DISS. ETH No. 25104

BEYOND PROFILING: PRACTICAL APPROACHES TO EXPOSE PERFORMANCE ANOMALIES

A dissertation submitted to attain the degree of
DOCTOR OF SCIENCES of ETH ZURICH
(Dr. sc. ETH Zurich)

presented by

LUCA DELLA TOFFOLA
Laurea magistrale, Università della Svizzera italiana
born on 06.10.1985
citizen of Lugano, TI – Switzerland

accepted on the recommendation of
Prof. Thomas R. Gross (examiner)
Prof. Matthias Hauswirth (co-examiner)
Prof. Shan Lu (co-examiner)
Prof. Michael Pradel (co-examiner)

2018

ABSTRACT

The common practice among developers is to use a CPU time profiler to inspect a program that shows sub-optimal performance. A problem of CPU time profilers is that they indicate to the developer the code locations of the program where the time is spent, but not the code locations where the time is wasted. Consequently a CPU time profiler offers limited help to find the cause(s) of a performance bottleneck and it offers limited help to a developer to fix such performance issue(s) in a program.

A major drawback of using CPU time profilers to debug a program's performance is that a CPU time profiler cannot pin-point the exact code location(s) that cause(s) a performance issue. A performance issue may manifest in one code location (e. g., a slow function execution) but the source of the problem (e. g., setting a function's argument to a specific value) can be located in a different code location.

To identify and to fix the cause(s) of a performance problem a developer may execute multiple manual costly tasks. For example, a developer manually inspects the profiler reports to identify the code locations that may contain the cause(s) of a slow execution, and then he or she fixes the performance problem. To fix a performance problem, a developer is required to perform multiple iterations of code changes until he or she finds a suitable fix, i. e., to test the different implementations of the fix. Additionally to check that the fix does not introduce any performance regression, the developer must execute the program with the bottleneck triggering input and exercise the fix with new inputs.

In this dissertation we present and implement three novel approaches that aim to mitigate these issues: MemoizeIt, PerfSyn, and TestMiner. The main goal of these novel approaches is to reduce the manual work required by a developer during the performance debugging cycle.

To solve the problem of finding the exact program locations with a performance problem, we introduce MemoizeIt, a dynamic program analysis that records runtime execution traces and that suggests to the developer code locations that may benefit from memoization. Applying memoization to a code location that performs redundant computations is one of the most effective techniques to speedup a program's execution. However identifying these locations is difficult because of the pervasive presence of side-effects, and because a conservative program analysis may discard profitable memoization opportunities (i. e., the analysis is too pessimistic). MemoizeIt takes a non-conservative, opposite, approach. MemoizeIt identifies memoization opportunities by comparing inputs and outputs of method calls in a scalable yet precise way, by iteratively increasing the level of details at which objects are compared. We show that MemoizeIt finds previously unknown performance bottlenecks such as redundant computations that would be otherwise discarded by a conservative analysis.

The second approach, PerfSyn, is an automatic program synthesis technique to generate bottleneck-exposing programs for a given method under test. PerfSyn solves the problem of manually providing inputs to validate a developer's assumptions about the performance of a method.

The idea behind PerfSyn is to repeatedly mutate a program that uses the method under test to systematically increase the amount of work done by the method. We formulate the problem of synthesizing a bottleneck-exposing program as a combinatorial search, and we show that it can be effectively and efficiently addressed using well known graph search algorithms. Our evaluation shows that PerfSyn is capable to synthesize programs that expose previously unknown and known performance bottlenecks.

Unfortunately PerfSyn suffers from a major limitation: it fails to provide relevant domain-specific input values, as a result, PerfSyn is effective for generic classes, such as collections, but less successful for domain-specific software.

To address this limitation we introduce TestMiner. The key idea of TestMiner is to exploit the information available in existing code bases, in particular in existing tests. TestMiner uses an information retrieval inspired mining technique to predict input values suitable for a particular method. The approach extracts literals from the source code of existing tests and it indexes them for quick retrieval. The indexed values can be used by a program generator that queries the mined data for values suitable for a given method under test. We show that TestMiner is able to increase code coverage of a state-of-the-art random test generator.

Once a developer identifies a code location that has sub-optimal performance, the developer must find a fix for the performance issue. We extend MemoizeIt to infer cache configurations from executions and to present to the developer the configurations in a textual form. Our hypothesis that the suggested fixes are effective in speeding-up a program's execution is validated by an evaluation of the caching configurations with real-world programs. The evaluation shows that applying the fixes to a series of bottleneck containing programs results in statistically significant speed-ups with different inputs.

SOMMARIO

Gran parte dell'analisi delle prestazioni di un programma è basata sull'uso di un profiler. L'obiettivo di un profiler è di quantificare il tempo che viene speso in ogni singola funzione del programma quando è eseguito con un determinato input.

Le attuali tecniche di profiling sono tuttavia affette da importanti limitazioni. La prima limitazione di un profiler è di indicare allo sviluppatore dove un programma spende il tempo durante un'esecuzione ma non come questo tempo viene inutilmente consumato. Di conseguenza, un profiler è di minima utilità per lo sviluppatore se lo sviluppatore deve cercare le cause di un collo di bottiglia in un programma. La seconda limitazione di un profiler è di offrire un limitato supporto in aiuto allo sviluppatore per rimuovere un problema di performance. Per esempio, un profiler non aiuta lo sviluppatore a trovare un'implementazione più efficiente di una funzione.

L'incapacità di indicare con precisione la locazione delle cause di un collo di bottiglia limita l'utilità di un profiler alla comprensione delle cause della performance non ottimale di un programma. Il primo problema, cioè la determinazione delle cause di un collo di bottiglia, è principalmente dovuto alla difficolta di correlare quest'ultime coi loro effetti. Un problema di performance può manifestarsi in una specifica parte del programma (per esempio, in una funzione che richiede troppo tempo per essere eseguita), ma la causa della lenta esecuzione (per esempio, un valore non ottimale assegnato ad un argomento della funzione) può essere situata in una diversa parte del programma.

Il secondo problema, quello di identificare ed eliminare il collo di bottiglia, richiede allo sviluppatore di eseguire molteplici e costosi compiti. Per esempio, uno sviluppatore deve ispezionare i report generati da un profiler e manualmente identificare il codice che è causa di un'esecuzione lenta. In una fase finale lo sviluppatore per risolvere il problema di performance deve applicare delle modifiche al codice senza alterarne la correttezza. Per questo motivo lo sviluppatore deve modificare il codice sorgente eseguendo diverse iterazioni di test per verificare configurazioni multiple del nuovo programma. Infine, lo sviluppatore deve validare il programma ottimizzato con nuovi input in modo da verificare che le modifiche non introducano alcun nuovo collo di bottiglia. Per concludere, lo sviluppatore deve eseguire il programma con i vecchi input che esibiscono il problema di performance verificando che il problema di performance è stato eliminato.

In questa dissertazione, proponiamo tre nuovi approcci che mirano a mitigare le problematiche appena descritte, e la loro implementazione in tre differenti sistemi: MemoizeIt, PerfSyn, e TestMiner.

MemoizeIt elimina il problema di trovare la locazione esatta di un problema di performance attraverso un' analisi dinamica dell'esecuzione di un programma. Tale analisi suggerisce allo sviluppatore indicazioni precise su locazioni del programma a cui si può apportare tecniche di memoizazation. Applicare questa ottimizzazione a programmi che eseguono computazioni ridondanti è una tecnica effettiva per velocizzare l'esecuzione di un programma. Tuttavia, identificare queste porzioni di codice è difficile poiché un analisi dell'esecuzione del programma conservativa può scartare delle opportunità di ottimizzazione se troppo pessimistica. Al contrario, MemoizeIt esegue un' analisi non conservativa identificando le opportunità di memoization osservando i dati elaborati ed i dati prodotti da una chiamata di una funzione, in modo scalabile ma preciso. MemoizeIt incrementa iterativamente il livello di precisione con cui gli oggetti elaborati da una funzione sono comparati per identificare funzioni con comportamento ridondante. In questa dissertazione dimostriamo che MemoizeIt è capace di trovare dei bug di performance che erano precedentemente sconosciuti agli sviluppatori dei programmi utilizzati nei nostri esperimenti.

Il secondo sistema PerfSyn implementa un approccio automatico di sintesi di programmi per l'identificazione di colli di bottiglia. PerfSyn risolve il problema di dover fornire manualmente input per verificare la performance di una funzione. L'idea alla base di PerfSyn è di mutare sistematicamente e ripetutamente un programma in modo da aumentare il carico di lavoro che la funzione esegue. Formulando il problema di sintesi come un problema di ottimizzazione, dimostriamo che PerfSyn è capace di sintetizzare programmi in poco tempo utilizzando algoritmi di ricerca basati su grafi. I nostri risultati dimostrano che PerfSyn è capace di sintetizzare programmi che espongono problemi di performance dovuti a scelte di algoritmi non ottimali dove l'implementazione di una funzione ha una eccessiva complessità computazionale, e problemi di performance introdotti da modifiche effettuate in una nuova versione della funzione.

Tuttavia, PerfSyn non affronta il problema di sintetizzare programmi che richiedono valori di un particolare dominio. Il risultato è che PerfSyn è efficace a sintetizzare programmi che usano strutture di dati generiche, ma è limitato a testare programmi che utilizzano librerie specifiche in un particolare dominio, come per esempio un compilatore.

Per minimizzare questa problematica introduciamo TestMiner. TestMiner utilizza tecniche di information retrieval su codice esistente per predire valori compatibili con i parametri di un metodo. TestMiner estrae constanti, come stringhe, dal codice di test unitari esistenti, ed indicizza questi valori in modo

che possano essere ricercati velocemente ed automaticamente. I valori indicizzati possono, per esempio, essere utilizzati da un generatore automatico di test unitari quando necessita i valori per un metodo testato. In questa dissertazione dimostriamo che TestMiner è capace di incrementare la quantità di codice eseguito da test unitari creati con un moderno generatore di test unitari come Randoop.

Infine, in questa dissertazione estendiamo MemoizeIt con funzionalità di supporto all'eliminazione dei problemi di performance identificati attraverso suggerimenti di configurazioni di cache pensate per ottimizzare una funzione attraverso memoization. Tali configurazioni vengono presente allo sviluppatore in forma testuale e vengono estratte dalle tracce di esecuzione del programma testato. La nostra ipotesi è che le configurazioni di cache suggerite da MemoizeIt sono effettive ed aiutano lo sviluppatore ad incrementare la velocità di esecuzione di un programma. I risultati presentati nella dissertazione dimostrano che applicando le configurazioni suggerite da MemoizeIt migliora in modo statisticamente significativo le prestazioni dei programmi utilizzati nei nostri esperimenti.

ACKNOWLEDGMENTS

This short space won't be long enough to thank the long list of people that helped me, supported me, and that had a lot patience with me during the past years.

First, I would like to thank my advisor Prof. Thomas Gross for giving me the possibility to purse a Ph.D. and for his guidance, his support, and to allow me to freely follow my research ideas.

I want to thank Matthias Hauswirth for the support and for the inspiring mentoring during my graduate studies.

I would like to express my gratitude to all the LSTers that I met and that helped me in this adventure: Faheem Ullah, Albert Noll, Martin Bättig, Stefan Schmid, Virag Varga, and Ivana Unkovic.

My deepest gratitudes go to the LLTers that over the past couple of years contributed with many fun moments during our double-dinners: Aristeidis Mastoras, Michael Faes, Remi Meier. Thank you guys!

I would like to thank all my ETH Zurich colleagues Zoltán Majó, Gagandeep Singh, Pavol Bielik, and Tyler Smith for the help in the past months and for the good moments over the years.

A special thank you goes to Michael Pradel for all the discussions, for the good and the interesting work done together during these years. It was a pleasure! Together with Michael I would like thank his fabulous Ph.D. students Cristian-Alexandru Staicu and Marija Selakovic for the all the long, productive, and interesting chats.

Without all the people in H65 that helped me in very difficult moments I would have never made it. Thanks to Daniele Spampinato for teaching me how to be a more moderate person, thanks to Georg Ofenbeck for the support and for all the time we spent together outside work. I have to thank Victoria Caparrós Cabezas for all the hate sharing and to support me in dark times. I would also express my gratitude to François Serre to be a living encyclopedia and to teach me a lot of fun unknowns. Thanks to Alen Stojanov for all the discussions and funny moments during lunch breaks. Thanks to Eliza Wszola for just being naturally nice and funny.

A special thought goes also to all the friends I met during the time at Mozilla, in particular to: Nicholas D. Matsakis, Shy-yu Guo, and Vincent St-Amour.

I owe Alessandro Puiatti and Paolo Bonato a lot because they made possible one of the greatest experiences of my life by shipping me up to Boston.

A lot of good memories that will never be forgotten have been created by all of the people met at the Motion Analysis Laboratory: Shyamal and Chiara, Peps, Yalgin, Giacomo, Patrick, Silvia, and all the others.

I will be always be grateful to the people from SUPSI that encouraged and supported me during my undergraduate and graduate years, thank you Silvia, Salvatore, Daniele, Michela and Alan.

I wanted to thank all my Ticino friends here in Zurich with whom we spent many distracting evenings: Paolino, Cris, Andi, Andreino, Elia and Gabri.

An immense gratitude goes to my two lebanese friends Badih Assaf and Samer Chucri. Thanks for all the constructive (and sometimes not very much) discussions over these years, and for the amazing time we spent together, you guys taught me a lot.

Thank you Alayna for have been on my side, and for helping me grow in the past years.

Thank you Cecy for the infinite support and the infinite patience you had with me. Thank you for bringing that vibrant positive energy sprinkled with a bit of craziness to all my days. I love you.

To finish, I would like to thank all my family for their unconditional support in all these years, without you I would have never went this far.

CONTENTS

1 INTRODUCTION

The performance of software is critical in many domains, e. g., to achieve the desired throughput of a server, to reduce the energy consumption on resource-constrained devices, to maximize the usage of computing resources for scientific calculus, or to shield users from waiting for an unresponsive application.

The research literature does not lack of previous work that addresses the problem of optimizing the performance of system components that range between the small scale (e. g., small kernel numerical functions [142]) to larger systems (e. g., distributed systems like search engines [78]).

In a large application the many abstraction layers of the system affect the end-to-end performance of the system in many interesting and unexpected ways. For example, not only the single component performance matters but also the interactions within these components affect the overall system performance [19, 35, 49, 78, 79, 95, 98, 128]. Despite the analysis of the interactions across system components is still subject of investigation, this dissertation focuses on analyzing a single component's performance. In this dissertation we consider a component a sub-set of classes that are part of a larger application. Their declared public methods define the mean of interaction with the component.

