commit belonging to the training set, and empty nodes belong to the testing set.

Figure 1.7: Example of a commit graph. In real-real world scenarios, the repository graph would have multiple branches and merges.

Example

We now provide an example of how a developer can segment the commit graph in multiple epochs to train a classifier for one of its performance tests. Figure 1.7a illustrates a development history with six commits. We segment the graph in four epochs and select one commit ahead of the epoch ($k = 1$) to test the classifier. Figure 1.7b shows how the commits for each epoch would look like, including the commit selected for testing (white node). For instance, the training commits of e_4 are all commits from the beginning of the development until the date the epoch. Then, we train and test the classifier for each epoch and evaluate the combination of all class prediction probabilities.

1.8 Evaluation of the Performance Test Classifiers

In this section, we present the scores of the classifiers trained for each performance test using Method 4 and five testing commits for each epoch ($k = 5$). In our evaluation, we choose five test commits because it is not too low so the training time would be considerable high, and not too large so the commits predicted ahead of the parent would likely contain a statistically significant performance change. In Section 1.9.2 we also evaluate the scores for different k values.

The results are summarized in Table 1.5, and they show the classifier performance of each study case and benchmark for all epochs combined. One way to assess the overall score from Table 1.5 is averaging the sum of the best AUROC scores for each performance test and study case. As a result, on average, the likelihood the classifiers will be able to

Algorithm 2 Training and testing a classifier on a segmented development history.

```
Input: Set $C$, the ordered set of commits; $k$ the number of test commits per epoch; $b_j$, the target performance test; $l$, number of training commits for the first epoch.
Output: The predicted class probabilities $P$.
$S_{tr} \leftarrow \emptyset$, $S_{te} \leftarrow \emptyset$
while $n < |C|$ do
    $T_{tr} \leftarrow \{\langle c_n, c_o, b\rangle | c_n, c_o \in C\}$
    $T_{te} \leftarrow C[l] \times C[l : l + k]$
    for all $t$ in $T_{tr}$ do
        $S_{tr} \leftarrow S_{tr} \cup compute\_features(t, b)$
    end for
    for all $t$ in $T_{tr}$ do
        $S_{te} \leftarrow S_{te} \cup compute\_features(t, b)$
    end for
    $m \leftarrow train(S_{tr})$
    $P \leftarrow P \cup test(m, S_{te})$
    $m \leftarrow update(m)$
end while
return $m, P$
```

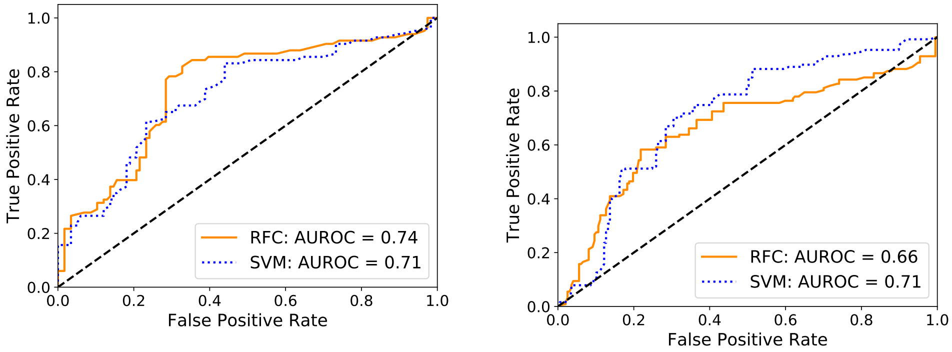
Table 1.5: Results for each case study and performance test.

Study Case	**Performance Test**	**RFC**			**SVM**		
		TPR	**TNR**	**AUROC**	**TPR**	**TNR**	**AUROC**
Brotli	Compress	0.39	0.79	0.74	0.32	0.82	0.71
	Decompress	0.58	0.76	0.66	0.38	0.85	0.71
Jq	Contains	0.31	0.81	0.66	0.07	0.96	0.72
	Compare	0.07	0.96	0.67	0.04	0.99	0.66
Git	Add	0.38	0.91	0.71	0.44	0.55	0.59
	Init	0.87	0	0.35	0.81	0.2	0.50
	Diff	0.27	0.94	0.62	0.86	0.35	0.66
	Clone	0.95	0.52	0.89	0.65	1	0.93
Redis	Set	0.58	0.75	0.75	0.41	0.95	0.83
	Get	0.82	0.32	0.62	0.36	0.96	0.79
	Lpop	0.67	0.43	0.63	0.66	0.58	0.67
	Lpush	0.65	0.64	0.71	0.31	1	0.72

predict whether or not there will be a relevant performance variation between commits is approximately 72%. Redis is the study case with best likelihood of predicting a significant change in performance, with approximately 75% on average, and Brotli the lowest, approximately 69%.

1.8.1 Brotli

Models trained for Brotli have the area under the curve above 0.7 for both benchmarks. Also, RFC is more efficient on Compress test while SVM performs better on Decompress test. Figure 1.8 shows the ROC curves for this case study. Using the RFC for Compress benchmark as example (Figure 1.8a), the developer could choose the probability threshold that gives 0.8 and 0.4 for TPR and FPR, respectivelly. This operating point indicates that the test is dismissed on 60% of the true negative commits (66 commits in total) while detecting 80% of the performance affecting changes.



(a) Compress test.

(b) Decompress test.

Figure 1.8: ROC curves for Brotli.

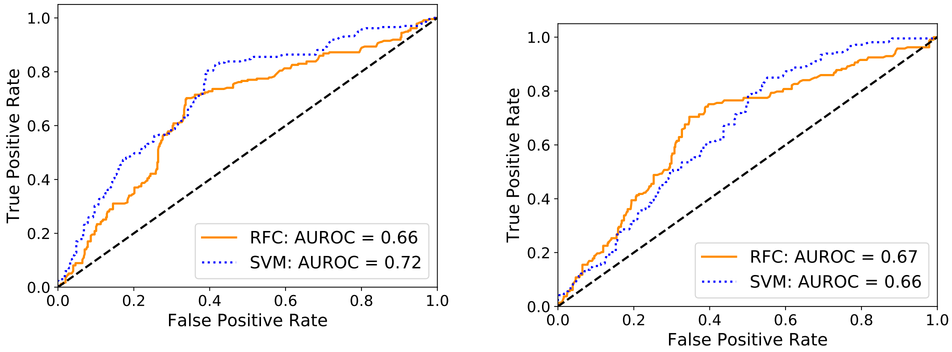
1.8.2 Jq

The AUROC for all classifiers trained for Jq is 0.67 on average. Figure 1.9 shows the ROC curves for each performance test. If we use as example the SVM classifier trained for the Contains test, the developer could choose from Figure 1.9a the probability threshold which results in 0.8 TPR and 0.4 FPR. With this operating point, the classifier correctly dismisses 60% of the true negative samples, which represents approximately 74 of all commits, while detecting 80% of the performance affecting commits.

1.8.3 Git

Figure 1.10 shows the ROC for Git study case. The classifier for Clone test has the best AUROC overall, reaching up to 0.93 for SVM classifier. From Figure 1.10d, one could choose, for example, the probability threshold which results in 0.4 and 0.9 FPR and TPR, respectively. Then, the classifier would correctly dismiss tests on 60% of the commits that do not impose risk to performance (11 commits) and detect 90% of the commits that affect performance.

The predictor performance for Init test, in Figure 1.10b, was no better than random guessing since its ROC curve follows the black diagonal. Investigating the results for this test in Table 1.5, we observe that the classifier was very conservative, detecting 87% of the true positives but none of the true negative samples. The bias towards one class is an indication that the features generated from the data collected for this performance test did



(a) Contains test.

(b) Compare test.

Figure 1.9: ROC curves for Jq.

not capture the changes that does not impose risks to performance; therefore the algorithm was incapable of distinguishing the two classes effectively.

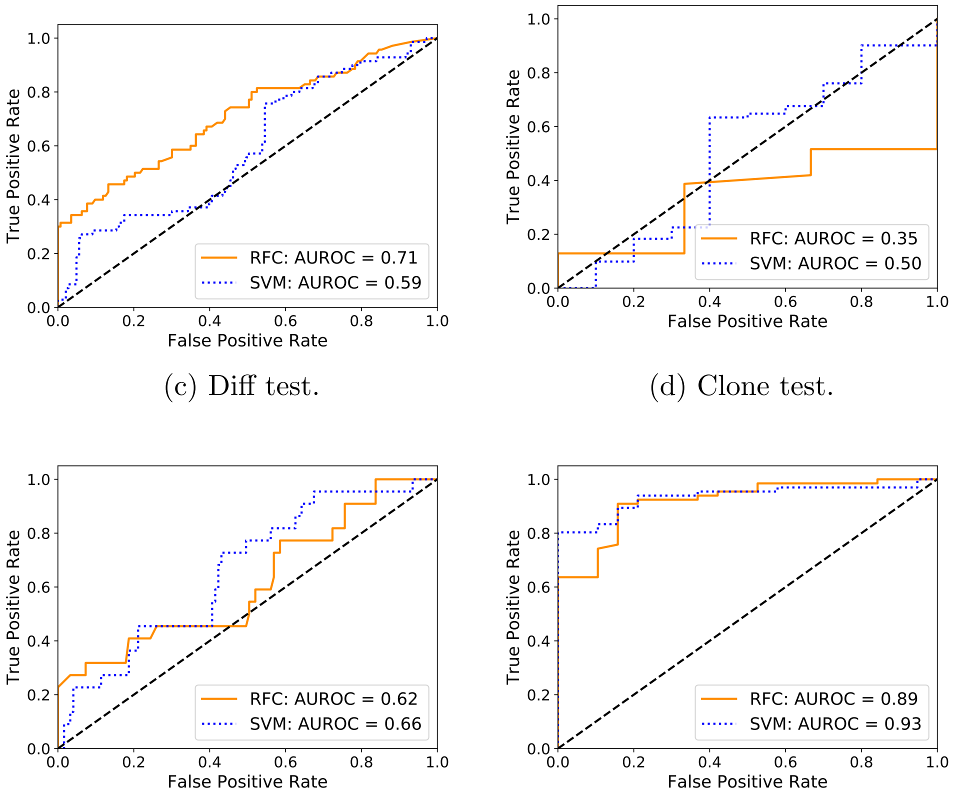
1.8.4 Redis

From Table 1.5, Redis is the only case study which the Support Vector Machine was better than Random Forest for all test cases. Also, the discrepancy between AUROC scores for each classifier type is significant for both Get and Set tests. This difference is also shown in Figures 1.11a and 1.11b.

To assess the performance of the SVM classifier for Set test, for instance, the developer could choose the probability threshold (operating point) from Figure 1.11a which gives 0.4 and 0.9 for FPR and TPR, respectively. For this threshold, the classifier would correctly dismiss test execution on 60% of commits that do not affect the performance of Set benchmark (40 commits) and detect approximately 90% of the commits that induce a change in performance.

(a) Add test.

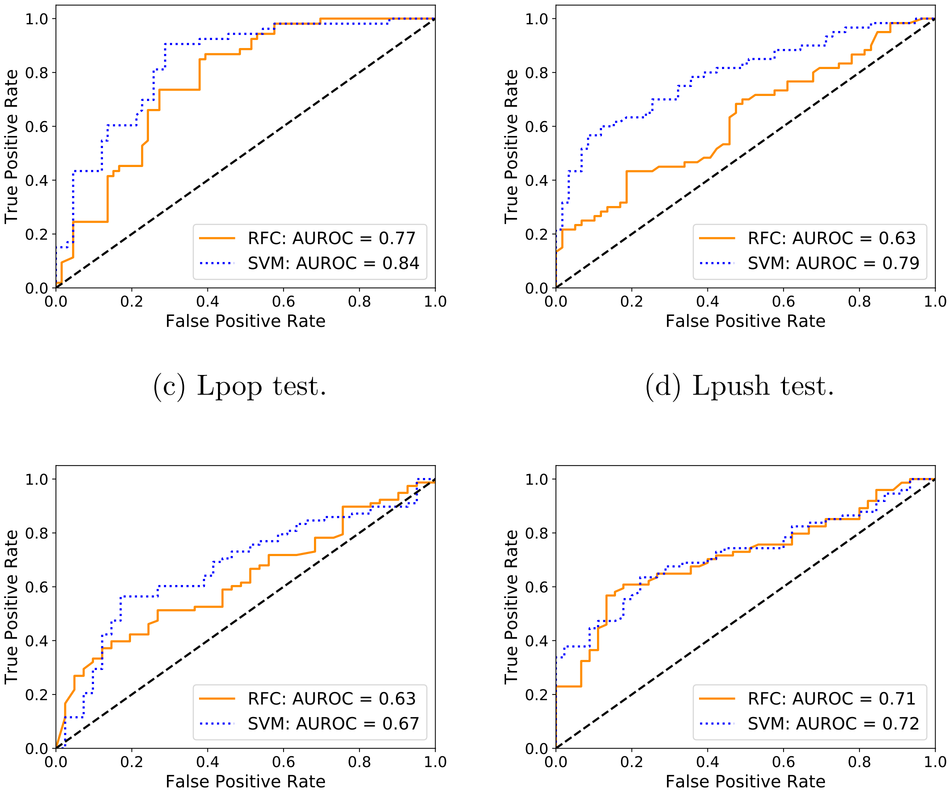
(b) Init test.



(c) Diff test.

(d) Clone test.

Figure 1.10: ROC curves for Git tests.



(a) Set test.

(b) Get test.

(c) Lpop test.

(d) Lpush test.

Figure 1.11: ROC curves for Redis tests.

1.9 Discussion

This section presents a further investigation on the performance of the classifiers.

1.9.1 Classification Scores Throughout the Development History

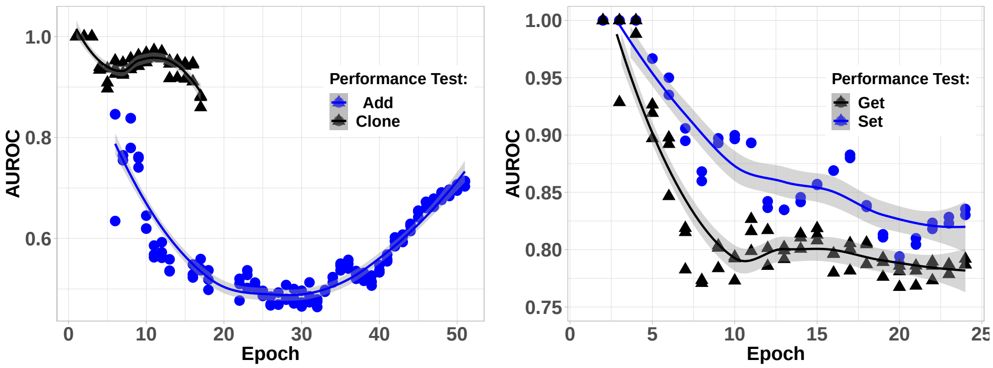
The AUROC scores provided in Section 1.8 refer to the class prediction probabilities over all epochs combined. Now we use Method 4 to visualize how classifiers evolve over the project development. We chose Git and Redis study cases and plotted the smoothed curves for the AUROC for each epoch, as illustrated in Figures 1.12.

Figure 1.12a shows the scores for Git Add and Clone tests, where the RFC was trained and tested repeated times using 100, 500 and 1000 decision trees. Add test have the most noticeable variability and the AUROC reaches 0.71 by the end of the development history. The plot also indicates that starting from epoch 30, the AUROC improves once more samples are included in the training set. In contrast, scores for the *Git Clone* benchmark are relatively more stable, reaching 0.91 over the 17 epochs. Clone test has fewer epochs because there are fewer data samples available for this case study compared to the Add benchmark; that is, fewer commits were successfully profiled when we collected data for this study case. As a consequence, there will be fewer epochs to train the performance classifier.

Figure 1.12b shows the scores for Redis Get and Set tests, where a SVM classifier was trained and tested repeatedly using three kernels types [26]: sigmoidal, polynomial and radial basis. We observe in the figure that the classifiers initially overfit when less data is available for training and then reach 0.8 for Get and 0.85 for Set test in the last epoch when the training set has almost all the samples available for the test.

1.9.2 Effect of the Testing Set Size

In Section 1.8, we test the classifier against the first five child commits ahead of the parent commit ($k = 5$). Now, we investigate the effect of the test set size on the overall classifier performance using Method 4. Table 1.6 presents the results for Jq and Brotli for a range of values for k in Algorithm 2. We observe that the AUROC tends to increase for bigger test sets. This increase is evidence that the larger the distance between parent and child commits higher is the likelihood of a variation in performance performance. Also, this is an indication that choosing large values for k is not ideal for evaluating a classifier



(a) Git.

(b) Redis.

Figure 1.12: Classifier score over multiple epochs.

because, intuitively, the larger the distance between the child and parent commits, the higher the likelihood of a performance change. Further, a One-way ANOVA statistical test was performed for Jq, under the null hypothesis that there is no difference in the AUROC score for different k values. The results show a F-statistic value of 3352 and p-value of 10^{-33}, and Tukey's HSD post hoc analysis indicates that the null hypothesis should be rejected for all values of k except 1 and 50.

1.9.3 Comparison to a Conservative Test Selection Algorithm

We compare the classifier generated for Redis Set test to a dummy algorithm that always indicates that new commits will affect performance. In other words, the conservative algorithm classifies all samples as positive, which means it never dismisses tests and TPR = 1. Using our approach, the developer could choose the class probability threshold (operating point) from Figure 1.11a which gives a TPR of 1 and a FPR of 0.72 for SVM. Therefore, using this threshold we would correctly dismiss 28% of the commits (skip tests on 17 of them) that did not affect the performance while also detecting all performance-affecting commits. In contrast, the dummy approach does not skip the tests on any of the commits that did not affect performance which means unnecessary testing.

Further, a naive algorithm that always indicates that new commits will not affect performance (TNR = 0 or FPR = 1) would not capture any of the performance changes. Although this dummy approach may save time by skipping performance tests, it is not a recommended software development practice, and it may incur many risks to the detection of root causes of performance regressions. Therefore, with the chosen threshold,

Table 1.6: Classification scores for different numbers of test commits per epoch (variable k in Algorithm 2).

Performance Test	**Test Set Size (k)**	**AUROC**	
		RFC	**SVM**
Jq Compare	1	0.63	0.67
	10	0.58	0.59
	20	0.50	0.6
	50	0.62	0.63
	100	**0.7**	**0.75**
Brotli Decompress	1	0.62	0.68
	10	0.6	**0.72**
	20	0.64	0.64
	50	**0.65**	0.52

our classifier would not only correctly dismiss tests but also perform early detection of all performance changes between child and parent commits.

1.9.4 Classifier Evaluation Using K-fold Cross Validation

We also assess the classifier performance for two Git tests via k-fold cross validation [56]. In this scenario, instead of dividing the development history into multiple epochs (Method 4), we use all samples, $\langle c_n, c_o, b_j \rangle$, available for the performance test. For every sample, c_o is the closest ancestor of c_n with profiling information available. Then, we measure the AUROC scores for multiple k values (2, 5, 10 and 20) with at least three repetitions.

Figure 1.13 illustrates the classifier performance for Clone and Add tests for each fold (the smooth curve represents the confidence interval). The scores for both tests level off at 0.85 and 0.725 for Clone and Add tests.

1.9.5 Comparison to Perphecy

This work differs from Perphecy [63] in many ways. The first important difference is the adoption of AUROC which is a more comprehensive scoring metric than TPR and True

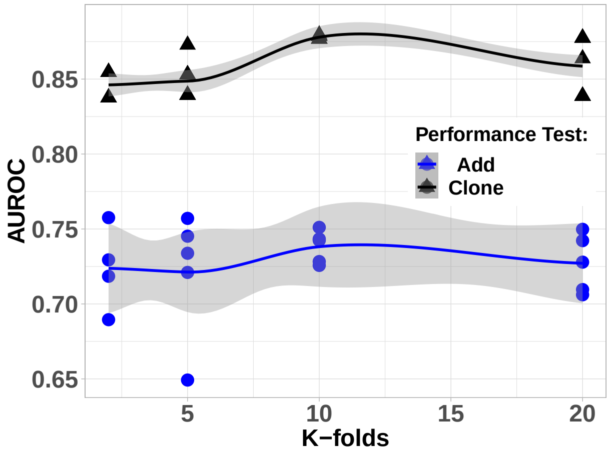


Figure 1.13: K-fold cross-validation using Git.

Negative Rate (TNR). Second, we evaluate the classifiers over the project's development lifetime. However, the contrast in approaches imposes challenges when comparing the two studies.

The overall score obtained for the Git project in the previous study was 0.77 for true positives and 0.84 for true negative rates. Roughly comparing with our study, and assuming that we use the same set of commits, we notice that the classifier for Clone test was the only one that performs as well as the previous study (in the previous work, the score for Git was 0.8 TPR and 0.84 TNR. However, this observation is not a decisive manner of comparing the two studies since, as we have discussed in Section 1.7, the approach for selecting commits can affect the classifier performance. Also, Table 1.6 shows an example in which choosing different test commits (k) changes the AUROC score by almost 10% for certain projects.

1.9.6 Feature Design

We designed the features described in Section 1.5 to capture most of the information that could indicate a change in performance. The features are meant to be generic and applicable to any software application that generates an ELF binary. Also, they could be customized for each project. Developers can use their experience with the source code to extend these features and capture better possible performance-affecting changes.

Jq has the most significant training size which is almost two times the amount of Redis and Brotli. However, the AUROC scores seen in Table 1.5 indicate the classifiers for this project did not perform exceptionally well, even though the dataset is relatively bigger. To further investigate this problem, we analyzed a false negative sample from the Jq study case. Listing 1.1 shows the code snippet with the difference between a parent commit and its direct child[11]. This snippet is from a procedure that reads the contents of a stream and outputs to a buffer, and it shows that line 8 is substituted by line 10 on the new version. The commit author pointed an I/O bug in the parent commit which caused the procedure to pull only 8 bytes of the stream at a time, instead of the amount available for a previously declared buffer of 4096 bytes in size. After fixing the bug, the code makes fewer application I/O requests, which reduced the total execution time by 34%.

The sample representing the bug fix was studied to verify whether the features extracted from it captured the performance-affecting change. For instance, the feature *Instruction difference*, which is the difference in the total number of static instructions between parent and child, has its value equal to 2 (90 in the child and 88 in the parent). Therefore, the addition of only two instructions caused a significant improvement in performance which shows that the set of features did not adequately capture this performance change.

A developer working on this project, who is aware of the risks that this function imposes, could profile past versions of Jq to gather, for example, the amount of cache-misses, system calls or disk I/O requests that each function makes. This information can then be used in combination with the difference of static instructions to form a new feature. For this reason, in the section below we study the addition of a memory-related feature, such as *cache-misses*, to the data collection phase and verify how it changes the classifier performance.

```
1    diff --git a/main.c b/main.c
2    static int read_more(char* buf, size_t size)
3    {
4      ...
5   -- if (!fgets(buf, sizeof(buf), current_input))
6        buf[0] = 0;
7   ++ if (!fgets(buf, size, current_input))
8        buf[0] = 0;
9      return 1;
10    }
```

Listing 1.1: Code difference between a parent commit and its direct child with respect to procedure *read_more*. The performance of the new version was significantly better because of the bug fix in line 7.

[11] Parent and child commit references are d32777 and 52db80, respectively.

Effect of Additional Features

Linux offers hundreds of performance counters available for the users (e.g. *Perf list* reports 1711 in total). In this section, we investigate whether adding a memory-related counter would improve the results seen in Section 1.8. We chose the two commits analyzed in Listing 1.1 and obtained the difference of *cache-misses* using *Perf record* to count the number of times a data was not present in the CPU cache. When a *read* system call is made to fetch data from the file system, it might trigger a cache-miss event in case the data is not present in the cache. Also, we compute the difference of misses between parent and child with *Perf diff* tool. Column Delta in Listing 1.2 shows the top five functions in the child commit that caused the most significant changes in the cache-misses.

We designed three more features to verify whether cache-misses affect our results: *Relevance by cache-misses*, *Changed count by cache-misses* and *Deleted by cache-misses*. Then, we collected profiling information for Jq and evaluated the classifiers with the new features added. The AUROC score, illustrated in Figure 1.14, shows no significant improvement on the classifier performance when the new features are added. This finding indicates that further investigation on Jq is needed to obtain features that capture its performance-affecting changes.

```
# Event  'cache−misses'
# Baseline      Delta         Symbol
# ........     .......      ...........................
  38.60%     −0.07%      [.]  jvp_array_write
  29.33%     +0.35%      [.]  jvp_array_free
   9.21%     −0.18%      [.]  jvp_refcnt_inc
   8.14%     −0.33%      [.]  jvp_refcnt_dec
   3.18%     +0.08%      [.]  jv_get_kind
```

Listing 1.2: Performance difference between Jq commits d32777 and 52db80 with respect to cache misses.

1.10 Lessons Learned and Future Work

We present the lessons learned throughout the implementation of this performance test selection and discuss how our approach could be improved in future works below.