To determine if a component suffers from a performance problem, a component developer must perform two steps. The first step is to determine a performance specification for the application component (e. g., service-level agreements). For example, a developer may set a quality of service metric that requires a maximum response time for a component APIs method or may demand that a method of the APIs is of a certain computational complexity class. The second step is to verify that the component's measured performance fulfill the performance specification, i. e., the component does not suffers from a performance issue.

In practice finding an adequate performance specification is not easy:

- because it can be subjective to the user perception of an application performance, or it may not be measurable
- because of the lack of the APIs performance documentation, therefore possibly unknown for a client of the APIs or
- because a performance problem may not be known in advance.

In this dissertation we assume that a developer already determined a set of performance specifications for the application therefore we don't focus on finding good performance specification(s) [23, 30, 33, 69, 130]. Instead the focus of this dissertation is to advance the current state-of-the-art of performance analysis tools by introducing novel techniques to reduce the manual effort that a developer must invest to debug a component's performance. In this dissertation we introduce novel approaches to help a developer finding previously unknown performance problems, and a novel technique to help the developer validate his or her assumptions about the performance of a method.

One of the most widely used approach to debug performance is CPU time profiling. A CPU time profiler determines the distribution of time spent across functions calls during a program execution with one input to the program. There exist multiple implementations of CPU time profilers that work with different operating systems and for different programming languages. For example, a list of the most widely used CPU time profiles includes perf [9], OProfile [8], GProf [61], DTrace [28], JFluid [47]. All these tools share the same idea: to report to the developer where the execution time of a program is spent.

In this dissertation we focus on Java programs that execute on the Java Virtual Machine (JVM). We analyze Java programs because previous work that describes the state-of-the-art performance debugging tools mainly focuses on Java programs, and because Java is one of the most popular programming languages.

Despite the common use of profilers to debug performance, profilers suffer from two types of limitations. The first limitation of profilers is strictly technical. Profilers adopt sampling techniques to keep their overhead manageable and to reduce the program execution perturbations due to the observer effect. In other words, sampling is used to reduce the large overhead that results from recording the execution time for every function called by a program. Sampling approximates a function execution time by recording the program state at regular intervals (e. g., the call stack content at the sampling interval). The assumption behind this kind of a sampling techniques is that if a function takes a large portion of the execution, it will be seen in proportionally many samples. However due to limitations dictated by the implementation of a JVM, Java profilers often report misleading results [100].

The focus of this dissertation is not on the technical limitations of profilers. We assume that commonly used Java profilers are at least an actionable profiler, because a perfect profiler does not exists in practice. An actionable profiler is a profiler that if acted upon its reports, the outcome is what expected [100]. In other words if a developer optimizes a method that a profiler deems to be slow, then an optimized version of the method execution time will be reduced proportionally to what reported by the profiler.

The other limitations profilers suffer from are non-technical but rather conceptual. In this dissertation we focus on these other important limitations because they limit the developer understanding of a program's performance. We believe that the contributions of this dissertation can help practitioners in their daily tasks, and that the presented techniques could be used alongside profilers to help developers getting a better understanding of a program's performance.

It is common practice to debug a program that has a larger than expected execution time using the results provided by a profiler report. A Java profiler indicates to the developer the code locations of the program where the time is spent, or the code locations that allocate a large amount of memory, or the state of the program's threads during a program run [47]. In this dissertation we focus on performance problems that can be optimized by reducing the amount of work done by a single or multiple interacting methods, and that are caused by a larger than expected execution time under, a possibly not yet known, input in one single thread of execution. Since we analyze and optimize Java programs, a reduction of the execution time in an optimized program may be due to the effects caused by the different interactions between the runtime system and the optimized program (e. g., a reduced garbage collection pressure because we allocate fewer objects). The performance problems we tackle on this dissertation are referred in the literature to as *performance bug*, a type performance problem that can be fixed with small program changes [75]. In the dissertation we refer to a performance bug interchangeably as (performance) bottleneck, problem, or issue.

The major drawback of using a profiler to discover a performance issue is that a profiler won't indicate to the developer *why* a program is spending a large amount of time in a method. Therefore the first profiler limitation we address in this dissertation is:

Limitation 1

Profilers indicate where the time in a program is spent but not how time is wasted.

A profiler's report provides only limited information to the developer to enable him/her to understand the causes of a performance problem. A developer must inspect the profiler reports, identify the code locations that may contain the causes of the slow execution, and fix the performance problem. This task is manually performed, tedious, and error prone because the profiler may mislead the developer to attribute a performance problem to the wrong code location [16, 38, 81].

A profiler does not provide the additional information required to find the cause(s) of a performance problem. Therefore, there is a need for better tools to guide a developer into debugging a program's performance. A body of work presents program analyses that record runtime program information that is matched against a signature of a specific performance anti-pattern [75, 88, 93, 103, 110, 145, 147, 156]. These approaches execute the program with a fixed input and they dynamically analyze a program, and they report program locations where the performance anti-patterns manifests.

Similarly to existing approaches, we introduce MemoizeIt, a dynamic analysis that detects, as performance anti-pattern, redundant computations. Given a program's input, MemoizeIt dynamically analyses the program's execution and it suggests to the developer methods that may benefit from memoization. These methods may benefit from memoization because the method's executions produce the same output given the same input multiple times. To speedup a program's execution, a developer applies a cache to the method which maps previously computed output to method inputs.

To trigger a performance bottleneck, a developer must find a bottleneck exposing input for a program. Then, once a performance bottleneck location is found, a developer can proceed to create a fix for the problem. To find a suitable fix, a developer may perform multiple changes to the code until he/she finds a set of code changes that alleviates the bottleneck. To test the fix, the developer must execute the program with the bottleneck triggering input, and then exercise the fix with new inputs to test for unexpected performance regressions.

However manually finding these new inputs is costly because it requires the developer to search them from a possibly large set of inputs combinations. In addition while performing this task a developer may overlook inputs that trigger a performance regression. Unfortunately, also in this case, a profiler helps a developer only when the new testing inputs are available:

Limitation 2

Profilers work with a sample of the program's input. The understanding of the program's performance behaviour is limited to the provided available input.

To address this limitation we introduce PerfSyn, an approach that automatically synthesizes bottleneck-exposing programs for a given method under test.

The key ideas of PerfSyn are: (i) to repeatedly mutate a program that uses the method under test to systematically increase the amount of work done by the method, and (ii) to formulate the problem of synthesizing a bottleneck-exposing program as a combinatorial search. PerfSyn helps to reduce the manual effort required to write performance testing programs by automatically generating testing inputs.

Once a developer found a method that has sub-optimal performance, and he/she verified that indeed a method suffers from a performance bottleneck, the developer has to fix the performance issue. First the developer decides if the method is worth optimizing. For example, a developer may decide to not optimize a method because the required changes to the application may affect the application correctness. To decide if a set of program transformations that remove the bottleneck exists, a developer must at least perform the following tasks:

1. Determine which existing code is touched by a possible performance patch, and which of these parts may affect the program correctness.
2. Explore and consider several code changes because in general there exists more than one way to transform a piece of code.
3. Test the updated code with the bottleneck exposing input, and with other inputs, to verify that the new program changes do not introduce a slowdown compared to the original program.

However, profilers can only help the last debugging task, i. e., to verify that a program change effectively reduces the execution time of a method. Nevertheless, the other debugging activities are still performed manually [102, 137], and automatically patching code is a difficult task [13, 89, 99]. The last profiler limitation we tackle in this dissertation is:

Limitation 3

Profilers do not provide actionable suggestions on how and on where to fix a program if there is a performance issue.

To solve this limitation we extend MemoizeIt to guide the developer towards finding a suitable fix that removes a redundant computation from a method. MemoizeIt suggests to the developer actionable configuration fixes, but we leave to the developer the ultimate choice how to implement the code changes.

We believe that the developer must be involved in the performance debugging cycle. We argue that we can improve how developers debug and fix performance bottlenecks by introducing tools that provide actionable information, but that require minimal effort by developer to be used, and that scale for real-world software.

Given the profilers limitations and the premise to keep the developer in the performance fixing loop, the goal of this dissertation is:

Goal of this dissertation
The goal of this dissertation is to investigate novel techniques that tackle the discussed three major profilers limitations. The aim of this dissertation is to reduce the developer's manual effort by increasing the level of automation developers detect/debug/fix performance bottlenecks.

ORGANIZATION OF THE DISSERTATION. This dissertation is divided in three chapters that reflect our research contributions. The contributions of Chapters 2-4 are based on the research work published in following three peer-reviewed articles [137–139].

In Chapter 2 we describe MemoizeIt and we evaluate how MemoizeIt addresses the first and the last profilers limitations. To address the first profilers limitation we show an efficient approach that scales with large heap graphs to detect redundant computations. To address the last profilers limitation we investigate an automatic approach that analyzes an execution trace and which it generates suggestions for cache configurations to avoid redundant computations. The content of Chapter 2 is based on the work published in [137].

We proceed by introducing PerfSyn in Chapter 3, a program synthesis approach that constructs short programs as input to a method under test. The program synthesis is directed towards generating programs that trigger a performance bottleneck. The generated program helps the developer to debug a performance bottleneck (e. g., to find performance corner-cases) and to test a performance fix (e. g., to validate the developer's performance assumptions). The PerfSyn program synthesis approach is introduced in the work published in [138].

To address PerfSyn's main limitation, i. e., failing to provide domain specific values, we introduce TestMiner in Chapter 4. The key idea of TestMiner is to exploit the information available in existing test suites. TestMiner uses information retrieval techniques to predict input values suitable for a particular method. We evaluate TestMiner and we show that TestMiner is able to significantly increase code coverage for a state-of-the-art random test generator. A shorter description of TestMiner is presented the research work published in [139].

We conclude our dissertation by summarizing the lesson learned and by verifying our claims in Chapter 5.

2 PROGRAM ANALYSIS FOR MEMOIZATION OPPORTUNITIES

The first limitation of profilers reflects that a profiler reports the CPU usage distribution across program locations (i. e., "hot code") for a single program run. A developer cannot easily determine from a profiler report whether the identified "hot" code contains an optimization potential and how to use this potential. Nevertheless CPU time profiling is still among the most widely used approaches to identify and debug performance bugs. Even worse, profilers may even mislead developers to refactor code with little or no performance improvements [100].

In this chapter we tackle the first profilers limitation by introducing MemoizeIt, an approach that suggests to the developer memoization opportunities at the method level. Memoization is an easy way to optimize a method by storing results from earlier calls of the method to reuse when the same input reappears. Memoization is a successful optimization that is applied in many contexts. For example, symbolic execution engines use memoization to avoid repeatedly traverse paths [73, 141, 151], or web services make extensive use of caches to avoid to recompute expensive queries results [32, 95]. Furthermore a recent study suggests that in practice code suffers from wasted work, and that developers fix slow code by applying memoization to code locations that perform a redundant operation multiple times [120].

However fixing a program location that suffers from excessive redundant computations is non-trivial. First because the inputs to a code location may vary over time, finding an optimal fix may require to try a large number of possible configurations. Second, because each fix is subject to performance variations due to hardware and software configurations, complicating the task of finding a suitable performance fix even further. To address the problem of finding an effective fix for a performance problem, i. e. the second profiler limitation, in Section 2.2.4 we present an enhanced version of MemoizeIt that reports hints on how to implement memoization for a method in an efficient way. The hints-enhanced version of MemoizeIt provides actionable suggestions on how to fix a program that suffers from a bottleneck. The hints help the developer to implement a cache, but ultimately, it is the developer's responsibility to

```
class DateUtil {
  private static final Pattern date_ptrn = Pattern.compile(..);
  private static String lastFormat;
  private static boolean cachedResult;
  static boolean isADateFormat(String format) {
    if (format.equals(lastFormat)
      return cachedResult;
    String f = format;
    StringBuilder sb = new StringBuilder(f.length());
    for (int i = 0; i < f.length(); i++) {
      // [..] copy parts of f to sb
    }
    f = sb.toString();
    // [..] process f using date patterns
    cachedResult = date_ptrn.matcher(f).matches();
    return cachedResult;
  }
}
```

Listing 2.1: Memoization opportunity in Apache POI that can be used by adding the highlighted code.

implement a correct and efficient cache. MemoizeIt strikes a balance between compiler optimizations, which automatically transform the code but must guarantee that the transformation preserves the semantics of the program, and existing profilers, which focus on hot but not necessarily optimizable code. Recent research approaches help developers understand the cause of a performance problem but not necessarily how to address it [75, 88, 93, 103, 110, 145, 147, 156]. In other words, the performance problem may be relevant but not actionable. Other work proposes automatic fixing of loop-related performance problems by performing automatic source-code transformations. However these fixes are very limited and follow simple pre-defined patterns [102].

Overall, the challenge of pointing developers to actionable performance fixes still waits to be fully addressed.

To illustrate the challenges of finding easy to address performance bottlenecks, consider Listing 2.1, illustrates a performance bug in the document converter tool suite Apache POI [2]. The method in the example analyzes a given string and returns a boolean that indicates whether the string matches one of several possible date formats. The method is called frequently in a typical workload, possibly repeating the same computation many times. The method can be optimized using memoization. The highlighted code in the figure shows a simple implementation of memoization, which leads to speedups ranging from 7% up to 25% in different workloads we experimented with.

Unfortunately, a developer may miss this opportunity with CPU time profiling because the method is one of many hot methods. Likewise, the opportunity is missed by existing compilers [46, 149] because the method has side effects.

Existing work on finding repeated computations [101] also misses this opportunity because that work focuses on call sites that always produce the same output, not on methods where multiple inputs yield the same outputs repeatedly.

The key idea of MemoizeIt is to systematically compare the inputs and the outputs of different invocations of the same method with each other. If a method repeatedly obtains the same inputs and produces the same outputs, it is reported as a memoization candidate. For each memoization candidate, MemoizeIt reports hints on how to implement memoization for the method. MemoizeIt is the first profiling approach that focuses on opportunities for method-level caching:

- The main contribution of this chapter is to present the first profiler that focuses on memoization opportunities at the method level. The profiler allows developers to find easy-to-implement optimizations that are difficult to detect with existing profiling approaches because these approaches are too conservative [46, 149], or because they focus on call-sites [101]. MemoizeIt takes a novel approach towards memoization by handling side effects and focusing only on equivalent, therefore cacheable, method input-output pairs.
- The second contribution of this chapter is an iterative approach that increases the degree of detail of a dynamic analysis to make MemoizeIt applicable to large programs. The iterative approach shrinks the set of program locations to analyze at each iteration, refining and reducing the set of memoization candidates over time. We show in our evaluation that the approach is effective and beneficial in practice.
- The third contribution of this chapter is a technique that gives hints on implementing memoization. The hints are based on the profiled data recorded during program execution, and they are in the form of cache configurations to apply to a method. A developer immediately apply the hints and he/she can tailor the optimization based on multiple input profiles.
- The last contribution is an empirical evaluation of an implementation of MemoizeIt. The evaluation shows that MemoizeIt is able to efficiently and effectively find previously unknown memoization opportunities that result in statistically significant speedups in popular and widely used Java programs.