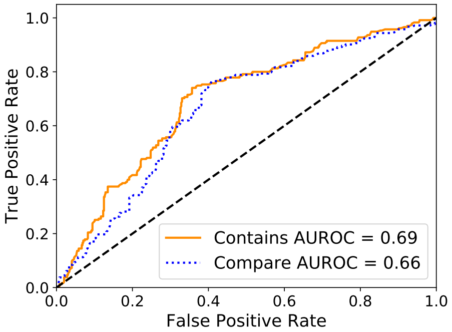


Figure 1.14: ROC for Jq including features for cache-misses.

Evaluation of Multiple Classifiers and Parameter Tuning

In total, we analyzed more than 3,000 commits and studied 24 performance classifiers for multiple epochs in a development history. The costs imposed by the data collection phase and the high number of classifiers we evaluate restricted how in depth we explored different machine learning techniques. From our results, we have evidence that RFC or SVM can be used to aid performance test selection. However, we believe that an in-depth evaluation of multiple models is needed to find the best machine learning algorithm. Additionally, one could use the newest techniques on deep learning techniques to improve our results. Finally, we believe that the results in Method 4 could be improved if we tune the model parameters.

Dataset Sizes

The first lesson learned from this study refers to the challenges of obtaining a very large amount of data samples to train and test classifiers. Automating compilation and execution of performance tests on all commits in the repository is not a simple task because compilation commands could change from time to time, causing scripts to break and experiments to fail. Therefore, this issue bounds the size of the data set and limits the number of samples available in the evaluation the classifier.

Software Development Practices

Correct usage of the VCS is essential history-based performance test selection techniques. The best practices for Git mention, for instance, are that code commits should be small and occur often. This practice facilitates finding the root cause of performance regressions. As well, it is beneficial for performance predictors since they become sensitive to functions that impose high risks to performance.

Further, Git offers the option to squash commits together via *git rebase* command, which imposes a threat to any history-based performance test selection approach. Sometimes commits are squashed to clean the repository or decrease its total size. However, combining commits could make the delta between parent and child even larger because the removal of intermediary code changes increases the number of differences between them.

Design of Features

In Section 1.9.6, we presented a data sample that was not predicted correctly by the classifier. Capturing performance-affecting changes can be challenging, and code changes as small as the addition of two instructions can have a significant effect on performance. The features designed in this study (in Table 1.3) are intended to be generic and applicable for most applications. However, it is possible that some projects have their own set of relevant performance counters which the developer is aware of. Therefore, we believe that better results would be achieved if developers include their own application-specific features when using our approach.

Thorough Evaluation of Classifiers

In Section 1.7, we explore different ways to generate data samples for the testing set, and we conclude that random selection of commits do not seem to be a realistic approach to form training sets. Although the classifiers have higher scores for Methods 1 to 3, most of them did not demonstrate how well they perform with true unseen data. This important threat to validity motivated us evaluate a method that tests the classifier simulating a real-world software development scenario. In future work, one could evaluate additional cross validation approaches, such as walk-forward backtest, to observe how changes in the dataset affect the predictions.

1.11 Conclusion

Performance testing can increase the costs of software development significantly since developers might spend hours executing benchmarks. To diminish the costs, developers need to make a wise decision on which tests execute for a given commit. Our approach to performance test selection uses the information from previous versions of the software to train a binary classifier that provides the likelihood of a performance change when developers record a new commit in the repository. Our approach does not substitute the performance test execution. However, it is a fast technique that one can use to assist the decision of whether or not to test the new software version after a commit push.

In Section 1.7, we studied multiple ways of using the Version Control System to generate data samples for machine learning algorithms. Since the majority of the methods presented are not realistically enough for solving the test selection problem, we use an approach that allows developers to evaluate the classifier effectiveness over the development history. From our results, we have evidence that that one can use machine learning to predict performance changes and assist developers in selecting test. We further evaluate our approach using Support Vector Machine and, for some case studies, it has superior scores than Random Forest Classifier. Finally, the classification scores were assessed with a variable number for the test set size and the results demonstrate that different sizes have a statistically significant effect on the prediction score.

Chapter 2

A Study of Binning Effect in Memory Allocators

2.1 Introduction

Performance measurements of computer systems do not always follow a symmetric normal distribution and it is common to observe log-normal distributions from measurement samples. Several factors inherent to the system and hardware may interfere with the execution of programs. For instance, network contention can be one of the various factors that cause high response times [78] when a server process the client's request. Therefore, the peaks in execution time and the lack of knowledge of the entire system turns performance evaluation a laborious [25, 74] task because the practitioner might not fully understand the hidden causes of such outliers.

In this chapter, we study the binning effect, which is a type of measurement variation that occurs in memory managers, and we investigate how it is present in two user-space programs. Nowadays, almost all memory allocators [23, 46] use an array of free lists [57] to manage memory regions. The arrays are used to segment memory areas into chunks and these chunks are further segmented into smaller areas to form pools of memory blocks. This approach improves cache locality and performance, and diminishes internal and external fragmentation. However, binning is a side effect of this method and causes longer response times in certain allocation requests. Each memory manager has its own peculiarities specific to the application that uses it, and we show that this effect occurs at different layers concerning the proximity to the user, for instance, we review its presence in the Linux Kernel Slab Allocator [53] and investigate it in CPython and Redis memory allocators.

2.1.1 Definition

A bucket or bin is a container which memory blocks are added or removed. At some moment in the execution of memory-intensive programs, the memory allocator will reach the limit (threshold) of free blocks in a bin or pool of memory regions. This event, referred in this work as *binning event*, causes the allocator control flow to take a slow path to serve the next memory request which results in a higher allocation response time. As example, the threshold for a bin could represent the size of the private heap available for a program and, once the threshold is reached, the memory manager executes a routine to request more memory from the operating system and initialize its internal structures to store the in new blocks. In the next sections we explain in detail how binning occurs on each case study.

2.1.2 Problem Statement

In this chapter we address the following problem: Given a particular program and memory allocator measurements, explain the performance outliers.

2.1.3 Related Work and Chapter Organization

Amir et al. [28] investigated binning events on Kernel Slab Allocator [53] and studied its effect on the performance of many system calls. The experiment for *mmap* system call was was reproduced in Section 2.3 of this work and we perform further analysis on our results. Further, in this study we develop a *Pintool* to detect binning events on Python Programs. Unlike Peiris et al. [67] who created *EMAD*, a dynamic instrumentation tool that identifies excessive memory allocations, we do not focus on studying the process memory usage or memory leaks. Also, the module *tracemalloc.py* available for Python can be used as a debug tool to understand the program's memory usage. This module works in combination with trace points in the interpreter to collect statistics and tracebacks on memory usage. Although the *Pintool* is not intended to track memory usage, it also suggests the developers the code locations where memory is being mostly used.

This chapter is organized as follows: Section 2.3 presents binning effect in Kernel Slab Allocator ant it serves as a basis for our additional case studies. Sections 2.4 and 2.5 investigates binning in Redis and CPython memory allocators. In Section 2.6, we introduce a tool to detect the presence of binning in Python programs and evaluate it on multiple performance benchmarks.

2.2 Methodology

Robust, rigorous and scientific experimentation in software engineering is not always a simple task to perform since the number of variables that interferes the experiment can be enormous. For this reason, this work tries to control the data collection phase to minimize the risks of taking the wrong conclusions about the performance measurements. The performance test[1] used to identify binning events are either a standard benchmark provided by the developers (for Redis) or synthetic workloads implemented for this study. The computer hardware and system setup utilized in our experiments is the following:

Hardware

The machine is an Intel^R Core^{TM} i5-4460S 64 bits 2.90 GHz, quad-core (hyperthreading enabled) and 16 GB RAM. The System was configured with a minimal Gentoo installation, Linux Kernel 4.14.65 and GCC 7.3.

System Configuration

We set up the system in which our experiments were conducted to diminish disturbances from external controlled variables [36]. First, the system load was kept low during the experimentation phases, and only the essential system processes were running. Also, we execute the performance tests on a single CPU that is isolated from the system processes. This action not only avoids external disturbances but also eliminates latencies due to CPU migrations and context switches. Second, the machine clock frequency is fixed to its minimum value possible, which removes variations in clock frequency due to thermal throttling. Third, the Intel Turbo Boost support was disabled to cease sudden processor acceleration during peak loads [3]. Fourth, we disable Address Space Layout Randomization (ASLR) due to its possible effect on performance [66]. According to O. Augusto et al. [15] the impact of ASLR is not insignificant, however, since our performance tests are heavily dependent on memory mappings, we opt to factor out this possible source of variability. Lastly, caches are dropped and slab objects are reclaimed before every experiment execution.

[1] Performance test is also referred as workload or benchmark.

2.2.1 Analysis of Binning Effect

Besides plotting the performance measurements of memory allocation requests, this study adopts the following approaches to visualize and quantify the binning effect on performance of Redis and CPython memory allocators:

Effect Magnitude

The study of the binning effect on performance is done via the Mann-Whitney U test and Games-Howell post hoc analysis. The U-test null hypothesis (H_0) is: the distributions of the execution time of memory allocations with and without binning are equal. The alternative hypothesis (H_a) is: the distributions of the execution times of the two groups are different (or the mean ranks of the two groups are different). Rejecting the null shows that binning has a statistically significant effect on the performance. Once a difference between the two groups is found, post hoc analysis quantifies this difference providing the effect magnitude.

Since we cannot assume a specific distribution of the dependent variable (execution time) and homoscedasticity, classic mean regression methods such as ordinary linear regression can lead us to wrong conclusions about the effects of binning. Long-tailed and log-normal distributions is commonly present in timing measurements of software [12, 78, 45](Figure 2.3b). Therefore, as an attempt to make the distribution close to normal, some practitioners apply transformations (such as Box-Cox or simply taking the square root of the response variable) to achieve normality or near normality. In this work, we avoid the adoption of these techniques since it would be difficult to generalize this study. For this reason, we choose the non-parametric statistical test instead.

Distribution Skewness

The relationship between the distribution skewness and binning events is also studied. The study is done via Fisher-Pearson coefficient of skewness [2], G_1, adjusted for the sample size which tells how the distribution asymmetry increases (and consequently its variability) in the presence of binning. The coefficient is defined by:

$$G_1 = g_1 \frac{\sqrt{n(n-1)}}{n-2} \quad \text{where} \quad g_1 = \frac{\frac{1}{n}\sum_{i=1}^{n}\left(\frac{x_i-\bar{x}}{s}\right)^3}{\left(\frac{1}{n}\sum_{i=1}^{n}(\frac{x_i-\bar{x}}{s})^2\right)^{\frac{3}{2}}} \tag{2.1}$$

[2] Available in R via package e1071.

n, $\bar{x}$ and s are the number of samples, mean and standard deviation, respectively. Joanes et. al. [37] has shown that Fisher-Pearson coefficient has less bias and small mean-squared error for an asymmetric distribution with small amount of samples, which is suitable for this study since we expect to obtain skewed distributions from our data. Our reference to compare is the normal distribution, whose G_1 is zero (symmetric). Positive and negative values of G_1 indicates right and left skew, respectively, in our data.

To observe the binning effect on the distribution we plot G_1 versus the number of binning events to visualize how it affects the shape of distribution under variable number events. The plot is constructed by taking n consecutive samples and computing G_1. Then, the coefficient is plotted against the number of events that occurred among those requests.

Variability Around the Mean

The execution time variability caused by binning events are shown via a plot of the Product Moment Coefficient of Variation ($CV = 100 * (s/\bar{x})$) (or Relative Standard Deviation (RSD)) versus the number of binning events. There are multiple statistical methods for computing variability [31], for instance, one can use statistics such as the *L-moment Coefficient of Variation*, estimated by the second L-moment divided by the first L-moment ($\frac{l_1}{l_2}$). However, in this is study we use CV since it is more intuitive for practitioners. The plot of the coefficient of variation versus the number of binning events allows us to visualize the performance dispersion when the memory allocation is under binning effect. Similar to the skewness plot mentioned above, the plot is generated by computing the CV for n consecutive data samples and counting the amount of binning events that occurred.

2.2.2 Study Cases

This study presents the binning effect in three essential applications: Linux Kernel, Python Core, and Redis. Amir et. al.[28] have studied the performance outliers in the Slab Allocator in the Kernel; therefore we will not explore in depth in this work. However, since the Slab Allocator is situated in Kernel space, we will introduce it to serve as a baseline for our the next case studies.

Similar memory allocation techniques are also widely used in user-level space libraries, and we show in Sections 2.4 and 2.5 that Redis and CPython memory allocators are also prone to binning.

2.3 Linux Kernel Slab Allocator

The SLAB Allocator is a memory manager that handles allocation of kernel objects of fixed size. These objects store metadata information for kernel structures such as memory mapped areas, network information, file inodes, etc. SLAB was first introduced to Solaris by Bownwick [5], later extended to SMP computers [6] and today is widely used in Linux Kernel. This allocator improves overall performance and reduces memory fragmentation.