We begin this chapter by motivating our work with an illustrative example in Section 2.1. In Section 2.2 we describe the approach in detail. In Section 2.3 we describe how we implemented a prototype for MemoizeIt and we evaluate our implementation in Section 2.4. We list the limitations of the approach in Section 2.6 and we summarize our findings in Section 2.7.

```
class Main {
  static void main() {
    Main main = new Main();
    Logger logger = new Logger();
    Result res1 = main.compute(new Input(23));
    boolean ok1 = logger.append(res1);
    Result res2 = main.compute(new Input(23));
    boolean ok2 = logger.append(res2);
    if (ok1 && ok2) System.out.println("done");
  }
  Result compute(Input inp) {
    Result r = new Result();
    // complex computation based on inp
    r.p.fst = ..
    r.p.snd = ..
    return r;
  }
}
class Input {
  int n;
  Input(int n) { this.n = n; }
}
class Result {
  Pair p = new Pair();
}
class Pair {
  int fst;
  int snd
}
class Logger {
  Writer wr = ..;
  int ctr = 0;
  boolean append(res) {
    wr.write(ctr+": "+res);
    ctr++;
    return true;
  }
}
```

Listing 2.2: Program with a memoization opportunity.

First iteration (depth 1):

Call	Input (target, arguments)	Output (return value)
compute (line 5)	$\boxed{\text{Main}}$, $\boxed{\text{Input: n=23}}$ $\boxed{\text{Logger: ctr=0}} \xrightarrow{wr}$ some Writer ,	$\boxed{\text{Result}} \xrightarrow{p}$ some Pair
append (line 6)	$\boxed{\text{Result}} \xrightarrow{p}$ some Pair	true
compute (line 7)	$\boxed{\text{Main}}$, $\boxed{\text{Input: n=23}}$ $\boxed{\text{Logger: ctr=1}} \xrightarrow{wr}$ some Writer ,	$\boxed{\text{Result}} \xrightarrow{p}$ some Pair
append (line 8)	$\boxed{\text{Result}} \xrightarrow{p}$ some Pair	true

Second iteration (depth 2):

compute (line 5)	$\boxed{\text{Main}}$, $\boxed{\text{Input: n=23}}$	$\boxed{\text{Result}} \xrightarrow{p} \boxed{\text{Pair: fst=42, snd=23}}$
compute (line 7)	$\boxed{\text{Main}}$, $\boxed{\text{Input: n=23}}$	$\boxed{\text{Result}} \xrightarrow{p} \boxed{\text{Pair: fst=42, snd=23}}$

(a) Executing MemoizeIt's input-output profiling for the program in Listing 2.2 reveal a memoization oppportunity after two iterations.

Potential performance bug in method `Main.compute(Input)`:

- Same input-output pair occurs twice.
- Suggestion: Add a global single-element cache

```java
private static int key = INVALID_CACHE; // Global cache key
private static Result cache = null;     // Global cache value
Result compute(Input inp) {
  if (key != INVALID_CACHE && key == inp.n) {
    return cache;
  } else {
    Result r = new Result();
    // complex computation based on inp
    r.p.fst = ..
    r.p.snd = ..
    key = inp.n;
    cache = r;
    return cache;
  }
}
```

(b) Report produced by MemoizeIt together with the method `Main.compute(Input)` with a cache that implements the suggested fix.

Figure 2.1: The running example shows how MemoizeIt's iterative profiling traverses the input and the output objects of a method call (top figure) and an example of the report that MemoizeIt produces for a developer to inspect (bottom figure).

2.1 MEMOIZEIT BY ILLUSTRATION

Listing 2.2 shows a program that repeatedly calls two methods, `compute` and `append`. One of them, `compute`, redundantly performs a complex computation that can be avoided through memoization. The method computes a result that depends only on the given argument, and calling the method multiple times with equivalent arguments yields equivalent results. In contrast, the other method, `append`, cannot be memoized because it logs messages into a writer and because it increments a counter at each call. A static or dynamic analysis that conservatively searches for memoization opportunities [46, 149] misses the opportunity in `compute` for two reasons. First, the method has side effects because it creates a new object that escapes from `compute`. In general, memoizing a method with side effects may change the semantics of the program. Second, the method's return value has a different object identity at every call. In general, memoizing such a method may change the semantics because the program may depend on object identities. In the given program, however, the side effects of `compute` are redundant because the created `Result` objects are structurally equivalent and the program is oblivious of object identities. Therefore, memoizing the results of `compute` improves performance while preserving the program's semantics.

A key insight of MemoizeIt is that a method may benefit from memoization even though it has side effects and even though the object identities of its inputs and outputs vary across calls. Instead of conservatively searching for memoization opportunities that can certainly be applied without affecting the semantics, MemoizeIt searches for memoization opportunities

Definition 1 (Memoization opportunity)**.** *We consider a method m as a potential memoization opportunity if all of the following memoization conditions hold:*

- *(MC1) The program spends a non-negligible amount of time in m.*
- *(MC2) The program repeatedly passes structurally equivalent inputs to m, and m repeatedly produces structurally equivalent outputs for these inputs. We formally define "structurally equivalent" in Section 2.2.2.*
- *(MC3) The hit ratio of the cache, i. e., the number of times that a result can be reused over the total number of lookups, will be at least a user-defined minimum. This condition ensures that adding a cache will lead to savings in time because the time saved by reusing already computed results outweighs the time spent for maintaining the cache.*

MemoizeIt detects memoization opportunities with a dynamic analysis that checks, for each method that is called in the analyzed program execution, whether conditions MC1 to MC3 hold. To check for MC1, MemoizeIt uses a state of the art CPU time profiler to identify methods where a non-negligible amount of time is spent. To check for MC2, MemoizeIt records the inputs and outputs of methods to identify methods that repeatedly take the same input and produce the same output. To check for MC3, MemoizeIt estimates the hit ratio that a cache would have if the developer added memoization to a method. This estimate is based on the inputs and outputs observed in the execution.

To implement this idea, we must address two challenges. First, we must define which inputs and outputs of a method call to consider. Second, we must address the problem that the inputs and outputs of a call may involve complex heap structures that are too large to record in full detail for each call.

2.1.1 *Challenge 1: Inputs and Outputs*

A conservative approach to detect memoization opportunities must consider all values that a method execution depends on as the call's input, and all side effects and the return value of the call as the call's output. The approach that MemoizeIt takes is to deliberately deviate from this conservative approach to detect memoization opportunities in methods that have redundant side effects. As *input* to a call, MemoizeIt considers the arguments given to the method, as well as those parts of the call's target object that influence the method's execution. As *output* of a call, MemoizeIt considers the call's return value. These definitions of input and output may ignore some state that a method depends on and ignore some side effects that a method may have. E. g., in addition to method arguments and the target object, a method may depend on state reachable via static fields and on the state of the file system. Furthermore, a method may modify state reachable from the passed arguments and the state of the file system. Our hypothesis is that focusing on the inputs and outputs given above summarizes the behavior of many real-world methods well enough to decide whether these methods may benefit from memoization. Our experimental results validate this hypothesis. The cost for considering methods despite side effects is that MemoizeIt may report methods that cannot easily be memoized because it would change the program's semantics. Our experiments show that this problem is manageable in practice and that MemoizeIt reports valid memoization opportunities missed by all existing approaches we are aware of.

2.1.2 *Challenge 2: Complex Heap Structures*

In object-oriented programs, the inputs and outputs of method calls often involve complex objects. Recording these objects, including all other objects reachable from them, for each call does not scale well to large programs. Consider the inputs and outputs of method `append` in Listing 2.2. Since we consider the state of the call target as part of the input, a naive implementation of our approach would have to record the state of `logger` at lines 6 and 8. This state includes the `Writer` object that `logger` refers to, which in turn refers to various other objects. In general, fully recording the state reachable from an object may involve arbitrarily many other objects, in the worst case, the entire heap of the program.

To address the challenge of recording large input and output objects, MemoizeIt uses an iterative analysis approach that gradually refines the set of methods that are considered as memoization candidates. The key idea is to repeatedly execute the program, starting with an analysis that records objects without following their references, and to iteratively increase the level of detail of the recorded inputs and outputs while pruning the methods to consider. After each iteration, MemoizeIt identifies methods that certainly fulfill the memoization conditions MC2 and MC3, and methods that certainly miss one of these two conditions. Methods that miss a condition are pruned and not considered in subsequent iterations.

Figure 2.1a illustrates the iterative profiling approach for our running example. We illustrate the recorded objects as heap graphs, where a node represents an object with its primitive fields, and where an edge represents a reference. In the first iteration, MemoizeIt records inputs and outputs at depth 1, i. e., without following any references. For example, the output of the first call to `compute` is recorded as a `Result` object that points to a not further analyzed `Pair` object. The information recorded at depth 1 allows MemoizeIt to decide that `append` cannot fulfill MC2 because there is no recurring input-output pair. The reason is that the value of `ctr` is 0 at line 6 but 1 at line 8. Note that MemoizeIt prunes method `append` without following the reference to the `Writer` object, i. e., without unnecessarily exploring complex heap structures.

After the first iteration, MemoizeIt keeps `compute` in the set of methods that may benefit from memoization and re-executes the program for the second iteration. Now, MemoizeIt records inputs and outputs at depth 2, i. e., it follows references from input and output objects but not references from these references. The lower part of Figure 2.1a shows the information recorded at depth 2. MemoizeIt now completely records all inputs and outputs of `compute` and determines that both calls to `compute` have structurally equivalent inputs and outputs, i. e., the method fulfills MC2. Furthermore, the method fulfills

MC3 because memoizing the results of `compute` would lead to a cache miss at the first call and a cache hit at the second call, i. e., a hit ratio of 50%.

Since MemoizeIt has fully explored all methods after two iterations, it stops the analysis and reports `compute` as a potential memoization opportunity. To help developers use this opportunity, MemoizeIt suggests how to implement memoization for every reported method. Based on the analyzed execution, the approach suggests to add a global single-element cache, i. e., to store the last observed input and output in static fields of `Main`, and to reuse the already computed output if the input matches the most recently observed input.

Figure 2.1b shows an implementation of MemoizeIt's suggestion. The optimized code has two additional static fields `key` and `cache` that store the most recently computed result and the corresponding input value `inp.n`. To avoid redundantly recomputing the result, the optimized method reuses the result whenever the same input value appears again. As a result, the program in Listing 2.2 performs the complex computation in `compute` only once.

2.2 APPROACH

The input to MemoizeIt is an executable program. MemoizeIt executes the program multiple times while applying dynamic analyses and reports performance bugs that can be fixed through memoization. MemoizeIt consists of four parts:

1. *Time and frequency profiling.* This part executes the program once and uses traditional CPU time profiling to identify an initial set of methods that may benefit from optimization (Section 2.2.1).
2. *Input-output profiling.* The main part of the analysis. It repeatedly executes the program to identify memoizable methods by analyzing the inputs and outputs of method calls (Section 2.2.2).
3. *Clustering and ranking.* This part summarizes the analysis results and reports a ranked list of potential memoization opportunities to the developer (Section 2.2.3).
4. *Suggest cache implementation.* This part suggests for each memoization opportunity how to implement a cache for the respective method (Section 2.2.4).

In the remainder of this section we describe each part in detail.

2.2.1 *Time and Frequency Profiling*

The first part of MemoizeIt uses state of the art CPU time profiling to identify the set of *initial memoization candidates*. We record, for each executed method m, the time t_m spent in m (including time spent in callees) and the number c_m of calls of m. Furthermore, we also measure the total execution time t_{prgm} of the program.

Definition 2 (Memoization candidate)**.** *As the initial set of memoization candidates, MemoizeIt considers all methods that fulfill three requirements:*

1. *The average execution time of the method is above a configurable minimum average execution time:* $\frac{t_m}{c_m} > t_{min}$.
2. *The relative time spent in the method is above a configurable threshold:* $\frac{t_m}{t_{prgm}} > r_{min}$.
3. *The method must be called at least twice,* $c_m \geqslant 2$.

The first two requirements focus MemoizeIt on methods that are worth optimizing (MC1). The third requirement is a necessary condition for MC2 because a method can repeat a computation only if the method is called multiple times.

2.2.2 *Input-output Profiling*

The core of MemoizeIt is input-output profiling, which computes a set of memoization candidates by comparing method calls with each other to check whether MC2 and MC3 hold.

Representing Input-output Data

To detect method calls that perform computations redundantly, MemoizeIt records the input and output of each call. For full precision, we could consider the following input and output data:

- Input state (before the call):
  - The state of the target object of the call. (*)
  - The state of each argument passed to the call. (*)
  - All heap state that is reachable via static fields.
  - The environment state outside of the program's heap, e. g., file system and network state.
- Output state (after the call):
  - The four kinds of state listed above.
  - The state of the return value of the call (if any). (*)

Recording all these inputs and outputs of each method call is clearly infeasible, partly because such an approach would not scale to large programs and partly because acquiring the complete state is practically impossible. Instead, MemoizeIt focuses on those parts of the input and output state that are marked with (*). The rationale for focusing on these parts of the input and output state is twofold. First, we made the observation that these parts of the state are sufficient to describe the relevant input and output state for many real-world methods. E. g., most methods read the arguments given to them, but only few methods depend on modifiable heap state reachable through static references. Second, recording some of the state that is ignored by MemoizeIt would lead to redundant information that does not improve the precision of the approach. E. g., recording the state of the target object after each call would often replicate the state recorded for the same target object before the next call. If a method is not memoizable because it depends on the state of the target object, then recording the target object's state once as part of the input is sufficient to identify this method as non-memoizable.

To compare input and output data of method calls, the analysis flattens data items into a generic canonical representation. The canonical representation describes the data itself and, in case of complex object structures, the shape of the data item. The representation describes objects structurally and is independent of the memory locations where objects are stored or other globally unique identifiers of objects. A data item d is flattened as follows:

- If d is a primitive value, it is represented by its string representation.
- If d is an object, it is represented as a pair (R_n, F), where
  - R_n identifies the object d using its runtime type R and an identifier n that is unique within the flattened data representation,
  - F is a list of flattened data representations; each element of F represents the value of one of d's fields.
- If d is the `null` value, it is represented by $NULL$.
- If d is an object that is already flattened in this representation and that has the identifier R_n, it is represented by $@R_n$. We use this notation to deal with object structures that have cyclic references.