The gain in performance occurs because of the caching of commonly used objects. The motivation behind this technique is the high overhead of object initialization, which sometimes can be higher than allocating memory for it. Objects of the slab cache are initialized only once, and its memory is not entirely freed after it is deallocated. The free object goes to an array of objects so it can later be re-utilized without the overhead of another initialization. Therefore, reallocating these objects can be very quick; for instance, our experiments have shown latencies of less than 5 μs.

In addition to caching, another motivation to slab allocation is the reduced memory fragmentation. A cache of a certain type (e.g., to store file descriptors) can have several slabs, each one with a typical size of one page (4KB). The division of a slab into small segments of equal size, one for each one kernel object, prevents memory fragmentation. For instance, object `vm_area_struct` of 208 bytes in size [3] stores information about the processes memory mapped areas. Since the page frame size is by default 4KB, 19 objects fit in each slab of this cache type [4].

Objects allocated and freed are placed in a per-CPU cache. In case the CPU does not have free objects in its cache to serve an allocation requested, the used objects are transferred to a pool in RAM, and the kernel routine `cache_alloc_refill` refills the CPU cache with free objects [53]. This routine to move objects to and from memory causes a relatively significant delay in execution time. Additionally, the slab allocator could potentially interact with the page allocator to get a new memory page and subdivide it into smaller segments to store more cache objects. Therefore, this extra delay represents a source of binning. More details regarding the binning effect on Kernel Slab Allocator is found in [28].

[3] Size is specific the kernel compiled for the machine described in Section 2.2. Information about kernel objects can be found inspecting /proc/slabinfo in case the kernel was compiled with SLAB as its default allocator.

[4] Coloring techniques are utilized to improve hardware cache utilization.

2.3.1 Experimental Setup

A different method from [28] was used to study binning on Kernel Slab Allocator. In our approach, we leverage dynamic instrumentation to measure the latency caused by binning. This method works on top of Perf, a profiling tool available for Linux, in conjunction with Kprobes. It also is possible to use Debugfs [5] directly to trace binning events, however, Perf [54] contains all features needed for this experiment. Perf Kprobes are inserted at `mmap` system call handler entry and exit addresses to measure the time taken to complete the request. Then, binning events are detected by tracing calls to `cache_alloc_refill` Kernel function.

Kprobes [40, 52] allows us dynamically insert traps in almost any kernel function (entry and exit) to collect tracing and debug information. Also, they have low overhead (0.07 to 0.1 microseconds to process [52]) and the advantage that there is no need to modify the process's source code. The only requirement is that the kernel must have debugging capabilities and symbols enabled during compilation.

The experiment workload, shown in Listing 2.1 [6], is a C program that performs 1000 consecutive memory map calls of 20 bytes in size. Every *mmap* call issues a system call that creates a private copy-on-write mapping and, besides obtaining a memory area, it internally allocates one *vm_area_struct* Kernel object from the CPU's free list. The sequential request of memory areas consumes several Kernel objects which induce the initialization of new slabs and multiple CPU-caches refills.

```c
int fd = open('/dev/zero', O_RDWR);
for(int i = 0; i < 1000; i++){
  ptr = mmap(0, 20, PROT_READ | PROT_WRITE, \
  MAP_PRIVATE, fd, 0);
}
```

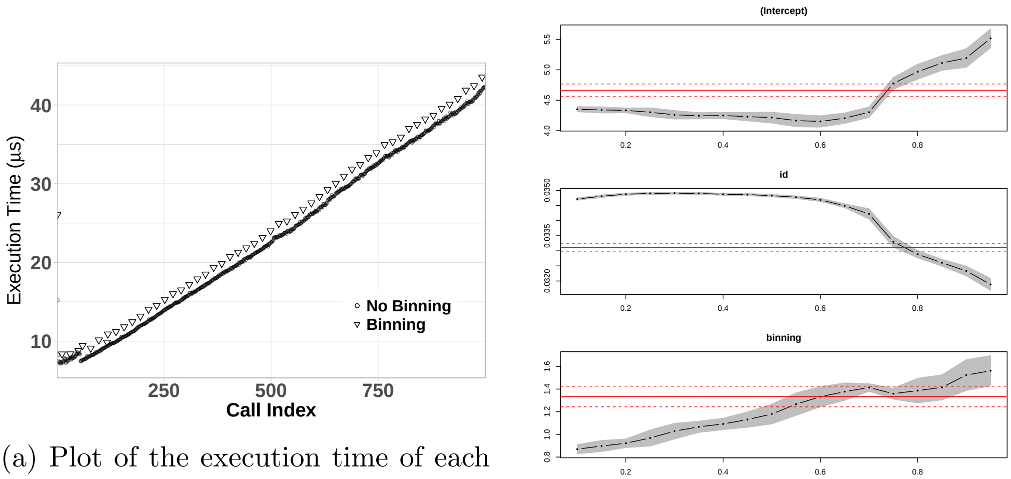
Listing 2.1: Workload for Kernel Slab Allocator study case.

2.3.2 Analysis

Figure 2.1a shows the execution time for all consecutive `mmap` calls versus the call index. The peaks in latency and the rate at which binning occurs is noticeable. As explained

[5] Debugfs filesystem exposes kernel information and it is typically mounted on /sys/kernel/debug.

[6] Irrelevant code snippets are not included.



(a) Plot of the execution time of each mmap request ordered by call index.

(b) Plot of the binning effect on multiple quantiles.

Figure 2.1: Results for kernel Slab Allocator workload.

above, the CPU holding the kernel objects has to refill its cache with free object regularly, which increases the time to process the call. The events occur periodically; however, differently from [28], they are triggered in multiples of 19 calls. This is equivalent to the tunable Kernel parameter called *objperslab* that defines the number of `vm_area_struct` objects a slab holds. Furthermore, this value is ideal because, including metadata overhead, 19 objects fit in a 4 KB slab.

Figure 2.1b shows the quantile regression from the 5^{th} to the 95^{th} quantile. This type of regression offers a robust method to verify how binning affects different quantiles of the distribution [41, 10, 43, 14]. The following conditional quantile model was estimated:

$$Q_Y(\tau|X) = \beta_0(\tau) + \beta_1(\tau)x_1 + \beta_2(\tau)x_2 \tag{2.2}$$

where X represents the binning and call index factors. Binning factor has two categorical levels: *no binning* (0) and *binning* (1). $Q_Y(\tau|X)$ is the model we estimate at the τ^{th} quantile, β_0 is the intercept parameter (not relevant for this study) and $\beta_1(\tau)$ is the estimated parameter effect for binning. $\beta_2(\tau)$ is the estimated parameter for the call index.

The effect of binning and call index at different quantiles in Figure 2.1b is shown via dashed black lines. Red lines indicate the effect estimated via linear regression. At the 95^{th} quantile, the contribution of binning is about 1.56 μs and the magnitude reduces with lower quantiles. The effect of the call id decreases for higher quantiles, which conforms with the curved scatter plot. Quantile regression does a better job in estimating the effect since it varies for different quantiles of the data.

2.4 Redis' Memory Allocator

Redis [42] is a data structure store that uses main memory to reach high performances on data manipulation. Redis uses Jemalloc [23] as underlying memory allocator, which is a fast general purpose implementation of the `glibc malloc`, known to reduce fragmentation and be highly scalable. Jemalloc has been used in Facebook, Mozilla Firefox, FreeBSD, among other relevant projects.

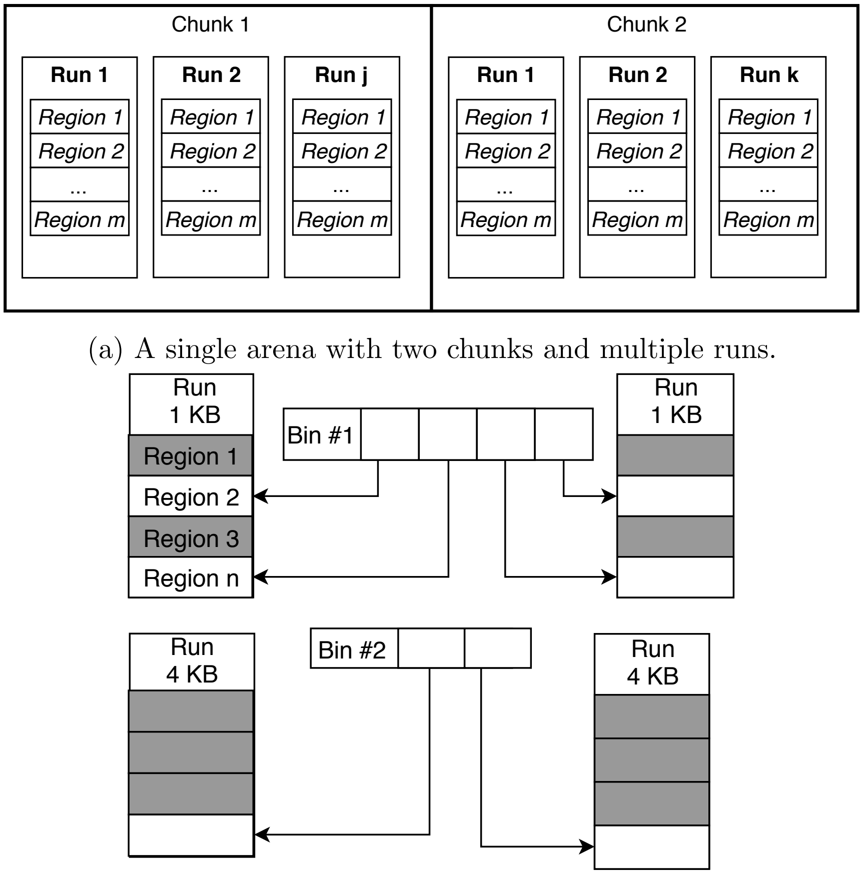
Redis works on top of a client-server model. Clients connected to the server submit Load (e.g. GET, LPOP) or Store (e.g. SET, LPUSH) requests to the server, and the server responds to the client accordingly. When a client submits a Store command such as LPUSH, Redis uses Jemalloc to obtain a free memory area of approximate or equal the request size to store the client's data.

The memory allocator architecture is shown in Figure 2.2a. In Jemalloc terminology, an *Arena* is an independent memory area managed by a thread. This is one of the most significant advantages of Jemalloc since associating memory areas to individual threads reduces synchronization of allocation requests via lock contention. An Arena is divided into multiple chunks (Figure 2.2a show only two chunks) and each one has the same size (1 to 4MB depending on the version). Chunks are divided into page *Runs* and a Run contains memory areas of same size class called *Regions*. Runs are responsible for storing all metadata necessary to manage their Regions, including which ones are allocated (in use) and free (or deallocated). When the user requests a memory area, Jemalloc returns the reference pointing to the Region according to the request size.

Memory allocations requests in Jemalloc are classified in two categories [7]: small and large. Small requests are less than four times a page size (less than 16 KB) and large requests are bigger than four times a page size and up to a maximum threshold. Also, small and large requests are further divided in size classes. To keep track of the free regions of a specific size class, the memory allocator uses a tree structure called *Bin*. Finally, each arena has its own set of bins, one for each size class. Figure 2.2b shows two bins storing references to free Regions.

When a user requests memory, the algorithm first looks into the request size and obtain a region from the specific Run class. Bins provide a quick way of finding free regions for

[7] Category types might change depending on the version.



(a) A single arena with two chunks and multiple runs.

(b) Bins containing array of free Regions. Bin 1 stores references to free Regions (unfilled) in Runs of class size 1 KB. Bin 2 points to regions of size 4 KB.

Figure 2.2: Jemalloc memory architecture.

the user. Binning effect is present on Redis when the user submits a command of type *Store* and there are no free Regions to serve the memory allocation request or no memory is available to initialize new chunks. In this case, the control flow takes a slow path to request memory from the OS, initialize a new *Chunk*, update Bins and initialize a Run. This additional overhead causes a peak in latency to process the client's memory allocation request.

2.4.1 Experimental Setup

The Binning effect in Redis was studied by measuring the time the server takes to process the client's command. The target procedure in the server's source code responsible for executing any received command is `call` [8]. If binning occurs, it increases the average time taken to process the command. Similar to Section 2.3.1, the measurement is done via *Perf* dynamic probes, which are inserted at `call` function entry and exit addresses and the difference of timestamps between entry and exit give us the execution time. Binning events are detected by capturing *mmap* system calls while the client's command is processed.

Redis 5.0.1 compiled with Jemalloc 5.1 was used for this study case. We use *redis-benchmark* tool to generate the workload which consists of 10 thousand consecutive and pipelined requests issued to the server from one client [9]. Every request is an LPUSH command to insert an object of 256 KB in size in a list structure. Eventually, the server's preallocated memory will run out of free regions of 256KB in size. Then, memory is obtained from the system and routines to initialize new Chunks and Runs are executed.

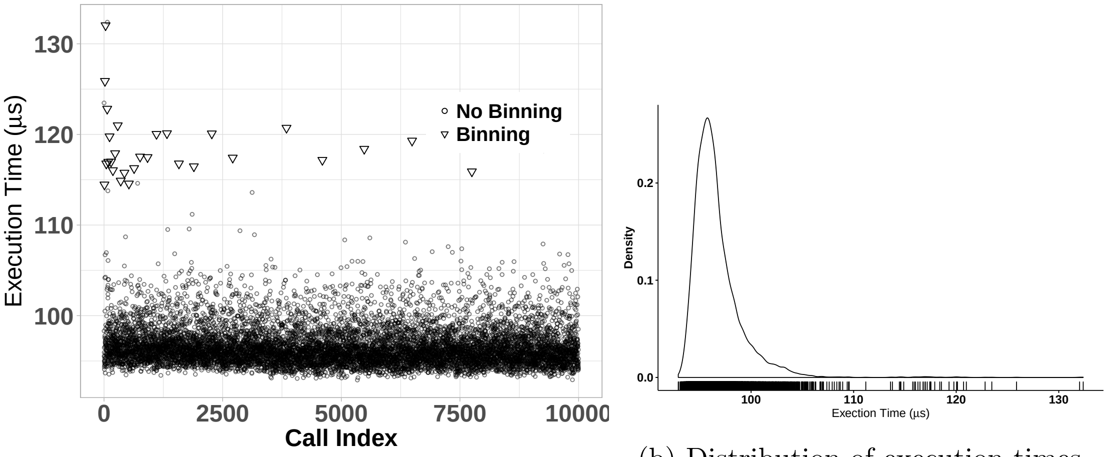
2.4.2 Analysis

The scatter plot in Figure 2.3a shows the execution time for each request. The outliers in the plot correspond to a peak in latency caused by binning. Figure 2.3b presents the distribution of the execution times. Preliminary exploratory data analysis on the results shows that the density approaches a log-normal distribution. The Mann-Whitney U test shows that the binning effect is statistically significant (P-value close to zero and $W = 7482$). Also, post hoc analysis shows an effect of magnitude was 23.79 μs, and the maximum latency observed when binning occurred was 147.96 μs.

The frequency pattern which binning occurs deserves some attention: it is not strictly periodic as the one observed in Figure 2.1a. This pattern indicates that Jemalloc performs

[8] Located in src/server.c

[9] One client is sufficient since this study does not covers overhead from network layers.



(a) Plot of the execution time for every `call` command ordered by index. Triangles represent commands affected by binning.

(b) Distribution of execution times.

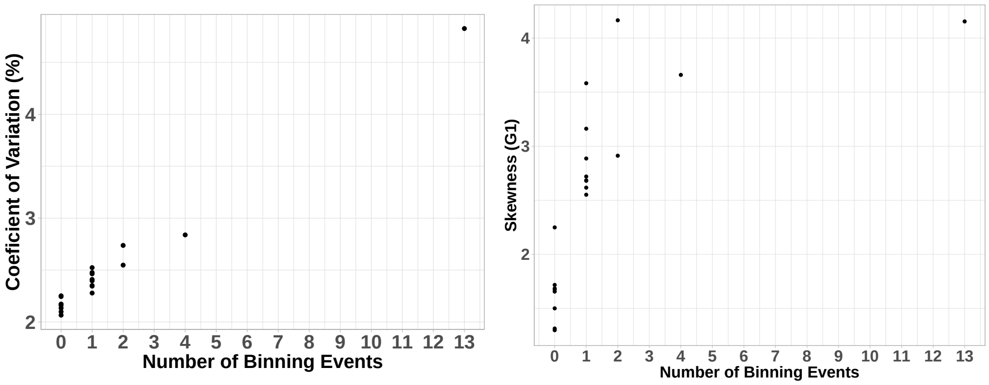
Figure 2.3: Results for the Redis case study.

an increasing over allocation every time more memory is needed. This behaviour is seen by the burst of binning events in the initial `LPUSH` requests, and it reduces once more commands are processed. In other words, the binning effect should be prevalent when not many memory chunks have been consumed.

Figure 2.4a shows the plot of the Relative Standard Deviation of 500 consecutive commands ($n = 500$) versus the number of binning events that occurred among them. According to the figure, when 13 out of 500 calls triggers binning the dispersion around the mean is almost 5%. To understand how the distribution changes, Figure 2.4b shows skewness versus the number of binning events. We observe the distribution skew increase significantly with the number of binning events. G_1 reaches 4.03 when 13 events occur and 2.6 when there is no binning (better symmetry).

2.5 CPython's Memory Allocator

Python has a dedicated object allocator which uses heap memory to store all objects and data structures that a Python program needs. The allocator, shown as abstraction level +2 in Figure 2.5, serves all small memory allocation requests for any Python object, except when the object has its dedicated allocator. Python memory manager uses the concept of



(a) Plot of the Performance variation in the presence of binning events.

(b) Plot of the distribution skewness in the presence of binning events.

Figure 2.4: Binning effect on the performance variability of `call` procedure.

an array of free lists to create (i) arenas, (ii) pools, and (iii) blocks. An arena comprises of a virtual memory area of 256 KiB which is divided into memory segments of 4 Kb in size called *pools* (therefore an arena has 64 pools in total). Figure 2.6a shows an arena with multiple pools.

Each pool is further segmented into smaller blocks of a fixed size class. When memory is requested, a block is taken from the pool according to the request size. Figure 2.6b shows a list of pools that store blocks of 8, 16 and 512 bytes, respectively. Figure 2.6c shows a pool with n memory blocks and blocks whose first two blocks are in use (grey). During the execution of a Python script, free blocks from these pools are allocated to store data. For example, every distinct *Integer* object in Python allocates one block from pools that stores objects of 25-32 bytes in size.

In this context, binning occurs in two moments: first, it happens when there are no pools with free blocks available for the requested size class. Then, a new pool has to be initialized and the additional overhead slows down the memory allocation request. Second, binning also occurs when there are no pools with free blocks and no free arenas to initialize a new pool. In this condition, more memory is requested from the operating system via a *mmap* system call and, consequently, a new pool is initialized. Function *_PyObject_Alloc* is called for every object allocation, except when an object has its specific allocator [10]. Dashed arrows in Figure 2.7 shows the occurrence of the binning events.

Any container object in Python is susceptible to binning and some examples are lists,

[10] The object-specific allocator can invoke *_PyObject_Alloc* when it needs to allocate more memory.

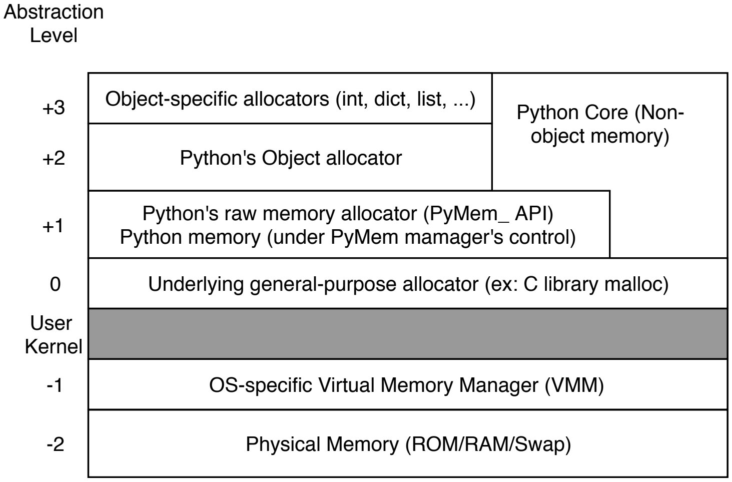
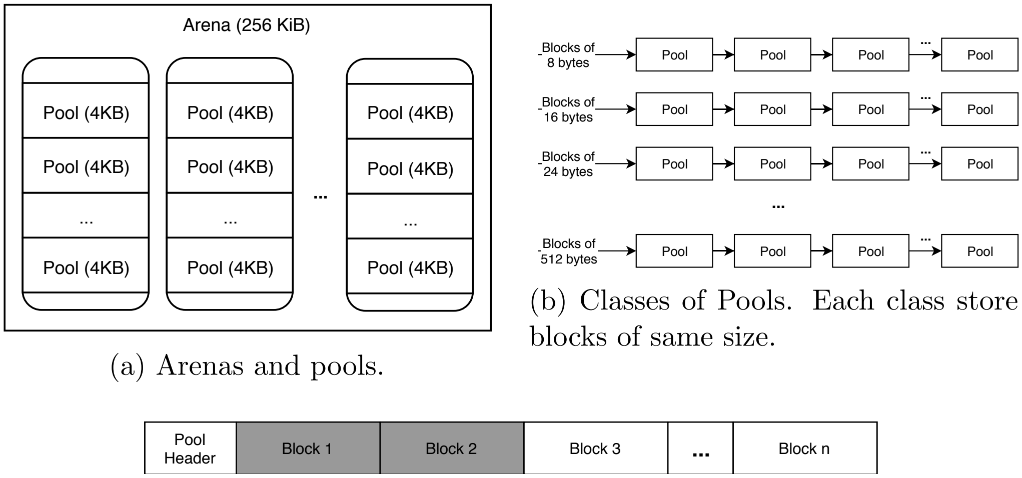

Abstraction Level
+3
Object-specific allocators (int, dict, list, ...)
Python Core (Non-object memory)
+2
Python's Object allocator
+1
Python's raw memory allocator (PyMem_ API)
Python memory (under PyMem mamager's control)
0
Underlying general-purpose allocator (ex: C library malloc)
User
Kernel
-1
OS-specific Virtual Memory Manager (VMM)
-2
Physical Memory (ROM/RAM/Swap)

Figure 2.5: Python Memory Layers [18].



(a) Arenas and pools.

(b) Classes of Pools. Each class store blocks of same size.

(c) Blocks in a pool. Blocks in gray are in use.

Figure 2.6: CPython memory management.

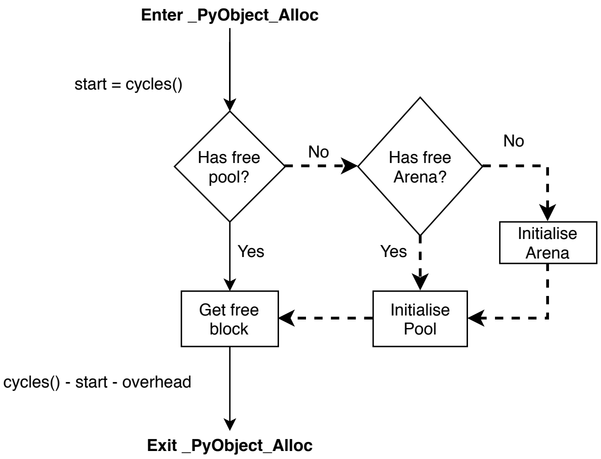


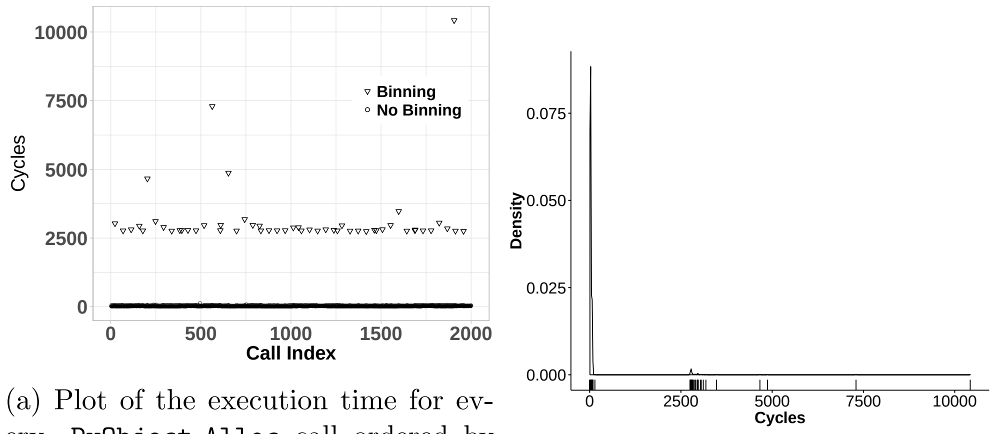
Figure 2.7: Diagram for Python's object allocator. Dashed arrows represent occurrence of binning.

sets and dictionaries. Binning occurs when the amount of objects they store increase and the allocator executes a routine to expand the object. This expansion routine represents an object-specific binning event which requests memory from the object's allocator. This type of binning will be investigated in Section 2.5.3.