To ensure that structurally equivalent objects have the same flattened representation, each identifier is unique within its flattened representation but not globally unique. Furthermore, the list F contains fields in a canonical order based on alphabetic sorting by field name.

For example, the return values of the two calls of `compute` in Listing 2.2 are both represented as:

$$(Result_1, [(Pair_1, [42, 23])])$$

To illustrate how the representation deals with cyclic data structures, consider a double-linked list built from instances of `Item` with fields `next`, `previous`, and `value`. For a two-element list with values `111` and `222`, the flattened representation is:

$$(Item_1, [(Item_2, [null, @Item_1, 222]), null, 111])$$

Comparing Input-output Data

The goal of input-output profiling is to evaluate MC2 and MC3 by identifying memoizable methods where multiple calls use the same input and produce the same output. To achieve this goal, the analysis summarizes each call into a tuple that represents the inputs and the output of the call.

Definition 3 (Input-output tuple). *The input-output tuple of a call is $T = (d_{tar}, d_{p1}, \ldots, d_{pn}, d_{ret})$, where*

- *d_{tar} is the flattened representation of the target of the call,*
- *$d_{p1}, \ldots, d_{pn}$ are the flattened representations of the parameters of the call, and*
- *d_{ret} is the flattened representation of the call's return value.*

For each method m, the profiler builds a multiset $\mathcal{T}_m$ of input-output tuples observed for m, called the *tuple summary*. The tuple summary maps each observed tuple to the number of times the tuple has been observed during the program execution. When the profiler observes a call of m, the profiler creates an input-output tuple T for the executed call and adds it to the tuple summary $\mathcal{T}_m$.

Definition 4 (Tuple multiplicity). *The multiplicity $mult(T)$ of a tuple T in the tuple summary $\mathcal{T}_m$ is the number of occurrences of T in $\mathcal{T}_m$.*

E. g., the call in line 5 of Listing 2.2 gives the following input-output tuple:

$$T = ((Main_1, []), (Input_1, [23]), (Result_1, [(Pair_1, [42, 23])]))$$

This tuple T has $mult(T) = 2$ because the two calls at lines 5 and 7 have the same input-output tuple.

Based on the tuple summary for a method, MemoizeIt computes the potential hit ratio that a cache for the method may have:

Definition 5 (Potential hit ratio). *For a method m with tuple summary $\mathcal{T}_m$, the potential cache hit ratio is:*

$$hit_m = \frac{\sum\limits_{T \in \mathcal{T}_m} (mult(T) - 1)}{\sum\limits_{T \in \mathcal{T}_m} mult(T)}$$

The potential hit ratio indicates how often a method execution could be avoided by reusing an already computed result. The hit ratio estimates how profitable memoization would be, based on the assumption that the return value for a particular input is stored in a cache when the input occurs the first time and that the stored value is reused whenever the same input occurs again. The ratio is an estimate, e.g., because it ignores that a cache may have to be invalidated, a step that may reduce the number of hits.

Finally, MemoizeIt identifies methods that potentially benefit from caching:

Definition 6 (Memoization candidate). *Method m is a memoization candidate if $hit_m \geqslant h_{min}$, where h_{min} is a configurable threshold.*

By this definition, memoization candidates fulfill both MC2 and MC3. In particular, a method that is called only once is not a memoization candidate, because the potential hit ratio of such a method is zero.

For the example in Figure 2.1a, the potential hit ratio of `compute` is 50% because the call in line 5 must compute the return value, but the call in line 7 can reuse the cached value.

Iterative Refinement of Memoization Candidates

The approach described so far is effective but prohibitive for programs with complex heap structures. The reason is that recording input-output tuples requires the analysis to traverse all objects reachable from the target, the arguments, and the return value of a call. In the worst case, the analysis may traverse the entire heap multiple times for each call. To overcome this scalability problem, our analysis executes the program multiple times, while iteratively increasing the *exploration depth* up to which the analysis explores the object graph, and while shrinking the set of memoization candidates. After each iteration, the analysis discards two sets of methods: (i) methods that certainly do not satisfy Definition 6 and (ii) methods for which the input-output tuples include all objects and all their transitive references. We say that a method in the second set has been *fully explored*.

To support iterative refinement of memoization candidates, we refine the canonical flattened data representation by including a bound k for the exploration depth. Specifically the k-bounded flattened data representation of a data item d is the flattened representation of d, where only objects reachable from d via at most k references are included.

Similar to the above, we also adapt Definitions 3, 4, 5, and 6 to consider the exploration depth k. The k-bounded input-output tuple of a call is a tuple T_k where each element of the tuple is now k-bounded. For a method m with a k-bounded tuple summary $\mathcal{T}_{m,k}$, the k-bounded potential hit ratio becomes $hit_{m,k}$ and, a method m is a memoization candidate at depth k if $hit_{m,k} \geqslant h_{min}$.

For example, the previously illustrated call at line 5 of Listing 2.2 gives the following k-bounded input-output tuple for $k = 1$:

$$T = ((Main_1, []), (Input_1, [23]), (Result_1, [(Pair_1, [])]))$$

That is, in the k-bounded flattened data representation an object that is not expanded is represented as the pair (R_n, []), where R is the runtime type and where n is the unique identifier.

Based on these adapted definitions, Algorithm 1 summarizes the iterative algorithm that refines the set of memoization candidates. The algorithm starts with an initial set $\mathcal{C}_{init}$ of memoization candidates provided by time and frequency profiling. The outer loop of the algorithm iteratively increases the depth k and performs input-output profiling at each depth, which returns a list of k-bounded tuple summaries. Based on the tuple summaries, lines 5 to 12 compute the set $\mathcal{C}_{next}$ of methods to consider in the next iteration as the set of methods with a sufficiently high hit ratio.

Algorithm 1 Iterative refinement of memoization candidates.

```
Input: Initial method candidate set $\mathcal{C}_{init}$, profiling $timeout$ in seconds
Output: Candidates set $\mathcal{C}$, $\mathcal{T}_m$ for each method $m \in C$
 1: $k = 1$
 2: $\mathcal{C} = \mathcal{C}_{init}$
 3: while (stopping_condition($timeout$) != true) do
 4:     $\mathcal{T}_{m_1,k}, \ldots, \mathcal{T}_{m_j,k} = IOprofile(\mathcal{C}, k)$
 5:     $\mathcal{C}_{next} = \varnothing$
 6:     for method $m \in \mathcal{C}$ do
 7:         $hit_{m,k} = computeHitRatio(\mathcal{T}_{m,k})$
 8:         if $hit_{m,k} \geqslant h_{min}$ then
 9:             $\mathcal{C}_{next} = m \cup \mathcal{C}_{next}$
10:         end if
11:     end for
12:     $\mathcal{C} = \mathcal{C}_{next}$
13:     $k = nextDepth(k)$
14: end while
```

The algorithm iterates until one of the following stopping conditions holds:

- all remaining memoization candidates in $\mathcal{C}$ have been fully explored;
- there are no more memoization candidates that satisfy MC3.
- a user-provided timeout is reached;

When reaching the stopping condition, the algorithm returns the set $\mathcal{C}$ of memoization candidates, along with the tuple summary $\mathcal{T}_m$.

For illustration, recall the example in Figure 2.1a. Initially, both methods `compute` and `append` are in the set $\mathcal{C}$ of candidates. After the first iteration, the check at line 8 finds that the hit ratio of `append` is zero, i.e., below the minimum hit ratio, where as the hit ratio of `compute` is 50%, i.e., above the threshold. Therefore, only `compute` remains as a candidate for the second iteration. After the second iteration, `compute` is still a memoization candidate and the algorithm stops because all calls have been fully explored.

PROPERTIES OF ITERATIVE REFINEMENT. It is important to note that Algorithm 1 does not miss any memoization opportunities found by an exhaustive approach that analyzes all calls with unbounded tuples. Specifically, the iterative refinement of caching candidates provides two guarantees:

- If the analysis discards a method as non-memoizable at depth k, it would never find that the method requires memoization at a depth $> k$. That is, discarding methods is sound. The reason is that once two tuples are found to be different, this fact cannot be changed by a more detailed analysis (that is, a larger k), i.e., $\mathcal{C}$ is guaranteed to be a superset of $\mathcal{C}_{next}$.
- When a method is fully explored, the iterative approach yields the same set of methods as with unbounded exploration. This property is an immediate consequence of the first property.

As a result, iteratively refining memoization candidates reduces the complexity of input-output profiling without causing the analysis to miss a potential memoization opportunity.

To increase the depth k, Algorithm 1 uses a function *nextDepth*. This function must balance the cost of repeatedly executing the program against the ability to remove methods from the candidate set. Smaller increments result in more program executions, but they also allow the algorithm to discard methods at a lower k. E.g., suppose a method can be pruned from the candidate set at depth 2. If *nextDepth* increases k from 1 to 10, the algorithm will unnecessarily explore the method's inputs and outputs at depth 10. In contrast, incrementing the depth by one allows the algorithm to prune the method after exploring it at depth 2. In our experiments we find that doubling the depth at each iteration, i. e., $k = 1, 2, 4, ..$, provides a reasonable tradeoff. To further reduce the runtime of Algorithm 1, future work may adapt *nextDepth* based on knowledge from previous iterations. For example, the analysis could use larger increases of k if it discovers that many methods have deep reference structures.

Field Access Profiling

The following describes a refinement of input-output profiling that allows MemoizeIt to discover additional memoization opportunities and that improves the efficiency of MemoizeIt by reducing the size of flattened representations. The approach described so far considers all objects reachable from the input and output objects as part of the input and output, respectively. However, a method may read and write only parts of these objects. E. g., suppose that in Listing 2.2, class `Main` has a field `f` that is not accessed by `compute`. Recording the value of `f` as part of the target object of `compute` includes unnecessary data because the memoizability of `compute` is independent of `f`. Even worse, suppose that `f`'s value differs between the two redundant calls in lines 5 and line 7.

In this case, MemoizeIt would not report `compute` as a potential memoization opportunity because there would not be any repeated input-output tuple.

To avoid unnecessarily including fields of target objects in input-output tuples, we refine input-output profiling by considering only those fields of the target object as input to a method m that are used in some execution of m. To compute this set of *input fields* for a method m, MemoizeIt executes the program once before input-output profiling to track all field reads and writes. A field f is an input field of m if there exists at least one execution of m, including the callees of m, that reads f before writing to it. Other fields, e. g., a field that is never accessed in m or a field that is written by m before being read by m, are not part of m's input.

To further reduce the overhead and improve the precision of the analysis, a similar refinement can be applied for other inputs and outputs, namely, method parameters and return object. In practice, we have not observed many cases where these additional refinements would improve precision.

2.2.3 *Clustering and Ranking*

MemoizeIt constructs reports about potential memoization opportunities by clustering methods that should be inspected together and by ranking methods. To cluster methods that should be inspected together, MemoizeIt creates a static call graph and assigns two methods m_1 and m_2 to the same cluster if:

- m_1 is a direct caller of m_2, or
- m_1 is an indirect caller of m_2 and both methods are defined in the same class.

The clustering approach is based on the observation that a possibly memoizable method often calls other possibly memoizable methods. In this case, a developer would waste time by inspecting both methods separately. Instead, MemoizeIt presents both methods together.

The analysis ranks method clusters based on an estimate of their potentially saved time $saved_m = t_m * hit_m$ where t_m is the total time spent in the method during the initial time profiling run. For each cluster of methods, the analysis selects the method with the highest potentially saved time $saved_m$ and sorts all clusters by the potentially saved time of this method.

Table 2.1: Cache implementations suggested by MemoizeIt.

Size	Scope	Description
Single	Global	Stores the most recently seen input and output of all calls of the method, e.g., in static fields, and reuses the output when the same input appears in consecutive calls.
Single	Instance	For each target object, stores the most recently seen input and output of the method, e.g., in instance fields, and reuses the output when the same input appears in consecutive calls.
Multi	Global	Maps all inputs to the method to the computed output, e.g., in a static map, and reuses outputs whenever a previously seen input occurs.
Multi	Instance	For each target object, maps inputs to the method to the computed output, e.g., in an map stored in an instance field, and reuses outputs whenever an input has already been passed to this instance.

2.2.4 *Suggesting a Cache Implementation*

To use a memoization opportunity identified by MemoizeIt, a developer must implement a cache that stores previously computed results of a method for later reuse. Choosing an appropriate implementation is non-trivial and an inefficiently implemented cache may even reduce the performance of a program. In particular, a developer must make the following decisions. First, how many input-output pairs should the cache store? Common strategies include a single-element cache that remembers the last input-output pair and an associative map of bounded size that maps previously observed inputs to their computation output. Second, what should be the scope of the cache? Common strategies include a global cache that stores input-output pairs for all instances of a class and an instance-level cache that stores input-output pairs for each instance of the class.

To help developers decide on an appropriate cache implementation, MemoizeIt suggests a cache implementation based on the observed execution. To this end, the approach considers four common kinds of caches (Table 2.1). For each possible cache implementation, the approach simulates the effects of the cache on the analyzed execution based on the recorded input/output tuples, and it computes the following data:

- *Hit ratio.* How often can the method reuse a previously computed result? Depending on the cache implementation, the hit ratio may differ from Definition 5, which assumes that a global, multi-element cache is used.
- *Invalidation.* Does the program have to invalidate the cache because returning a cached value would diverge from the method's actual behavior? The cache simulator determines that the cache needs to be invalidated if there is a cache hit (i.e., a call's input matches a stored input) but the cached output does not match the recorded output of the call.
- *Size.* For multi-element caches, how many input-output pairs does the cache store, assuming that it never evicts cache entries?

Based on these data, MemoizeIt suggests a cache implementation with the following algorithm. First, the approach removes all cache implementations that lead to a hit ratio below a configurable threshold (default: 50%). Second, the approach picks from the remaining cache implementations the top-most implementation as listed in Table 2.1. The table sorts cache implementations by how simple they are to implement and by the computational effort of inserting and looking up input-output pairs. As a result, MemoizeIt suggests the simplest and most efficient cache implementation that yields a large enough hit ratio.

2.3 IMPLEMENTATION

We implement MemoizeIt into a tool for Java programs. The implementation combines online and offline analysis and builds upon several existing tools. Time and frequency profiling (Section 2.2.1) builds on JFluid [47] included in NetBeans 7.3 [7]. Input-output profiling (Section 2.2.2) uses ASM-based instrumentation [22] to inject bytecode that traverses fields. For some container classes, the flattened data representation contains internal details that are not necessary to determine whether two objects are conceptually equivalent. To deal with such classes, our implementation provides type-specific representations for arrays and for all classes implementing `java.util.Collection` or `java.util.Map`. Non-map collections and arrays are represented as a list of elements. Maps are represented as a list of key-value pairs. MemoizeIt summarizes flattened data representations into hash values using the `MurmurHash3` hash function [4], writes input-output tuples to a trace file, and analyses the file offline as described in Algorithm 1 and Section 2.2.4. For clustering optimization opportunities (Section 2.2.3), MemoizeIt uses Soot [140] to obtain call graphs. Our current implementation assumes that heap objects are not shared between multiple concurrent threads.

2.4 EVALUATION

We evaluate the effectiveness of the MemoizeIt performance bug detection approach by applying it to widely used Java programs. The analysis discovers nine memoization opportunities that give rise to speedups between 1.04x and 1.27x with the profiling input, and up to 12.93x with other inputs. Four of these nine memoization opportunities have already been confirmed by the developers in reaction to our reports. In the ranked list of methods reported by MemoizeIt, the memoization opportunities are first, second, or third for the respective program. In contrast, traditional CPU time profiling often hides these opportunities behind dozens of other methods and fails to bring them to the developer's attention.

2.4.1 *Experimental Setup*

Table 2.2 lists the programs and inputs used in the evaluation. We use all single-threaded programs from the DaCapo 2006-10-MR2 [18] benchmark suite (antlr, bloat, chart, fop, luindex, and pmd). We exclude jython because its use of custom class loading breaks our instrumentation system. The remaining benchmarks are excluded because they are multi-threaded. In addition we analyze Apache POI (a library for manipulating MS Office documents, 4,538 classes), the content analysis toolkit Apache Tika (13,875 classes), the static code checker Checkstyle (1,261 classes), and the Java optimization framework Soot (5,206 classes).

We apply MemoizeIt to each program with a profiling input. Since the speedup obtained by a cache depends on the input, we also experiment with other inputs to explore the potential benefit of adding a cache. For DaCapo, we use the "default" inputs for profiling and the "large" inputs (if available) as additional inputs. For the other programs, we use typical inputs, such as a spreadsheet with student grades for the spreadsheet conversion tool Apache POI, or the Java source code of CheckStyle for the static analysis CheckStyle. Columns 3 and 4 of Table 2.2 summarize the inputs.

For each profiling input, we run MemoizeIt until Algorithm 1 has fully explored all memoization candidates or until a one hour timeout occurs. We set the minimum average execution time $t_{min} = 5\mu s$ because it filters most short methods, such as accessor methods, in our environment. To study the overhead of the profiler with a large set of memoization candidates, we set the minimum relative execution time $r_{min} = 0.25\%$; to evaluate the reported memoization opportunities, we use the more realistic $r_{min} = 1\%$. For pruning reports, we set the minimum hit ratio $h_{min} = 50\%$.

To measure the performance of the DaCapo benchmarks, we use their built-in steady-state measurement infrastructure. Because the other programs are relatively short-running applications, we measure startup performance by repeatedly running them on a fresh VM, as suggested by [55]. To assess whether memoization yields a statistically significant speedup, we measure the execution time 30 times each with and without the cache, and compute the confidence interval for the difference (confidence level 95%). We report a performance difference if and only if the confidence interval excludes zero, i.e., the difference is statistically significant.

All experiments are done on an eight-core machine with two 3GHz Intel Xeon processors E5450, 8GB memory running 64-bit Ubuntu Linux 12.04.2 LTS, Java 1.6.0_27 using OpenJDK IcedTea6 1.12.5, with 4GB of heap assigned to the VM.

Table 2.2: Programs and inputs used in the evaluation, and a comparison between the iterative and exhaustive approaches. "Time" means the time for running the entire analysis (in minutes). "TO" means timeout. "Opp." is the number of reported opportunities. k_{max} is the maximum depth explored by Algorithm 1. The last column indicates whether the iterative approach outperforms the exhaustive approach.

Program	Description	Profiling input	Other input(s)	Exhaustive		Iterative			Iterative
				Time	Opp.	Time	Opp.	k_{max}	wins
Apache POI 3.9	Convert spreadsheets to text	Grades (40KB)	Statistics (13.5MB)	37	3	23	3	16	✓
Apache Tika 1.3 Excel	Convert spreadsheets to text	Two files (260KB)	Ten files (13.9MB)	56	1	58	1	64	✗
Apache Tika 1.3 Jar	Convert .jar files to text	Checkstyle, Gson	rt.jar, Soot	35	2	41	2	64	✗
Checkstyle 5.6	Check source code	Checkstyle itself	Soot	22	2	6	2	64	✓
Soot aeocec69co	Optim. & validate Java proj.	101 classes	Soot itself	TO	-	TO	13	1	✓
DaCapo-antlr	Benchmark input	Default	Large	TO	-	TO	3	16	✓
DaCapo-bloat	Benchmark input	Default	Large	TO	-	TO	4	8	✓
DaCapo-chart	Benchmark input	Default	Large	2	0	2	0	8	✓
DaCapo-fop	Benchmark input	Default	(none)	TO	-	18	2	128	✓
DaCapo-luindex	Benchmark input	Default	Large	TO	-	32	0	4	✓
DaCapo-pmd	Benchmark input	Default	Large	TO	-	TO	1	8	✓

Table 2.3: Summary of memoization opportunities found by MemoizeIt. “Rel. time” is the percentage of execution time spent in the method with the profiling input. The “Rank” columns indicate at which position a method is reported by inclusive CPU time profiling, self-time CPU time profiling, and MemoizeIt, respectively. All speedups are statistically significant, otherwise we write “-”. For DaCapo-fop there is only a single input because DaCapo’s large and default inputs are the same.

ID	Program	Method	Rel.	Calls	Pot. hit	Rank			Speedup	
			time		ratio	CPU time		MemoizeIt	Profiling	Other
			(%)		(%)	Incl.	Self		input	input
1	Apache POI	HSSFCellStyle.getDataFormatString()	12.4	7,159	99.8	12	70	1	1.11 ± 0.01	1.92 ± 0.01
2	Apache POI	DateUtil.isADateFormat(int,String)	5.3	3,630	99.9	27	6	2	1.07 ± 0.01	1.12 ± 0.01
3	Apache Tika Excel	DateUtil.isADateFormat(int,String)	1.0	4,256	99.9	189	10	1	-	1.25 ± 0.02
4	Apache Tika Jar	CompositeParser.getParsers(ParseContext)	14.2	4,698	80.9	27	2	1	1.09 ± 0.01	1.12 ± 0.02
5	Checkstyle	LocalizedMessage.getMessage()	1.3	6,138	72.4	110	12	2	-	9.95 ± 0.10
6	Soot	Scene.getActiveHierarchy()	18.4	201	99	23	253	1	1.27 ± 0.03	12.93 ± 0.05
7	DaCapo-antlr	BitSet.toArray()	14.4	19,260	96.3	36	6	1	1.04 ± 0.03	1.05 ± 0.02
8	DaCapo-bloat	PhiJoinStmt.operands()	12.0	6,713	50.9	49	52	3	1.08 ± 0.03	-
9	DaCapo-fop	FontInfo.createFontKey(String,String,String)	2.2	5,429	98.9	105	2	2	1.05 ± 0.01	NA

2.4.2 *Memoization Opportunities Found*

MemoizeIt reports potential memoization opportunities for eight of the eleven programs. We inspect the three highest ranked opportunities for each program and implement a patch that adds a cache for the most promising opportunities (Table 2.3). The second-to-last and the last column of Table 2.3 show the speedup when using the profiling input and another input, respectively. When adding a cache, we follow the implementation strategy suggested by MemoizeIt. As we have only limited knowledge of the programs, we add a cache only if it certainly preserves the program's semantics. We use the programs' unit tests to check the correctness of the modified programs.

The memoization opportunities detected by MemoizeIt confirm two important design decision. First, considering complex objects in addition to primitive input and output values is crucial to detect various memoization opportunities. Five of the nine reported methods in Table 2.3 involve non-primitive values. IDs 1 and 5 have a complex target object; ID 4 and ID 6, and ID 8 have non-primitive target objects and non-primitive return values; ID 7 returns an integer array. These results underline the importance of having an analysis able to analyze complex objects. Second, several methods have side effects but nevertheless are valid optimization opportunities (IDs 1, 4, 5, 6, 7, and 8). These methods are memoizable because the side effects are redundant. These examples illustrate how our approach differs from checking for side effect-free methods.

In the following, we describe representative examples of detected memoization opportunities.

APACHE POI The motivating example in Listing 2.1 is a memoization opportunity that MemoizeIt finds in Apache POI (ID 2). The analysis reports the method as memoizable because most calls (99.9%) pass a string that was already passed earlier to the method. The memoization opportunity is also reported when analyzing Apache Tika Excel (ID 3) because Tika builds upon POI. Adding a single element cache that returns a cached value if the string is the same as in the last call of the method gives speedups of 1.12x and 1.25x for Apache POI and Apache Tika, respectively. We reported this problem and another performance problem (IDs 1 and 2) to the Apache POI developers who confirmed and fixed the problems.

```java
class LocalizedMessage {
  // set at startup
  private static Locale sLocale = Locale.getDefault();
  private final String mKey;    // set in constructor
  private final Object[] mArgs; // set in constructor
  private final String mBundle  // set in constructor

  String getMessage() {
    try {
      // use sLocale to get bundle via current classloader
      final ResourceBundle bundle = getBundle(mBundle);
      final String pattern = bundle.getString(mKey);
      return MessageFormat.format(pattern, mArgs);
    } catch (final MissingResourceException ex) {
      return MessageFormat.format(mKey, mArgs);
    }
  }
}
```

Listing 2.3: Memoization opportunity in Checkstyle.

CHECKSTYLE The `LocalizedMessage` class of Checkstyle represents messages about violations of coding standards and its `getMessage` method constructs and returns a message string. Listing 2.3 shows the implementation of the method. The message string for a particular instance of `LocalizedMessage` is always the same because the message depends only on final fields and on the locale, which does not change while Checkstyle is running. Nevertheless, Checkstyle unnecessarily re-constructs the message at each invocation. MemoizeIt detects this redundancy and suggests to memoize the message. Memoization does not lead to a measurable speedup for the profiling input, which applies Checkstyle to its own source code, because the Checkstyle developers adhere to their own coding standards. As an alternative input, we configure Checkstyle to search for duplicate code lines, which leads to many calls of `getMessage`. For this alternative input, adding a cache yields a 2.80x speedup (not reported in Table 2.3). When applying Checkstyle to a larger code base (the 160 KLoC of the Soot project), the benefits of the cache become even more obvious: the execution time is reduced from 6–7 minutes to about 40 seconds, giving a speedup of 9.95x.

SOOT MemoizeIt identifies a memoization opportunity for a method, `Scene.getActiveHierarchy`, that already has a cache that is accidentally invalidated more often than necessary. The method either computes the class hierarchy of the program analyzed by Soot or reuses the already-computed hierarchy. MemoizeIt reveals this opportunity because the method repeatedly recomputes the same hierarchy in one of the last phases of Soot, i.e., at a point in time when the class hierarchy does not change anymore. The problem is that the existing cache is flushed every time before Soot validates the intermediate representation of a method body. To avoid redundantly computing the hierarchy, it suffices to remove the unnecessary flushing of the cache, which gives 1.27x speedup with the profiling input. An alternative, larger input that runs Soot on its own source code yields a speedup of 12.93x. The speedup for the second input is so much higher because the second input triggers significantly more recomputations of the class hierarchy than the profiling input. We reported this performance bug to the Soot developers who confirmed and fixed the problem.

Table 2.4: Data computed for suggesting a cache implementation and suggested implementation. The columns labels are abbreviations for HR=hit ratio, I=needs invalidation, S=size.

ID	Instance-level					Global					Suggestion
	Single		Multi			Single		Multi			
	HR	I	HR	S	I	HR	I	HR	S	I	
1	100	no	100	1	no	88	no	100	12	no	single, global
2	-	-	-	-	-	97	no	100	3	no	single, global
3	-	-	-	-	-	97	no	100	3	no	single, global
4	82	no	79	1	no	21	yes	72	10	yes	single, instance
5	72	no	72	1	no	21	no	72	1,696	no	single, instance
6	99	no	99	1	no	50	no	99	2	no	single, instance
7	96	no	96	1	no	5	no	96	708	no	single, instance
8	60	no	60	1	no	4	no	60	2,655	no	single, instance
9	-	-	-	-	-	41	no	99	57	no	multi, global

2.4.3 *Suggestions for Implementing Caches*

When implementing caches for the reported optimization opportunities, we follow the implementation strategies suggested by MemoizeIt. Table 2.4 lists the data that the approach computes to suggest a cache implementation and the kind of cache that MemoizeIt suggests. For example, for memoization opportunity 4 (as listed in Table 2.3), MemoizeIt's simulation of different kinds of caches shows that a global single-element cache would achieve only 21% hits and require to invalidate the cache, whereas an instance-level single-element cache would have 82% hits and not require invalidation. Therefore, MemoizeIt suggests to exploit this memoization opportunity with an instance-level single-element cache. For all but one of the memoization opportunities in Table 2.4, following MemoizeIt's suggestion yields a profitable optimization. The one exception is memoization opportunity 7, where the first suggestion does not lead to any speedup, but where the second suggestion, a global multi-element cache, turns out to be profitable. Such suboptimal suggestions are possible because the suggestions are based on a simple model of caching behavior, which, for example, ignores the effects of JIT optimizations.

```
 1 class HSSFCellStyle {
 2   // inputs
 3   private static short lastDateFormat = INVALID_VALUE;
 4   private static List<FormatRecord> lastFormats = null;
 5   // output
 6   private static String cache = null;
 7
 8   String getDataFormatString() {
 9     if (cache != null &&
10         lastDateFormat == getDataFormat() &&
11         lastFormats.equals(_workbook.getFormats())) {
12       return cache;
13     }
14     lastFormats = _workbook.getFormats();
15     lastDateFormat = getDataFormat();
16     cache = getDataFormatString(_workbook);
17     return cache;
18   }
19
20   void cloneStyleFrom(HSSFCellStyle source) {
21     _format.cloneStyleFrom( source._format );
22     if (_workbook != source._workbook) {
23       lastDateFormat = INVALID_VALUE; // invalidate cache
24       lastFormats = null;
25       cache = null;
26       // ...
27 } } }
```

Listing 2.4: Cache implementation in Apache POI.