2.5.1 Experimental Setup

To investigate the presence of binning we instrument CPython to measure the time taken for Python's object allocator complete a request. The time spent in *_PyObject_Alloc* function is obtained by counting the number of CPU clock cycles from function entry until it returns. With the *Time Stamp Counter Register* (TSC) we can count accurately the processor cycles and this register is accessed via *RDTSC* (read time stamp counter) assembly instruction. Then, we obtain the time spent in the function (in cycles) by computing the difference of cycles between function exit and entry minus the read overhead. The instrumentation overhead to read the register is calculated once, in the interpreter's initialization phase. Figure 2.7 shows the locations where the cycle count is collected during object allocation.

The workload is a synthetic program, shown in Listing 2.2. It uses Python 3.6 to perform two thousand consecutive append calls to a list object, and each one adds a `bytearray` object of 256 bytes to the list. We use a `bytearray` object to guarantee that a new block is allocated on every iteration. At some point in execution, the number of free



(a) Plot of the execution time for every _PyObject_Alloc call ordered by index. Triangles represent allocations affected by binning.

(b) Distribution of execution times.

Figure 2.8: Results for Python workload.

blocks, pools and arenas will be consumed, causing the memory allocator to request a new memory area from the operating system.

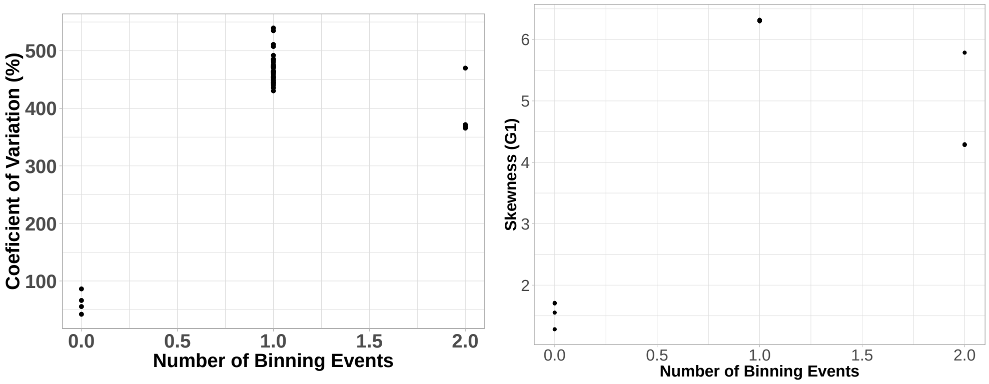
```
mylist = []
for i in range(2000):
    mylist.append(bytearray(256))
```

Listing 2.2: Workload for CPython case study.

2.5.2 Analysis

Figure 2.8a illustrates the latency of each object allocation call versus the call index. Calls which caused binning are significantly slower than the ones which did not trigger (seen as a black line in the plot). The P-value for Mann-Whitney U hypothesis test is zero, its test statistic, W, is 2.2e+8 and the effect magnitude was 3253.58 cycles according to Games-Howell post hoc analysis.

The density plot on Figure 2.8b reveals that the distribution has a high peak for execution times below 100 cycles and a significant amount of outliers as observed in Figure 2.8a. Calls which take 10k cycles or more represents initialization of new arenas.



(a) Plot of the Performance variation in the presence of binning events.

(b) Plot of the distribution skewness in the presence of binning events.

Figure 2.9: Binning effect on performance variability of Python object allocator.

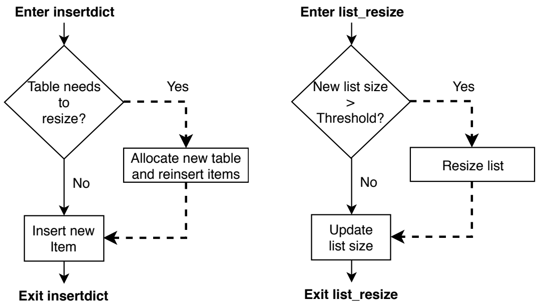
Figure 2.9 shows the plot of the Relative Standard Deviation versus the number of binning events for every 40 ($n = 40$) object allocation calls. According to the figure, when two out of 40 calls triggers binning the performance variability is above 300%. Figure 2.9b shows that the distribution skewness increases significantly with only two binning events.

2.5.3 Object-Specific Allocators

On top of Python's Object Allocator (level +2 in Figure 2.5) there are the *object-specific allocators* (level +3), which are specialized to the memory requirements of each object type such as dictionaries, sets and lists. Containers such as sets hold references to other Python objects, and they are initialized with a fixed size. Binning occurs when a container needs to expand to hold more references which cause the memory manager take a slower path. In this section, we investigate how binning is also present in these allocators, specially, we study the effect on the three most popular containers: dictionaries, sets and lists.

Dictionaries

A dictionary object in CPython is implemented as a hash table and elements are inserted in the table according to their hashed key. As described above, this container has to expand whenever there is not enough room to add a new item into the table (the binning event). The routine that performs the expansion allocates a new table and reinserts all elements



(a) Control flow of `insertdict` function.

(b) Control flow of `list_resize` function.

Figure 2.10: Simplified control flow diagrams for inserting new items in Python's dictionary and set objects. Dashed line indicate a slow path.

in it. Since memory needs to be allocated to grow the table and references are moved to the new table, this event can be several times slower than a normal insertion. Figure 2.10a shows a summary of the control flow described above and the dashed arrows indicate the slow path executed during table expansion.

We measured the time spent to insert items into the dictionary by instrumenting the interpreter to count the number of cycles spent in the CPython `insertdict` function [11]. This function handles the insertion of items in a dictionary and calls `dict_resize` when an expansion is needed. Also, the Python workload is similar to Algorithm 2.2, with the exception that items are inserted into a dictionary instead of a list [12]. The workload inserts items consecutively and eventually causes the hash table to expand.

Figure 2.11 shows the execution time for each insertion index and the peaks in execution time represent insertions that triggered table expansion. The P-value for Mann-Whitney test is zero with a test statistic of 1.6+6, and the effect magnitude was 30290 cycles according to Games-Howell post hoc analysis. The maximum latency observed when binning occurred was 753652 cycles. The frequency which binning occurs in dictionaries is similar to the one seen in Redis, where the memory manager over-allocates memory to approximately twice of the previous size.

[11] Located in Objects/dictobject.c file.

[12] We omitted the Python code for simplicity since it is similar to 2.2.

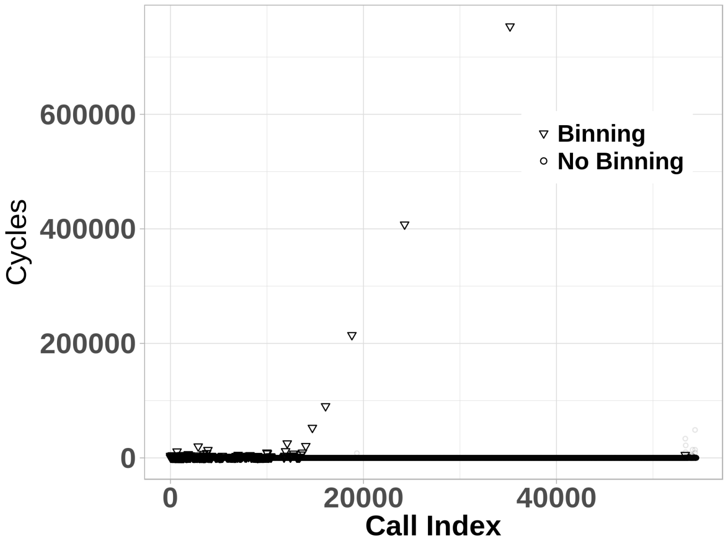


Figure 2.11: Plot of the number of cycles spent to insert one dictionary items. The data points are ordered by the insertion order index. Calls that did not trigger binning are significantly faster and are similar to a line in the bottom of the plot.

Sets

Set objects are similar to dictionaries since they are also implemented as a hash table. When elements are continuously inserted in the set, the table has to grow to fit more items which characterize the binning event. Equivalently, in the expansion routine, a new table is allocated and all items are reinserted. The control flow for inserting an item into a `set` object, handled by function `set_add_entry`, was omitted since it is similar to the one shown diagram in Figure 2.10a. The CPython routine that expands the hash table is `set_table_resize` [13].

We measured the time taken to insert elements in a set by counting the number of cycles spent in CPython `set_add_entry` function. The Python workload is similar to Algorithm 2.2, with the exception that we add unique elements to a set object instead of a list. The consecutive addition of elements causes the set to expand a few times. The Mann-Whitney hypothesis test shows that the binning effect is statistically significant (P value is 1.57e-21 and test statistic $W = 514923$). Also, the effect magnitude was 31824 cycles according to Games-Howell post hoc analysis, and the maximum execution time observed when binning occurred was 589480 cycles.

[13] Located in Objects/setobject.c file.

Lists

A *list* object is another container susceptible to binning. Before appending an item to the list, the algorithm first checks whether there is enough room for the new item. If the new list size is less than a threshold [14], the program executes a routine to resize it (binning event). The additional time to process the expansion is a consequence of binning. Figure 2.10b shows a simplified diagram for the program control flow.

We measure the time spent (in cycles) to compute the binning effect in the function `list_resize` [15], responsible for resizing the list when it needs to expand. The binning effect is statistically significant according to Mann-Whitney hypothesis test (P value is zero and test statistic, W equals to 38.8^6). The effect magnitude was 370 cycles according to Games-Howell post hoc analysis, and the maximum latency observed when binning occurred was 40828 cycles.

2.6 Detecting Binning Events on Python Programs

This section introduces a run-time analysis tool that instruments Python's interpreter and detects binning events on a Python program. It comprises of *Pin* [47] that attaches to the interpreter process and a and a *Pintool* that counts the occurrences of binning events during its execution. Besides, this tool is composed of a Python module that saves the program's stack trace whenever an event is detected.

The data collected shows how the code is affected by binning, and the stack trace help locating the most occurrences of binning in the source code (file and line number). Also, library developers, in particular, can use this information to perform code optimizations to avoid the program spend too much time on low-level memory management routines. Since the memory management is not thread-safe, object access must be done quickly because of the bottleneck imposed by the *Global Interpreter Lock* [17] in CPython.

Python provides the *tracemalloc* module [19] to describe memory usage statistics and trace allocations. In this study, a metric such as the number of blocks allocated by the program is not as relevant as the number of long-running allocation requests. Therefore, the analysis tool we create is concerned with the number of binning events occurred in the interpreter and not memory usage. The description of the tool is presented in the following sections, and Section 2.6.2 shows usage examples on the standard Python's benchmark suite.

[14] Half of its current size.

[15] Located in Objects/listobject.c file.

2.6.1 Pintool Implementation

Pin is a widely used Dynamic Binary Instrumentation (DBI) framework developed by Intel which allows creation of program analysis tools (*Pintools*) [69, 65]. It provides a C++ API to inject code in the program at any of the three levels: instruction, routine and image. We use *Pin* [16] to insert, at run-time, analysis routines that handles any of the binning events described in Section 2.5. Specifically, we count the number of (1) pool initialization, (2) new arenas created, (3) dictionary expansions, (4) list expansions, (5) set expansions and (6) memory maps system calls. To understand how the tool's design, two types of routines deserve attention: *Instrumentation Routines* and *Analysis Routines*.

Instrumentation Routine

The Pintool instrumentation routines define where and when in the interpreter process a jump to the analysis routine is inserted. Once *Pin* attaches to the interpreter, the instrumentation routine check whether the function about to be called is one of the functions in column *CPython Function* in Table 2.1 (these are functions source of binning discussed in Section 2.5). If the function is `new_arena`, `dictresize` or `set_table_resize`, the analysis routine is executed before them, that is, a jump is made before calling `new_arena` for instance.