EXAMPLE OF CACHE IMPLEMENTATION. Listing 2.4 shows a cache implementation that exploits optimization opportunity 1 in method `getDataFormatString()`. Following MemoizeIt's suggestion to implement a global single element cache, we add three static fields (lines 3 to 6): Fields `lastDateFormat` and `lastFormats` store the input key of the cache; field `cache` contains the cached result. We modify the method so that it returns the cached result if the stored inputs match the current inputs (lines 9 to 13). Otherwise, the method executes as usually and fills the cache. MemoizeIt suggests that the cache may not require invalidation because the profiled executions do not trigger any path that requires invalidation. However, inspecting the source code reveals that `cloneStyleFrom(HSSFCellStyle)` writes into fields that may invalidate a previously stored result. To ensure that caching preserves the program's semantics for all execution paths, we invalidate the cache in lines 23 to 25. We reported this cache implementation to the Apache POI developers who integrated the change into their code.

2.4.4 *Precision of the Analysis*

The effectiveness of an approach to find memoization opportunities largely depends on how quickly a developer can identify valuable optimization opportunities based on the results of the analysis. MemoizeIt reports a ranked list of methods, and we expect developers to inspect them starting at the top. As shown by the "Rank, MemoizeIt" column of Table 2.3, the memoization opportunities that give rise to speedups are quickly found: five are reported as the top opportunity of the program and three are ranked as the second opportunity.

COMPARISON WITH CPU TIME PROFILING. We compare MemoizeIt to two state-of-the-art profiling approaches: (i) Rank methods by inclusive CPU time, that is, including the time spent in callees, and (ii) rank methods by CPU self-time, that is, excluding the time spent in callees. In Table 2.3, the "Rank, CPU time" columns show the rank that CPU time profiling assigns to the methods with memoization opportunities. Both CPU time-based approaches report many other methods before the opportunities found by MemoizeIt, illustrating a weakness of CPU time profiling: It shows where time is spent but not where time is wasted [103]. Instead of overwhelming developers with hot but not necessarily optimizable methods, MemoizeIt points developers to a small set of methods that are likely candidates for a simple and well-known optimization.

NON-OPTIMIZABLE METHODS. MemoizeIt may report methods that are not easily memoizable. We find two causes for such reports in the evaluated programs. First, some methods already use memoization but are reported because their execution time is relatively high despite the cache. Even though reports about such methods do not reveal new optimization opportunities, they confirm that our analysis finds valid memoization opportunities and helps the developer understand performance problems.

Second, some methods have non-redundant side effects that the analysis does not consider. For example, a method reads a chunk of data from a file and advances the file pointer. Although the method may return the same chunk of data multiple times, its behavior cannot be replaced by returning a cached value because the file pointer must be advanced to ensure reading the expected data in later invocations of the method. In most cases, the analysis effectively addresses the problem of non-redundant side effects by considering the target object as part of the call's input, but the analysis fails to identify some side effects, such as advancing file pointers. A strict side effect analysis [149] could avoid reporting such methods but would also remove six of the nine valid optimization opportunities that MemoizeIt reveals, including the 12.93x-speedup opportunity in Soot.

```
 1 class CachingBloatContext extends PersistentBloatContext {
 2
 3   Map fieldInfos; // Declared and initialized in superclass
 4
 5   public FieldEditor editField(MemberRef field) {
 6     // Lookup in existing cache
 7     FieldInfo info = fieldInfos.get(field);
 8     if (info == null) {
 9       // Compute field infos and store them for reuse
10       FieldInfo[] fields = /* Get declaring class */
11       for (int i = 0; i < fields.length; i++) {
12         FieldEditor fe = editField(fields[i]);
13           if (/* field[i] is field */) {
14             fieldInfos.put(field, fields[i]);
15             return fe;
16           }
17         ...
18       }
19       ...
20     }
21     return editField(info);
22 } }
```

Listing 2.5: Non-memoizable method in DaCapo-bloat.

Another potential cause for reporting non-memoizable methods, which does not occur in our experiments, are non-deterministic methods. MemoizeIt can filter such methods by checking for inputs that lead to multiple outputs.

Listing 2.5 shows an example of a memoization candidate in Dacapo-bloat, which we cannot further optimize because the method already uses memoization. At line 7 the parameter `field` is used as the key of an instance-level, multi-element cache. If `field` is not yet in the cache, then the associated field information is created and added to the cache. MemoizeIt reports this method because the hit ratio is 57.65%, which confirms the developers' intuition to implement a cache.

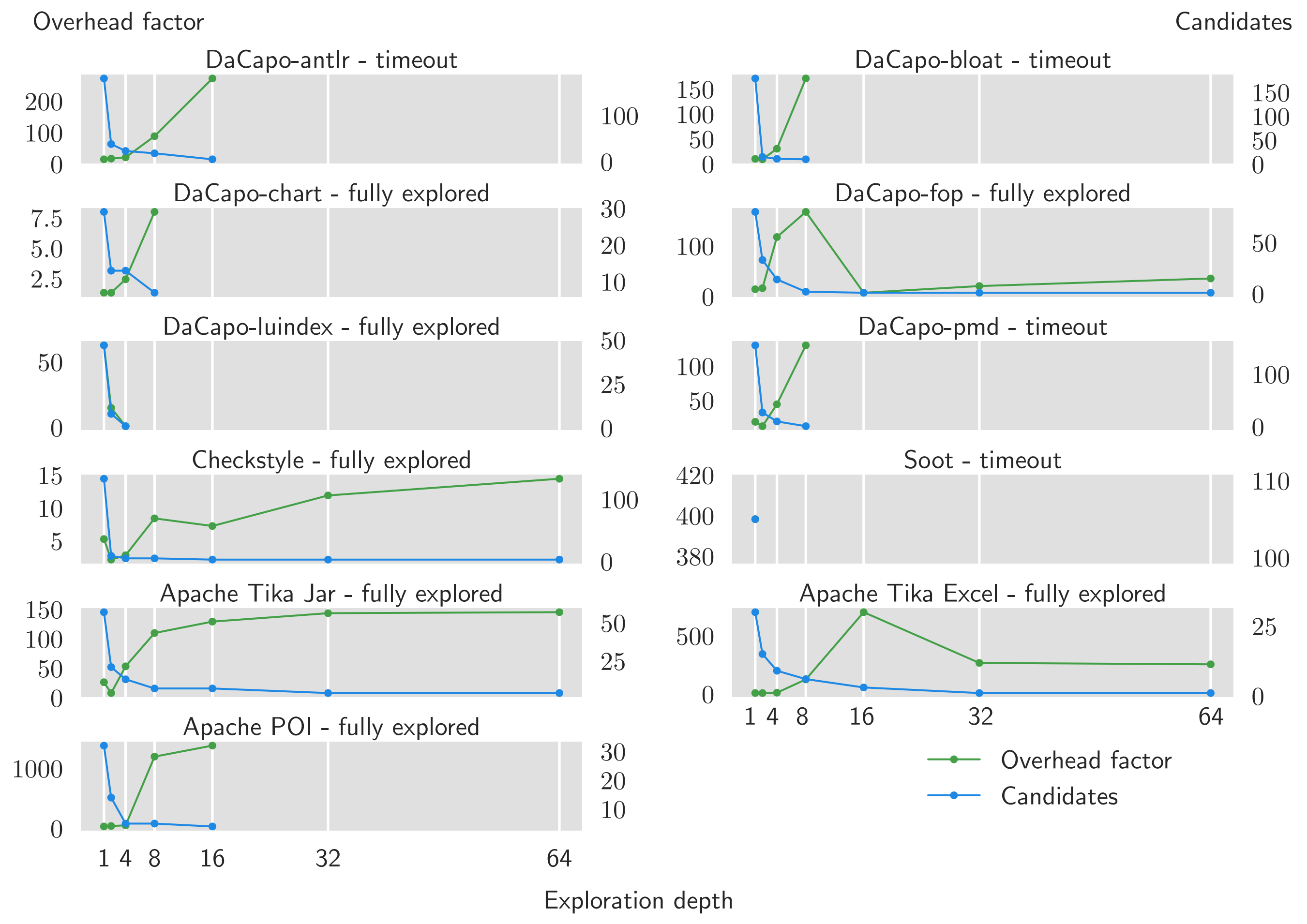


Figure 2.2: Number of memoization candidates and overhead depending on the exploration depth *k*.

2.4.5 *Iterative vs. Exhaustive Profiling*

We compare MemoizeIt's approach of iteratively refining memoization candidates to a more naive approach that exhaustively explores all input-output tuples in a single execution. The last six columns of Table 2.2 show how long both approaches take and how many opportunities they report. The last column shows that the iterative approach clearly outperforms exhaustive exploration. For six programs, exhaustive exploration cannot completely analyze the execution within one hour ("TO") and therefore does not report any opportunities, whereas the iterative approach reports opportunities even if it does not fully explore all methods within the given time. For three programs, both approaches terminate and the iterative approach is faster. For the remaining two programs both approaches takes a comparable amount of time, while reporting the same opportunities.

2.4.6 *Performance of the Analysis*

Our prototype implementation of the MemoizeIt approach yields actionable results after at most one hour (Table 2.2, column "Iterative, Time") for all programs used in the evaluation. Therefore, we consider the approach to be appropriate as an automated tool for in-house performance analysis.

The runtime overhead imposed by MemoizeIt depends on the exploration depth k. Figure 2.2 shows the profiling overhead (left y-axes) throughout the iterative refinement algorithm, i.e., as a function of the exploration depth k. The profiling overhead imposed by MemoizeIt is influenced by two factors. On the one hand, a higher depth requires the analysis to explore the input and output of calls in more detail, which increases the overhead. On the other hand, the analysis iteratively prunes more and more methods while the depth increases, which decreases overhead. The interplay of these two factors explains why the overhead does not increase monotonically for some programs, such as DaCapo-fop, and Apache Tika Excel. In general, we observe an increasing overhead for larger values of k because recording large object graphs requires substantial effort, even with a small set of memoization candidates.

Figure 2.2 also shows the number of memoization candidates (right y-axes) depending on the exploration depth. The figure illustrates that the iterative approach quickly removes many potential memoization opportunities. For example, MemoizeIt initially considers 30 methods of Apache Tika Excel and reduces the number of methods to analyze to 15, 9, and 6 methods after one, two, and three iterations, respectively.

2.5 RELATED WORK

2.5.1 *Detecting Performance Problems*

There exists various dynamic analyses that find excessive memory and CPU usage, e. g., by searching for equal or similar objects [93, 145], overuse of temporary structures [49], under-utilized or over-utilized containers [148], unnecessarily copied data [146], objects where the cost to create them exceeds the benefit from using them [147], and similar memory access patterns [103]. Yan et al. use reference propagation profiling to detect common patterns of excessive memory usage [150]. Jovic et al. propose a profiler to identify long latencies of UI event handlers [76]. All these approaches focus on particular "symptoms" of a performance problem, which help developers to identify a problem but not to find a fix for it. In contrast, MemoizeIt focuses on a well-known "cure", memoization, and searches for methods where this cure is applicable.

Several profiling approaches help developers find performance bugs due to asymptotic inefficiencies [39, 59, 157]. Xiao et al. describe a multi-execution profiler to find methods that do not scale well to large inputs [143]. These approaches relate the input of a computation to execution time to help developers reduce the complexity of the computation. Instead, MemoizeIt relates the input of a computation to its output to help developers avoid the computations.

2.5.2 *Understanding Performance Problems*

Approaches to diagnose performance problems include statistical debugging, which identifies program predicates that correlate with slow executions [127], the analysis of idle times in server applications caused by thread synchronization issues or excessive system activities [11], and dynamic taint analysis to discover root causes of performance problems in production software [14]. Other approaches analyze execution traces [155] or stack traces [65] from a large number of executions to ease performance debugging. These approaches are aimed at understanding the root cause of a performance problem, whereas MemoizeIt discovers problems that a developer may not be aware of.

2.5.3 *Fixing Performance Problems*

Nistor et al. propose a static analysis that detects loop-related performance problems and that proposes source code transformations to fix these problems [102]. MemoizeIt is orthogonal to their work because it addresses a different class of performance bugs. Chen et al. present CacheOptimizer, a framework to automatically find and fix caching opportunities in programs that use the Hibernate [5] framework [32]. In contrast to MemoizeIt, CacheOptimizer focuses only on one specific framework, when MemoizeIt can be use to detect memoization opportunities in any Java program.

2.5.4 *Compiler Optimizations*

Compilers and runtime systems can automatically memoize the results of some computations. Ding et al. propose a profiling-based, static compiler optimization that identifies deterministic code segments and that adds memoization via a source to source transformation [46]. Their approach seems to be limited to primitive input values, whereas MemoizeIt addresses the problem of summarizing large object structures. Xu et al. propose a dynamic purity analysis and apply it in a compiler optimization that caches method return values of dynamically pure methods [149]. Their analysis considers the parameters of a method as its only input, and adds a global cache if a method is found to be a memoization candidate. Guo et al. propose an optimization that adds caches to long running functions for programs written in a dynamically typed language [64]. They target applications that elaborate data in stages and add caches after each stage.

As conservative compiler optimizations, the above approaches can memoize a computation only if it is side effect-free. These optimizations miss various memoization opportunities identified by MemoizeIt because the methods have redundant side effects.

Shankar et al. propose a dynamic analysis that identifies methods that create many short-lived objects and a runtime optimization that inlines those methods to enable other optimizations [125]. Shankar et al. propose a code specialization technique embedded in a JIT compiler that optimizes execution paths based on runtime values observed during execution [126]. Costa et al. propose a JIT optimization that speculatively specializes functions based on previously observed parameters [42]. Combined with other optimizations, such as constant propagation, their approach can have an effect similar to automatically added memoization.

In contrast to MemoizeIt, their work focuses in primitive values instead of complex object structures.

Sartor et al. [119] and Huang et al. [70] use a compiler and runtime infrastructure to optimize programs towards better usage of hardware caches. While memoization can indirectly affect hardware caching, MemoizeIt focuses on software caches where performance is improved by avoiding repetitive expensive computations.

JITProf [60] profiles JavaScript programs to identify code locations that prohibit profitable JIT optimizations. Optimization coaching [129] aims at improving the feedback given by a compiler to the programmer to help the programmer enable additional optimizations. These approaches optimize a program for a particular compiler or execution environment, whereas MemoizeIt identifies platform-independent optimization opportunities.

2.5.5 *Other Related Work*

Ma et al. empirically study the effectiveness of caching web resources loaded in a mobile browser and suggest strategies for improving the effectiveness [90]. MemoizeIt shares the idea of analyzing caching opportunities but it addresses redundant computations instead of redundant transfers of data over the network.

Depth-first iterative-deepening is a tree search algorithm that repeatedly applies a depth-first search up to an iteratively increasing depth limit [82]. The iterative refinement of the dynamic analysis introduced in Section 2.2.2 shares the idea of iteratively increasing the depth of exploration to reduce the complexity of the problem. MemoizeIt's iterative refinement differs from iterative-deepening because it does not perform a tree search and because it does not stop once a solution is found but it prunes methods from the search space that are guaranteed not to be memoization opportunities.

Incremental computation, a technique to update the results of a computation when the input changes [111], memoizes partial results across program executions. Instead, MemoizeIt focuses on memoization opportunities within a single execution.

Biswas et al. present DoubleChecker, an atomicity checker that executes the program twice, with an imprecise and a precise analysis, respectively [17]. Similar to MemoizeIt, DoubleChecker increases precision based on the results of a less precise analysis. Their approach may miss some atomicity violations due to different thread-schedules. In contrast to DoubleChecker, MemoizeIt does not bound the number of iterations a-priori and guarantees that the iterative analysis does not miss any memoization opportunities.

2.6 LIMITATIONS

DETAILS OF CACHE IMPLEMENTATION. MemoizeIt identifies memoization opportunities and outlines a potential cache implementation, but leaves the decision whether and how exactly to implement memoization to the developer. In particular, the developer carefully decides whether memoization might break the semantics of the program and whether the benefits of memoization outweigh its cost (such as increased memory usage). Furthermore, MemoizeIt's suggestions may not be accurate because they are based on a limited set of executions. In particular, a cache may require invalidation even though MemoizeIt does not suggest it.

INPUT SELECTION. As all profiling approaches that we are aware of, including traditional CPU time profiling, MemoizeIt requires the developer to provide input that drives the execution of the program. The memoization candidates provided by MemoizeIt are valid only for the given inputs, i. e., other inputs may lead to other sets of memoization candidates.

MULTI-THREADING. The current prototype-implementation does not support multi-threaded programs; if multiple threads are running while traversing the heap graph there is the risk that an object is modified during the traversal. To avoid this issue an updated implementation can: (a) record a serialized execution of the a program and replay the serialized execution with our iterative approach, or (b) at each method entry and exit point stop all the running threads, and resume the program execution only after the heap-traversal for the current method is terminated.

NON-DETERMINISTIC PROGRAMS. The current implementation of MemoizeIt iterative refinement of memoization opportunities assumes that each execution of the program under the same input is deterministic. However this assumption may not always hold in practice. In our evaluation we did not encounter this problem.

2.7 SUMMARY

In this chapter we presented MemoizeIt, a dynamic program analysis that helps developer automate the task of discovering performance bottlenecks. The reports that MemoizeIt produces reveal to developers methods that can be optimized through memoization. The approach reports methods where memoization is likely to be beneficial because the method repeatedly transforms the same inputs into the same outputs. Key to scaling the approach to complex object-oriented programs is an iterative analysis algorithm that increases the degree of detail of the analysis while shrinking the set of methods to analyze. Based on the profiled information MemoizeIt give hints on applying memoization. The report that MemoizeIt produces provide simple but actionable suggestions how to apply this well-known optimization to a profiled method. The work introduced in this chapter lays the foundation for a practical tool that supports developers in improving their code's efficiency by providing them actionable reports about performance bugs. We demonstrate that our approach is useful practice by evaluating the approach to real-world Java programs. MemoizeIt finds nine previously unknown memoization opportunities reducing the program execution time up to a factor of 12.93x. We report these fixes to developers that integrated MemoizeIt suggestions into the program's code base.

3 PROGRAM SYNTHESIS FOR PERFORMANCE

The main contribution of this chapter is to address the second limitation of profilers, namely, that to fully understand the performance behavior of a program a developer must execute the program with multiple inputs and with different input sizes. However, this is difficult to achieve in practice because manually creating inputs is costly, and detecting an inefficient piece of code may require to exercise the code with a specific kind of input. To make the problem even worse, most of the code has manually crafted or automatically generated tests for correctness, but these tests typically focus on covering each statement, branch, or path once. In general, these tests are usually insufficient to detect performance problems, and the lack of performance tests is pervasive even among open-source projects [85]. Therefore we argue that there is a need for new tools to automatically generate inputs that attempt to trigger performance bottlenecks.

A body of previous work has shown that real-world software often suffers from performance bottlenecks [75, 88, 120], and that some of them can be fixed with relatively little effort [60, 102, 137]. Unfortunately, finding such bottlenecks often is non-trivial as this step requires executing the inefficient code with input data that brings the bottleneck to the attention of a profiling tool, a developer, or in the worst case, a user. Because it is desirable to find bottlenecks before deploying the software, techniques for exposing bottlenecks as part of in-house quality control are needed [85]. Existing work addresses the problem of detecting and reporting a performance problem in a given piece of code, e. g., by empirically measuring the complexity of code through profiling [39, 59, 157] or by identifying instances of known performance anti-patterns [60, 75, 88, 103, 120, 146, 148]. The focus of this chapter, to trigger an execution that exposes the bottleneck, is currently understudied.

To address this problem, we introduce PerfSyn, an automatic approach for synthesizing "inputs" that expose performance bottlenecks. PerfSyn focuses on analyzing a unit-level piece-of-code, i. e., to find bottlenecks in individual methods. In the case of unit-tests, "inputs" are test programs that prepare an object under test and then call the method under test. However PerfSyn allows to analyze software at a finer level of granularity, for example at the basic-block level, or even single expressions. PerfSyn starts with a minimal usage example of the method under test and then applies a sequence of program mutations that add, remove, or modify statements in the program.

PerfSyn is the first framework for synthesizing programs that expose performance-bottlenecks:

- The main contribution of this chapter is to formulate the problem of finding a sequence of program changes that yields a bottleneck-exposing program as a combinatorial search problem. To efficiently address the combinatorial search problem we adopt well-known graph search algorithms [48, 66]. To this end, the approach learns from the execution feedback of synthesized programs to steer the search towards program changes that generate bottleneck-exposing programs.
- The second contribution of this chapter is to design PerfSyn as a general framework that can expose different kinds of performance bottlenecks. Once the approach hits a program that exposes a specific bottleneck, this program along with information recorded from the program execution is reported to the developers, who then can fix the problem.
- The last contribution of the chapter is to provide empirical evidence that PerfSyn is effective and efficient in exposing performance problems in real-world Java classes. We evaluate PerfSyn in two usage scenarios. In the first scenario we compare the performance of two versions of a program, e. g., for performance regression testing. In the second scenario a developer validates his/her assumption about the computational complexity of code against the actual performance, i. e., execution time of the code.

We begin this chapter by motivating our work in Section 3.1 showing how the approach works with a running example. In Section 3.2 we describe the approach in detail. In Section 3.3 we describe how we implemented a prototype for PerfSyn. We evaluate our implementation in Section 3.4. We list the limitations of the approach in Section 3.6 and we summarize our findings in Section 3.7.

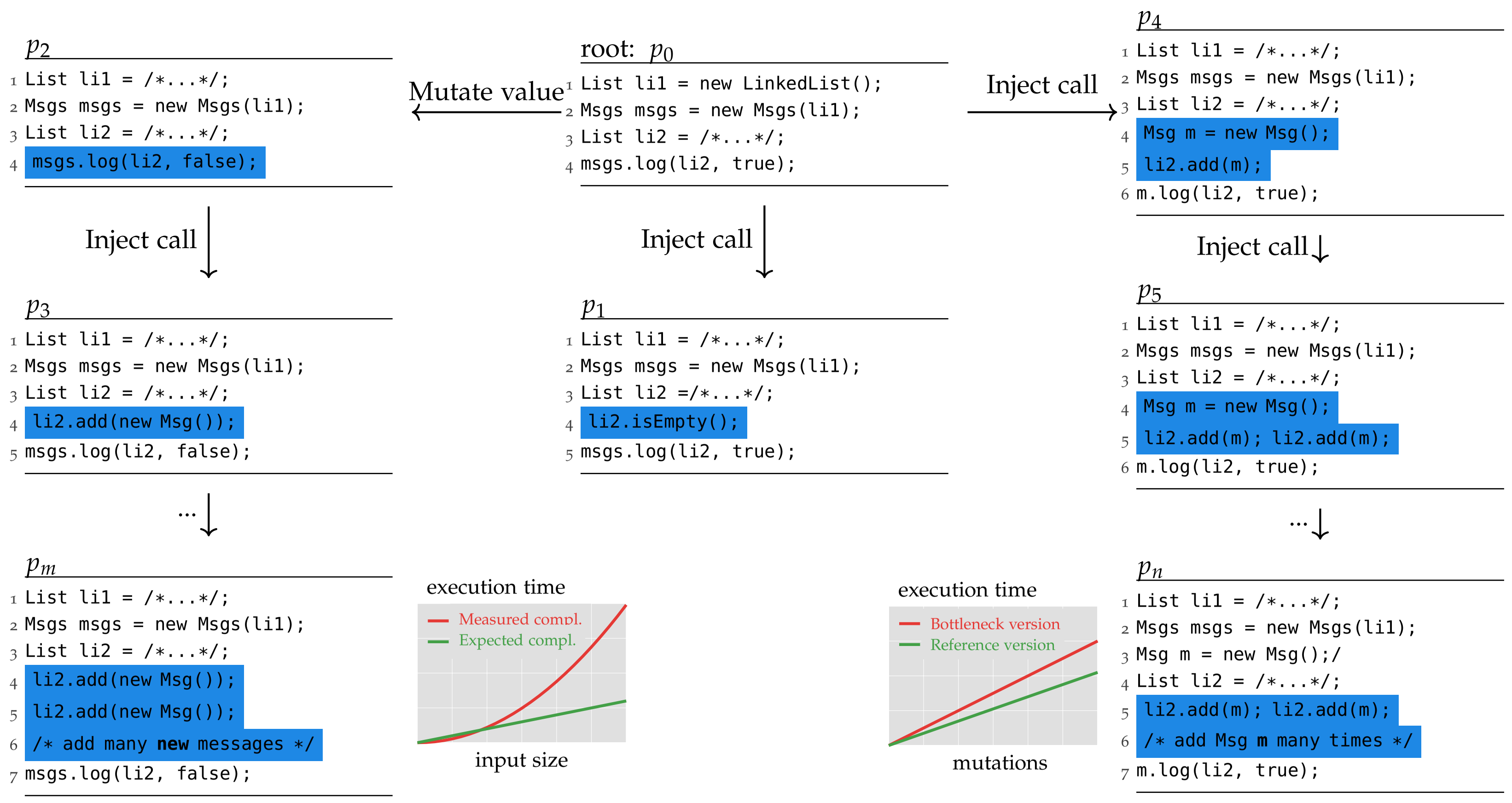
```java
class Messages {
  Set<Msg> set = ...
  List<Msg> list = ...

  Messages(List<Msg> msgs) {
    /* Add messages to set and list */
  }

  void log(List<Msg> newMsgs, boolean renameDups) {
    for (Msg msg : newMsgs) {
      if (renameDups) {
        if (set.contains(msg)) {
          logOne(makeUnique(msg));
        } else {
          logOne(msg);
        }
      } else {
        /* Bottleneck 1:
         * Inefficient containment check */
        if (!list.contains(msg)) {
          logOne(msg);
        }
      }
    }
  }

  void logOne(Msg msg) {
    /* Add msg to set and list */
  }

  /* Bottleneck 2: Expensive hash function */
  Msg makeUnique(Msg msg) {
    /* append expensive-to-compute hash to msg */
  }
}
```

Listing 3.1: Class with two bottlenecks.

3.1 PERFSYN BY ILLUSTRATION

Listing 3.1 shows the code of `Messages`, a container class for logging messages. The class ensures that each logged message is assigned a message identifier. To store messages the class provides the method `log`, which takes two arguments: a list of new messages and a flag to indicate whether duplicate messages should be renamed into unique messages. The implementation of the class stores messages in a list data-structure. It uses a list to check whether a message has already been logged and to represent the order of messages. The helper method `logOne` adds a new message to the list. If the user wants duplicate messages to be renamed, then another helper method, `makeUnique`, changes the message by appending an unique string. The example contains two performance bottlenecks:

- The first bottleneck at line 20 is an inefficient containment check that accidentally uses the expensive `List.contains` instead of the faster `Set.contains`. A developer may expect the containment check to require constant time, but instead, the required time is linear with respect to the number of already logged messages, making the overall complexity of `log` quadratic in the number of messages.
- The second bottleneck causes suboptimal performance when an already contained messages get renamed. The reason is that `makeUnique` uses an expensive hash function to append a unique string to the message (line 33). Instead, a more efficient implementation would be to append a unique number obtained via a global counter.



root: p_0
```
List li1 = new LinkedList();
Msgs msgs = new Msgs(li1);
List li2 = /*...*/;
msgs.log(li2, true);
```

p_1
```
List li1 = /*...*/;
Msgs msgs = new Msgs(li1);
List li2 =/*...*/;
li2.isEmpty();
msgs.log(li2, true);
```

p_2
```
List li1 = /*...*/;
Msgs msgs = new Msgs(li1);
List li2 = /*...*/;
msgs.log(li2, false);
```

p_3
```
List li1 = /*...*/;
Msgs msgs = new Msgs(li1);
List li2 = /*...*/;
li2.add(new Msg());
msgs.log(li2, false);
```

p_m
```
List li1 = /*...*/;
Msgs msgs = new Msgs(li1);
List li2 = /*...*/;
li2.add(new Msg());
li2.add(new Msg());
/* add many new messages */
msgs.log(li2, false);
```

p_4
```
List li1 = /*...*/;
Msgs msgs = new Msgs(li1);
List li2 = /*...*/;
Msg m = new Msg();
li2.add(m);
m.log(li2, true);
```

p_5
```
List li1 = /*...*/;
Msgs msgs = new Msgs(li1);
List li2 = /*...*/;
Msg m = new Msg();
li2.add(m); li2.add(m);
m.log(li2, true);
```

p_n
```
List li1 = /*...*/;
Msgs msgs = new Msgs(li1);
Msg m = new Msg();/
List li2 = /*...*/;
li2.add(m); li2.add(m);
/* add Msg m many times */
m.log(li2, true);
```

Figure 3.1: Running example. Each box represents a program. The plots at the bottom summarize execution feedback for the left and right path, respectively.

3.1.1 *Challenge 1: Expose Bottlenecks*

Finding bottlenecks, such as those in our motivating example, in complex software is non-trivial. The most successful approach to find bottlenecks in practice is profiling. However, effective profiling depends on inputs that trigger an execution that exposes a bottleneck. In practice, most code comes without extensive performance tests but only a correctness test suite, or maybe even only a set of minimal usage examples [85].