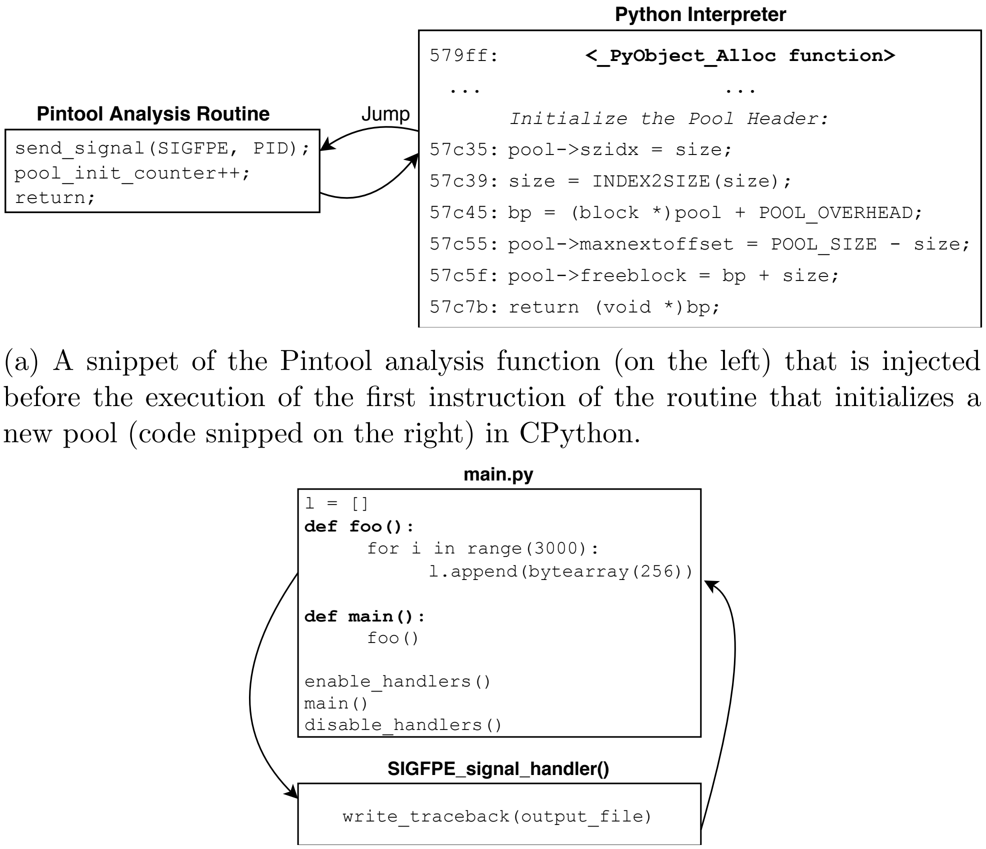
In contrast, to count binning events caused by pool initialization and list resize in functions `_PyObject_Alloc` and `list_resize`, respectively, the instrumentation routine adds a jump before a specific instruction address [17]. Figure 2.12a shows a code snipped that initializes a new pool. In the figure, a call to the analysis routine is inserted at instruction address *57c35* which causes the program control flow to jump before executing the corresponding instruction.

Analysis Routines

Analysis routine is a user-defined procedure dynamically added to the interpreter process which is called when a binning event occurs. In this work, there is one analysis routine for every source of binning. When the analysis routine is invoked, it increments the counter for the specific event and emits a signal to the python program. Signals are a type of Inter-Process Communication (IPC) that allows *Pin* notify the user application when binning occurred. Table 2.1 contains the signals for every binning event discussed in this study.

[16] Pin version 3.7.

[17] The addresses values are hard-coded in the tool.



```
send_signal(SIGFPE, PID);
pool_init_counter++;
return;
```

```
579ff:       <_PyObject_Alloc function>
  ...                   ...
       Initialize the Pool Header:
57c35: pool->szidx = size;
57c39: size = INDEX2SIZE(size);
57c45: bp = (block *)pool + POOL_OVERHEAD;
57c55: pool->maxnextoffset = POOL_SIZE - size;
57c5f: pool->freeblock = bp + size;
57c7b: return (void *)bp;
```

(a) A snippet of the Pintool analysis function (on the left) that is injected before the execution of the first instruction of the routine that initializes a new pool (code snipped on the right) in CPython.

```
l = []
def foo():
      for i in range(3000):
            l.append(bytearray(256))

def main():
      foo()

enable_handlers()
main()
disable_handlers()
```

```
write_traceback(output_file)
```

(b) Example of a Python program with a handler to the SIGFPE signal.

Figure 2.12: Tool design.

Table 2.1: Sources of binning events and corresponding signals.

Source	Description	Function	Signal
Pool Initialization	No free pools	_PyObject_Alloc	8 SIGFPE
New Arena	No arenas available	new_arena	10 SIGUSR1
Resize Dictionary	Dictionary needs to expand	dictresize	6 SIGABRT
Resize Set	Set needs to expand	set_table_resize	5 SIGTRAP
Resize List	List needs to expand	list_resize	7 SIGBUS

Figure 2.12a shows an example of a call to an analysis routine that occurs before a pool initialization. The routine emits a SIGFPE signal to the Python program and increments the `pool_init_counter` variable that holds the number of pool initializations. The remaining binning types have similar routine. Figure 2.12b illustrates the Python program under analysis. When the application receives a signal from the operating system, it calls a handler function that writes the traceback to a file.

The tool comes with a Python module called `dumptrace.py` that registers the handlers for signals listed in Table 2.1. To use it, the developer needs to import the module and call functions `enable_handlers()` and `disable_handlers()` before and after executing the main Python routine, respectively, as shown in Listing 2.3. When a signal is caught, the stack trace is recorded and the program resumes execution.

```
import dumptrace

dumptrace.enable_handlers()
main()
dumptrace.disable_handlers()
```

Listing 2.3: Registering signal handlers in a Python program.

Tool Execution and Output

The pintool attaches to the Python interpreter and executes the analysis routines until the target program finishes. Listing 2.4 shows how the tool is invoked.

```
1 $ python main.py & PID=$!
2 $ pin -follow_execv -pid $PID -t pintool.so & wait
```

Listing 2.4: Command to execute the pintool.

The first command starts the Python program we want to analyze. The `dumprace.py` module makes the program sleep for 5 seconds to avoid counting binning events in the interpreter's initialization phase and to give enough time to the user attach the pintool. Command 2 attaches the tool to the interpreter and wait until it finishes. Parameters *-t* and *-pid* expects the path to the pintool image and the interpreter's process id [18]. Once the program finishes, the tool writes the counter values and the python program writes the tracebacks to local files.

[18] Parameter *-follow_execv* enables instrumentation of child process.

2.6.2 Binning Events in Python Performance Tests

We analyse the Tests in the Python Performance Benchmark Suite [22] using the *pintool* described above. Also, we adapted the benchmarks to avoid detection of binning events in the test initialization phase [19]. Table 2.2 shows the amount of events for each test and binning type. *Sympy*, a library for symbolic mathematics, is the module that is most affected by binning, specifically *expand* benchmark contains more than 100 thousand dictionaries resizes and more than 240 thousand lists resizes.

For benchmark *Go*, the top five occurrences of *list resize* events are shown in Listing 2.5 (post processed output file). The listing informs the developer the file path, line number, function, quantity and percentage of occurrences. Algorithm 2.6 shows the the `move` function in *bm_go.py* that triggered the majority of *list resizes* (2136 events in line 192), which corresponds to the following append call: `self.history.append(pos)`.

```
File ”benchmarks/bm_go.py”, line 192 in move: 2136, 27.6720%
File ”benchmarks/bm_go.py”, line 326 in <listcomp>: 1809, 23.4357%
File ”benchmarks/bm_go.py”, line 241 in <listcomp>: 1809, 23.4357%
File ”benchmarks/bm_go.py”, line 130 in remove: 846, 10.9600%
File ”benchmarks/bm_go.py”, line 38 in <listcomp>: 162, 2.0987%
```

Listing 2.5: *List Resize* events for Go benchmark.

```
180 def move(self, pos):
181     square = self.squares[pos]
182     if pos != PASS:
183         square.move(self.color)
184         self.emptyset.remove(square.pos)
185     elif self.lastmove == PASS:
186         self.finished = True
187     if self.color == BLACK:
188         self.color = WHITE
189     else:
190         self.color = BLACK
191     self.lastmove = pos
192     self.history.append(pos)
```

Listing 2.6: Code snippet in `bm_go.py` benchmark that caused most *List Resize* events. 27.67% of binning events occured in line 192.

[19] We do not want to capture events in the interpreter's initialization or benchmark setup phases.

Table 2.2: Count of binning events occurred in the Python benchmark.

Benchmark	**Binning Type**				
	New Arena	**Init Pool Header**	**List Resize**	**Dict Resize**	**Set Resize**
Sympy sum	208	5047	83175	7175	1921
Sympy str	235	2077	89721	36344	1184
Sympy expand	38	1402	240528	112457	75
Sympy integrate	33	2837	5664	1348	326
Genshi xml	26	1145	50299	2038	2
Chameleon	6	284	581	174	0
Deltablue	2	75	1745	5	0
Pyaes	0	7	2	3	0
Django	0	27	21848	108	0
Dulwich	2	138	7853	24	241
Fannkuch	0	2	13	3	0
Genshi text	8	421	2336	2035	2
Go	0	187	7728	248	598
Json dumps	16	1396	75	1002	0
Json loads	0	82	171	402	0
Pickle	0	1	11	3	0
Unpickle	0	8	71	243	0
Pickle List	0	2	11	3	0
Unpickle List	0	2	321	3	0
Pickle Dict	0	2	11	3	0
Raytrace	1	7	45789	5	0
Regex Compile	0	1211	56893	2801	0
Regex Effbot	0	2	10	3	0
Regex V8	2	248	11755	4	0

2.7 Discussion and Future Work

In this section we discuss the known issues of the tool described above and present an example of how developers could optimize their code when one of the binning types is detected. Besides, we discuss potential sources of binning in CPython and how our study can be extended to other applications.

Pintool Known Issues

Obtaining the application's stack trace from the instrumented CPython is challenging. One of the consequences of using system signals is the possibility that some of the stack traces show the wrong line number (e.g., the next line where the event occurs) or few of them are not recorded in the file in case there is a burst of binning events. The reason is that the interpreter's main thread might handle the received signal only at a later point.

Further, the overhead in the program caused by the tool is proportional to the number of binning events detected. For instance, the *Float* Python benchmark is two times slower when we only count the binning events (trace dump disabled) and 40 times slower when the stack trace is also written to the output files. However, as seen in Section 2.6.2 the stack traces provided by the current implementation are still very useful for detecting and locating binning events. Also, we believe that the design can be improved using the *Pin* API to access the interpreter's context and retrieve, from the frame in the top of the stack, the file and line number of the current bytecode without the need for system signals. Our initial work shows that manipulating CPython objects in the *Pintool* image is not straightforward and more work is needed to verify whether or not this is a viable solution.

Optimization Example

The code snippet 2.6 shows an example of list manipulation that caused multiple binning events. If the final list size is known, the developer could initialize the list with an initial size to reduce binning. Also, even if the developer does not know the exact size, he could estimate an average final size for the list and pre-allocate the elements.

To exemplify the optimization above, we provide a simple example in Listings 2.7 and 2.8 which shows two code snippets that inserts 10k elements to a list. To add each element, Listing 2.7 uses `append` calls, and 2.8 first initializes the list and inserts them at a specific index. We verify the performance difference of the two approaches by counting the number of cycles spent on each one and the number of binning events using the *Pintool* implemented in this study [20]. The *List Resize* count for the unoptimized and optimized versions are 52 and 6, respectively. The cycle count difference is statistically significant (P-value < 0.01) and post hoc analysis shows that Listing 2.8 spend approximately 1M fewer cycles than the unoptimized version.

```python
size = 10000
l = []
for i in range(size):
    l.append(i)
```

Listing 2.7: Unoptimized code.

```python
size = 10000
l = [None]*size
for i in range(size):
    l[i] = i
```

Listing 2.8: Optimized code.

Additional Sources of Binning

In this work, we investigated five sources of binning in CPython in which three are from the most popular containers available in the language. We believe that other objects, such as tuples, can also be affected by binning, and finding additional sources increase the opportunities for code optimizations. However, further investigation is needed to verify whether or not they exist.

Tool Generalization

The design of the *Pintool* implemented in this study can be adapted to any application in user-level, including programs that use Jemalloc as memory manager. Also, *Perf Probes* can be used if analysis of binning events in the Linux Kernel is necessary instead. For instance, one of its possible applications is tracing and filtering block IO requests that might trigger binning. Since these requests can be queued, merged, split bounced, among others [2], we believe that it is a potential source of performance variability of *write* or *read* system calls.

2.8 Conclusion

This chapter investigates the effect of binning on memory allocators in the Kernel, Redis and CPython. Binning is an event that occurs when a particular condition of a Bin

[20] This experiment was repeated ten times.

is not met and it causes the memory manager executes a slower routine to serve the allocation request. We have shown that the additional execution time to complete the request increase variability of memory allocation by 4.7% and 500% in Redis and CPython, respectively. Further, we show that manipulation of Python container objects is slower when the binning occurs. This finding motivated the creation of a *Pintool* that detects binning events in a Python application and saves the program's stack trace. These traces aid library developers finding the location where the events occur and expose opportunities for code optimizations.

References

[1] Juan Pablo Sandoval Alcocer and Alexandre Bergel. Tracking performance failures with rizel. In *Proceedings of the 2013 International Workshop on Principles of Software Evolution*, pages 38–42. ACM, 2013.

[2] Michael Beck, Robert Magnus, and Ulrich Kunitz. *Linux Kernel Internals with Cdrom*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[3] Dirk Beyer, Stefan Löwe, and Philipp Wendler. Reliable benchmarking: Requirements and solutions. *International Journal on Software Tools for Technology Transfer*, pages 1–29, 2017.

[4] Cor-Paul Bezemer, Simon Eismann, Vincenzo Ferme, Johannes Grohmann, Robert Heinrich, Pooyan Jamshidi, Weiyi Shang, André van Hoorn, Monica Villaviencio, Jürgen Walter, et al. How is performance addressed in devops? a survey on industrial practices. *arXiv preprint arXiv:1808.06915*, 2018.

[5] Jeff Bonwick et al. The slab allocator: An object-caching kernel memory allocator. In *USENIX summer*, volume 16. Boston, MA, USA, 1994.

[6] Jonathan Bonwick. Magazines and vmem: Extending the slab allocator to many cpus and arbitrary resources. In *USENIX-01*, 2001.

[7] Andrew P Bradley. The use of the area under the roc curve in the evaluation of machine learning algorithms. *Pattern recognition*, 30(7):1145–1159, 1997.

[8] Leo Breiman. Random forests. *Machine Learning*, 45(1):532, Oct 2001.

[9] Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. Parallel symbolic execution for automated real-world software testing. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 183–198, New York, NY, USA, 2011. ACM.

[10] Brian S Cade and Barry R Noon. A gentle introduction to quantile regression for ecologists. *Frontiers in Ecology and the Environment*, 1(8):412–420, 2003.

[11] J. Chen and W. Shang. An exploratory study of performance regression introducing code changes. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 341–352, Sept 2017.

[12] Jiahao Chen and Jarrett Revels. Robust benchmarking in noisy environments. *arXiv preprint arXiv:1608.04295*, 2016.

[13] Kevin J Davey. *Building Winning Algorithmic Trading Systems: A Trader's Journey from Data Mining to Monte Carlo Simulation to Live Trading*. John Wiley & Sons, 2014.

[14] Augusto Born De Oliveira, Sebastian Fischmeister, Amer Diwan, Matthias Hauswirth, and Peter F Sweeney. Why you should care about quantile regression. In *ACM SIGPLAN Notices*, volume 48, pages 207–218. ACM, 2013.

[15] Augusto Born de Oliveira, Jean-Christophe Petkovich, and Sebastian Fischmeister. How much does memory layout impact performance? a wide study. In *Intl. Workshop Reproducible Research Methodologies*, pages 23–28, 2014.

[16] Brotli Developers. Brotli github page. https://github.com/google/brotli. Accessed: Jan 10, 2019.

[17] CPython Developers. Global interpreter lock. https://wiki.python.org/moin/GlobalInterpreterLock. Accessed: 2019-03-17.

[18] CPython Developers. Python's object allocator source code. https://github.com/python/cpython/blob/3.6/Objects/obmalloc.c. Accessed: 2019-03-01.

[19] Python Developers. Python tracemalloc module. https://docs.python.org/3.6/library/tracemalloc.html. Accessed: Jan 10, 2019.

[20] Redis Developers. How fast is redis? https://redis.io/topics/benchmarks. Accessed: 2019-03-20.

[21] Elfriede Dustin, Jeff Rashka, and John Paul. *Automated software testing: introduction, management, and performance*. Addison-Wesley Professional, 1999.

[22] Alex Gaynor et al. The python performance benchmark suite. https://pyperformance.readthedocs.io. Accessed: 2019-03-17.

[23] Jason Evans. A scalable concurrent malloc (3) implementation for freebsd. In *Proc. of the bsdcan conference, ottawa, canada*, 2006.

[24] D. G. Feitelson, E. Frachtenberg, and K. L. Beck. Development and deployment at facebook. *IEEE Internet Computing*, 17(4):8–17, July 2013.

[25] Philip J Fleming and John J Wallace. How not to lie with statistics: the correct way to summarize benchmark results. *Communications of the ACM*, 29(3):218–221, 1986.

[26] Tristan Fletcher. Support vector machines explained. *Tutorial paper*, 2009.

[27] Martin Fowler and Jim Highsmith. The agile manifesto. *Software Development*, 9(8):28–35, 2001.

[28] Amir Reza Ghods. A study of linux perf and slab allocation sub-systems. Master's thesis, University of Waterloo, http://hdl.handle.net/10012/10184, 2015.

[29] Milos Gligoric, Rupak Majumdar, Rohan Sharma, Lamyaa Eloussi, and Darko Marinov. Regression test selection for distributed software histories. In *International Conference on Computer Aided Verification*, pages 293–309. Springer, 2014.

[30] B. Gregg. *Systems Performance: Enterprise and the Cloud*. Always learning. Prentice Hall, 2014.

[31] Torsten Hoefler and Roberto Belli. Scientific benchmarking of parallel computing systems: twelve ways to tell the masses when reporting performance results. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*, page 73. ACM, 2015.

[32] Michael Httermann. *DevOps for developers*. Apress, 2012.

[33] Jin Huang and C. X. Ling. Using auc and accuracy in evaluating learning algorithms. *IEEE Transactions on Knowledge and Data Engineering*, 17(3):299–310, March 2005.

[34] Peng Huang, Xiao Ma, Dongcai Shen, and Yuanyuan Zhou. Performance regression testing target prioritization via performance risk analysis. In *Proceedings of the 36th International Conference on Software Engineering*, pages 60–71. ACM, 2014.

[35] Peter Isley. The title of the work. How it was published, 7 1993. An optional note.

[36] Raj Jain. *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling*. John Wiley & Sons, 1990.

[37] DN Joanes and CA Gill. Comparing measures of sample skewness and kurtosis. *Journal of the Royal Statistical Society: Series D (The Statistician)*, 47(1):183–189, 1998.

[38] Tomas Kalibera, Lubomir Bulej, and Petr Tuma. Benchmark precision and random initial state. In *in Proceedings of the 2005 International Symposium on Performance Evaluation of Computer and Telecommunications Systems (SPECTS 2005*, pages 853–862. SCS, 2005.

[39] Tomas Kalibera and Richard Jones. Rigorous benchmarking in reasonable time. In *Proceedings of the 2013 International Symposium on Memory Management*, ISMM '13, pages 63–74, New York, NY, USA, 2013. ACM.

[40] Jim Keniston, Ananth Mavinakayanahalli, Prasanna Panchamukhi, and Vara Prasad. Ptrace, utrace, uprobes: Lightweight, dynamic tracing of user apps. In *Proceedings of the 2007 Linux symposium*, pages 215–224, 2007.

[41] Roger Koenker and Kevin Hallock. Quantile regression: An introduction. *Journal of Economic Perspectives*, 15(4):43–56, 2001.

[42] Redis Labs. Introduction to redis, 2018.

[43] Benjamin Le Cook and Willard G Manning. Thinking beyond the mean: a practical guide for using quantile regression methods for health services research. *Shanghai archives of psychiatry*, 25(1):55, 2013.

[44] Philipp Leitner and Cor-Paul Bezemer. An exploratory study of the state of practice of performance testing in java-based open source projects. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ICPE '17, pages 373–384, New York, NY, USA, 2017. ACM.

[45] Eckhard Limpert, Werner A. Stahel, and Markus Abbt. Log-normal distributions across the sciences: Keys and clueson the charms of statistics, and how mechanical models resembling gambling machines offer a link to a handy way to characterize log-normal distributions, which can provide deeper insight into variability and probabilitynormal or log-normal: That is the question. *BioScience*, 51(5):341–352, 2001.

[46] Ran Liu and Haibo Chen. Ssmalloc: a low-latency, locality-conscious memory allocator with stable performance scalability. *APSys*, 12:15–15, 2012.

[47] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building

customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, volume 40, pages 190–200. ACM, 2005.

[48] Qi Luo, Denys Poshyvanyk, and Mark Grechanik. Mining performance regression inducing code changes in evolving software. In *Mining Software Repositories (MSR), 2016 IEEE/ACM 13th Working Conference on*, pages 25–36. IEEE, 2016.

[49] Matt Mahoney. Large text compression benchmark. http://mattmahoney.net/dc/text.html. Accessed: Mar 07 2019.

[50] Django Maintainers. Django web page. https://www.djangoproject.com/. Accessed: 2019-10-10.

[51] Jq Maintainers. Jq web page. https://stedolan.github.io/jq/. Accessed: 2019-01-10.

[52] Linux Kernel Maintainers. Kernel probes (kprobes. https://www.kernel.org/doc/Documentation/kprobes.txt. Accessed: 2019-01-20.

[53] Linux Kernel Maintainers. Slab allocator. https://www.kernel.org/doc/gorman/html/understand/understand011.html. Accessed: 2019-01-05.

[54] Linux Perf Maintainers. Perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page. Accessed: 2019-01-05.

[55] Francisco Javier Mesa-Martinez, Ehsan K Ardestani, and Jose Renau. Characterizing processor thermal behavior. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 193–204. ACM, 2010.

[56] Douglas C Montgomery. *Design and analysis of experiments*. John wiley & sons, 2017.

[57] Peter L Morse. Computer method and system for allocating and freeing memory utilizing segmenting and free block lists, October 1 1996. US Patent 5,561,786.

[58] Rajendrani Mukherjee and K Sridhar Patnaik. A survey on different approaches for software test case prioritization. *Journal of King Saud University-Computer and Information Sciences*, 2018.

[59] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing wrong data without doing anything obviously wrong! *SIGPLAN Not.*, 44(3):265–276, March 2009.

[60] Todd Mytkowicz, Peter F Sweeney, Matthias Hauswirth, and Amer Diwan. Observer effect and measurement bias in performance analysis. *Computer Science Technical Reports CU-CS-1042-08, University of Colorado, Boulder*, 2008.

[61] Thanh HD Nguyen, Meiyappan Nagappan, Ahmed E Hassan, Mohamed Nasser, and Parminder Flora. An industrial case study of automatically identifying performance regression-causes. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 232–241. ACM, 2014.

[62] 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.

[63] Augusto Oliveira, Sebastian Fischmeister, Amer Diwan, Matthias Hauswirth, and Peter Sweeney. Perphecy: Performance regression test selection made simple but effective. In *Proc. of the 10th IEEE International Conference on Software Testing, Verification and Validation (ICST)*, Tokyo, Japan, 2017.

[64] John Ousterhout. Always measure one level deeper. *Commun. ACM*, 61(7):74–83, June 2018.

[65] Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, and James Cownie. Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 2–11. ACM, 2010.

[66] Mathias Payer. Too much pie is bad for performance. *Technical report*, 766, 2012.

[67] Manjula Peiris and James H Hill. Automatically detecting excessive dynamic memory allocations software performance anti-pattern. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, pages 237–248. ACM, 2016.

[68] Jean-Christophe Petkovich, A Oliveira, Y Zhang, Thomas Reidemeister, and Sebastian Fischmeister. Datamill: a distributed heterogeneous infrastructure forrobust experimentation. *Software: Practice and Experience*, 46(10):1411–1440, 2016.

[69] Paruj Ratanaworabhan, Martin Burtscher, Darko Kirovski, Benjamin Zorn, Rahul Nagpal, and Karthik Pattabiraman. Detecting and tolerating asymmetric races. *ACM sigplan notices*, 44(4):173–184, 2009.

[70] James Roche. Adopting devops practices in quality assurance. *Communications of the ACM*, 56(11):38–43, 2013.

[71] Gregg Rothermel and Mary Jean Harrold. Analyzing regression test selection techniques. *IEEE Transactions on software engineering*, 22(8):529–551, 1996.

[72] Gregg Rothermel, Mary Jean Harrold, and Jeinay Dedhia. Regression test selection for c++ software. *Software Testing, Verification and Reliability*, 10(2):77–109, 2000.

[73] Ripon K Saha, Lingming Zhang, Sarfraz Khurshid, and Dewayne E Perry. An information retrieval approach for regression test prioritization based on program changes. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 268–279. IEEE Press, 2015.

[74] David Skinner and William Kramer. Understanding the causes of performance variability in hpc workloads. In *Workload Characterization Symposium, 2005. Proceedings of the IEEE International*, pages 137–149. IEEE, 2005.

[75] Victor Szalvay. An introduction to agile software development. *Danube technologies*, 3, 2004.

[76] Linus Torvalds. Git source code mirror. `https://github.com/git/git`. Accessed: Jan 10, 2019.

[77] J. Vitek and T. Kalibera. Repeatability, reproducibility and rigor in systems research. In *2011 Proceedings of the Ninth ACM International Conference on Embedded Software (EMSOFT)*, pages 33–38, Oct 2011.

[78] Nicholas J. Wright, Shava Smallen, Catherine Mills Olschanowsky, Jim Hayes, and Allan Snavely. Measuring and understanding variation in benchmark performance. In *DoD High Performance Computing Modernization Program Users Group Conference (HPCMP-UGC), 2009*, pages 438–443. IEEE, 2009.