For the example, program p_0 in Figure 3.1 shows a minimal usage example of the `Messages` class. The program initializes the class with an empty list of messages and then calls `log` with two arguments: an empty list and the constant `true`. Profiling an execution of this program does not expose the performance bottlenecks in the class. For bottleneck 1, the program fails to trigger the inefficient check because it requires setting the flag to `false` and to log at least two messages. For bottleneck 2, the program fails to expose the inefficiency because it requires repeatedly logging the same message.

3.1.2 *Challenge 2: Synthesizing Bottleneck-Exposing Programs*

PerfSyn searches for bottlenecks in the method under test m_{ut} starting from an initial program p_0. The program p_0 in Figure 3.1 is generated by PerfSyn and it contains the minimal number of statements required to execute m_{ut} without a crash. To search for a bottleneck PerfSyn modifies p_0, e.g., by inserting method calls or by modifying the values passed to the calls. The approach identifies a program as bottleneck-exposing based on a configurable performance oracle. The term "oracle" is inspired by test oracles, which decide whether a test exposes an error [15]. In contrast, the performance oracle decides to what extent a program exposes a performance bottleneck.

For example, the performance oracle may report that the measured complexity differs from the expected worst-case complexity [39, 59, 157] or that two implementations with supposedly equal performance have different performance properties.

For the two bottlenecks in our example, let `log` be the method under test. Suppose that PerfSyn modifies the initial program p_0 into the programs p_m and p_n in Figure 3.1, respectively. The sequence of program transformations that led from p_0 to p_m initialize the class and then add an increasingly large list of messages while setting the `renameDups` flag to `false`. That is, the programs reach the inefficient check at line 20 with an increasingly large input. PerfSyn profiles the execution of `log` with these programs and creates the performance plot shown next to program p_m. The plot shows the execution time depending on the input size and compares the measured values to the expected complexity.

For the second bottleneck, suppose that an alternative version of the `Messages` class uses a more efficient implementation of `makeUnique`. PerfSyn exposes the bottleneck in Figure 3.1 using the programs that lead to p_n in Figure 3.1. The programs pass to `log` an increasingly large number of duplicate messages, each of which triggers the `makeUnique` function. The performance plot next to p_n compares the two implementations with each other and shows that the implementation in Figure 3.1 performs sub-optimally.

3.1.3 *Challenge 3: Targeted Search*

The key challenge in exposing bottlenecks by automatically synthesizing programs is the large space of possible programs. PerfSyn is a novel approach to address this challenge based on feedback obtained from executing programs. Starting from the initial program p_0, PerfSyn represents the space of possible modifications as a tree, where p_0 is the root, each node is another possible program, and edges represent code modifications that turn one program into another one. The approach explores the tree while gathering feedback about how effective specific program changes are at getting closer to a bottleneck-exposing program. The feedback depends on the performance oracle used to identify bottleneck-exposing programs. For example, an oracle aimed at exposing an unexpected complexity class steers the approach toward programs where the observed complexity diverges more and more from the expected complexity. Likewise, an oracle aimed at exposing performance differences between two implementations targets the approach toward programs with such differences.

For the example in Figure 3.1, suppose that PerfSyn at first modifies p_0 by calling `isEmpty` on the list given to `log`, as shown in program p_1 in Figure 3.1. Because this mutation does not influence the performance of the method under test in the intended way, the approach decreases the priority of this part of the tree. Instead, PerfSyn learns over time that applying the mutations that lead to p_m and p_n are beneficial and steers the search along these mutations.

3.2 APPROACH

PerfSyn is a feedback-directed code synthesis approach with the goal to synthesize a program that exposes a bottleneck. The feedback is in the form of runtime performance measurements that are collected while running a synthesized program. We say that a method suffers from a bottleneck when the value of a performance property of the executed method with an input of a particular size exceeds the value expected by the developer. We design PerfSyn as a generic framework that supports different kinds of bottlenecks and strategies to expose them. In Section 3.2.7 we show two examples of these strategies, and how we integrated them in the PerfSyn framework.

The approach takes as input: (i) a class source-code and, (ii) one or multiple m_{ut} that must be declared in the source-code. The approach then analyses the source-code to extract type information and to build the type hierarchy. This phase is implementation dependent and will not be described in this section, Section 3.3 briefly describes how we implemented the type analysis for Java classes. The results of the type analysis are stored for later reuse during the program synthesis. The approach proceeds and analyzes one m_{ut} at the time. For each m_{ut} the approach builds an initial program p_0 that exercises the m_{ut}, i. e., p_0 is the input to the program synthesis phase (see Section 3.2.2). The main task of PerfSyn is the program synthesis phase where PerfSyn iteratively generates and executes multiple programs to trigger a performance bottleneck. The approach terminates by reporting to the developer a ranked list programs that exercise m_{ut} and that expose a bottleneck according to the performance oracle (see Section 3.2.8).

3.2.1 *Framework Components*

In this section we describe the main components of the PerfSyn framework.

Synthesized Program

The main data structure created and manipulated by PerfSyn is a program that uses the method under test once or multiple times

Definition 7 (Program). *A program $p \in \mathcal{P}$ for a method under test m_{ut} is a sequence of statements $s \in \mathcal{S}_p$ of the form $v_o = v_{arg_0}.m(v_{arg_1}, .., v_{arg_k})$, or $v_o = OP(v_{arg_0}, .., v_{arg_k})$ where*

- *$\mathcal{P}$ is the set of all programs,*
- *$\mathcal{S}_p$ is the set of statements in p,*
- *v_o and $v_{arg_{0..k}}$ are typed variables in the set of variable for the program $\mathcal{V}_p$,*
- *m is a method name,*
- *OP is a type-specific operator,*
- *and there exists at least one statement s where $m = m_{ut}$.*

This means that a program must execute m_{ut} at least once. A common instance of this formulation is the instance where the last statement executes the method under test. However there exists instances of the formulation where it could be necessary to execute the method m_{ut} multiple times because of the type of performance property that needs to be collected. For example, one can imagine a program analysis that looks for memoization opportunities. In this scenario multiple calls to the m_{ut} in a program are required to check for redundant computations [137].

Take for example the program p_0 in Figure 3.1. The program p_0 is represented in our formulation in the following form:

$$\begin{aligned} s_0 &: v_0 = constructor(LinkedList) \\ s_1 &: v_1 = constructor(Msgs, v_0) \\ s_2 &: v_2 = constructur(ArrayList) \\ s_3 &: v_3 = false \\ s_4 &: v_1.log(v_2, v_3) \end{aligned}$$

In the formulation variables are renamed as `li1` $= v_0$, `msgs` $= v_1$, `li2` $= v_2$ and the second constant argument to the m_{ut} is hoisted and assigned to v_3. Notice that each statement represents a single assignment to a new variable or a reassignment to an existing variable. This simplified program representation allows to trivially extend and compare generated programs (e. g., to compare for added statements, or to check for modified values). A similar representation has been successfully used in the past by other random unit test generators [53, 105].

Mutation Tree

The initial program p_0 can be automatically generated by PerfSyn, or in the case that PerfSyn is not able to automatically generate the program because the setup code is too complicated (e. g., because it requires a data-structure to be in a specific state), PerfSyn can start from an existing test. The existing test could be a manually written test by a developer, for example, a correctness test, or could be a test that is generated by another generator [45, 53, 105].

To derive a program that exposes a bottleneck, PerfSyn performs sequences of mutations that, starting from p_0, feed the result of each mutation into the subsequent mutation. The space of all possible sequences of mutations for a specific initial program forms a tree:

Definition 8 (Mutation tree)**.** *A mutation tree for an initial program* p_0 *is an acyclic, connected, and directed graph* $(\mathcal{P}, \mathcal{M})$*, where each node* $p \in \mathcal{P}$ *is a program and each edge is a mutation* $\mu \in \mathcal{M}$ *that modifies the source node's program into the destination node's program.*

The mutation tree represents all possible sequences of mutations that can be applied to a given initial program p_0. Since the number of possible mutations per program is finite, the number of outgoing edges per node is also finite. In contrast, the number of nodes in a mutation tree is unbounded, because each mutation results in a new program that can always be further extended with a mutation.

Algorithm 2 Synthesize a bottleneck-exposing program

```
Input: Initial program $p_0$, the $k_{best}$ number of programs to report
Output: A set $\mathcal{P}_{best}$ of $k_{best}$ programs that expose a bottleneck
 1: $\mathcal{F}_{best} = \emptyset$
 2: while no timeout do
 3:     for $nb_{mut} \leftarrow 1$ to $maxMuts$ do
 4:         $p \leftarrow$ copy of $p_0$
 5:         $F \leftarrow$ empty sequence
 6:         for $step \leftarrow 1$ to $nb_{mut}$ do
 7:             $\mu \leftarrow pickMutation(p, \mathcal{L})$                ▷ Mutate program
 8:             $p \leftarrow mutate(p, \mu)$
 9:             $(f_1, .., f_j) \leftarrow execute(p)$                  ▷ Execute program
10:             add $(f_1, .., f_j)$ to $F$
11:             $score \leftarrow oracle(F)$                        ▷ Rank best solutions
12:             update $\mathcal{F}_{best}$ with $(p, score)$
13:             $\mathcal{L} \leftarrow learn(p, F)$          ▷ Learn from execution of program
14:         end for
15:     end for
16: end while
17: $\mathcal{P}_{best} = maxByScore(\mathcal{F}_{best}, k_{best})$
```

3.2.2 *Program Synthesis Algorithm*

To expose a program with a bottleneck PerfSyn repeatedly performs five steps, described in Algorithm 2:

1. *Mutate a program.* Transforms a program into another program by modifying or adding statements. To this end, the function *pickMutation* takes an existing program p and it decides which mutation to apply next (line 7).
2. *Execute and gather feedback.* Executes a program while applying a dynamic analysis that collects runtime properties of the program and that serves as a feedback (line 9).
3. *Learn.* Checks if the applied mutation changed the performance of the method under test based on the oracle feedback and it infers which mutations are the most effective (line 13).
4. *Explore.* Steers the synthesis towards mutations of the initial program that may yield a bottleneck-exposing program. To this end, the algorithm iteratively explores the mutation tree (line 4) and ranks the most effective mutations (line 11).

The search continuously updates the set of best programs $\mathcal{F}_{best}$ until exceeding a configurable time budget and then returns the top-k_{best} programs that have the highest score, i.e., the programs that are most likely to expose a bottleneck (line 17). In the next sections we describe each of these steps in detail.

3.2.3 *Mutating Programs*

PerfSyn is a *black-box* program synthesis technique, in other words PerfSyn does not perform a program analysis to help the search for bottlenecks. PerfSyn assumes that to change the runtime behavior of m_{ut}, it is sufficient to mutate the program that exercises the m_{ut}.

Definition 9 (Mutation). *A mutation $\mu \in \mathcal{M}$ transforms a given program p into another program p'.*

Each mutation consists of a *mutation operator*, i.e., how to transform a program, and of multiple *mutation operands*, i.e., the program elements subject of the program transformation. One *mutation* to a program p adds or replaces one or multiple statements of the program. The *mutation operator* and the *mutation operands* define which and how many of the programs statements are touched by the mutation.

Mutation Operators

The approach supports four kinds of operators:

- *Inject call.* This operator mutates a program by calling a method on any of the existing objects. To this end, the operator inserts a new statement $v_o = v_{arg_0}.m(v_{arg_1}, .., v_{arg_k})$ into the sequence of statements. The rationale for including this operator is to modify the state that may influence the performance of the method under test. In Figure 3.1, this operator synthesizes p_1 from p_0 by inserting the call to `li2.isEmpty()`.
- *Modify constructor.* This operator mutates a program by replacing an existing constructor call with a subtype constructor call. PerfSyn supports this operator to trigger bottlenecks that require an instance of a specific class. In Figure 3.1, this operator could synthesize a variant of p_0 where the first statement calls `new ArrayList()` instead of `new LinkedList()`.
- *Mutate value.* This operator applies a type-specific operation to one of the existing variables. Specifically, the operator inserts a statement $v_o = OP(v_{arg_0}, .., v_{arg_k})$ that applies a type-specific operator to a set of variables. PerfSyn supports operators for primitive types, such as incrementing an integer variable or toggling a boolean variable. In Figure 3.1, the operator synthesizes p_2 from p_0 by mutating the boolean value passed to `log` from `true` to `false`.
- *Mutate field.* This operator is similar to the *mutate value* operator but it applies a type-specific operation to one of the *public* fields of a variable that is of reference type. PerfSyn supports this operator for a class and instance level fields. The (side-)effects of this mutation operator can be also emulated using a sequence of *inject call* operators, granted that the call changes an object state.

Previous work applied a similar set of mutation operators has been proven effective to generate unit-test targeted towards correctness bugs [53, 105]. Despite the different objective in covering the m_{ut} control flow, PerfSyn generates an unit-test like program as the cited approaches.

Apply Mutation Operators

Applying an operator to a specific program requires several decisions: (i) PerfSyn lists the mutation operators applicable to a program, (ii) it then concretizes the mutation operator by binding program variables to the operands, and lastly (iii) it generates the list of statements that the operator must add to the program and it inserts statements to the program. Decisions (i), and (ii) are described in Algorithm 3, and Algorithm 4 describes how PerfSyn generates statements for decision (iii). To insert and replace statements in a program PerfSyn follows simple rules that modify a program depending on the mutation operator type.

Take the example in Figure 3.1 and consider the mutation that synthesizes p_3 from p_2 by applying the "inject call" operator. When applying this operator, PerfSyn decides to apply the mutation before the call to `log` and to use the `List.add` method. Furthermore, when trying to bind the argument to `add` to a variable, PerfSyn decides to create a fresh value and recursively inserts a statement, for example, a call to the constructor `new Msg()`.

Algorithm 3 describes how PerfSyn determines the set of all possible to mutations for a program p. The algorithm traverses each statement s_i to collect the mutation operators that are allowed at or before each statement s_i. It then stores the mutation operators in the set $\mathcal{O}$. These mutation operators are selected first by collecting the program's variables that are visible at s_i (line 3 to 5).

Definition 10 (Variable visibility). *A variable is visible at a program statement s_i if and only if the variable has been initialized before s_i.*

For each variable that is visible at s_i the algorithm picks one or multiple mutation operator that are compatible with the variable's type (line 5).

For example, if a variable is of reference type the algorithm adds to $\mathcal{O}$ a mutation operator *inject call* for all the public declared methods in the type, and a *replace constructor* operator for all the declared constructors. In the program p_0 from Figure 3.1 the algorithm assigns to the set $\mathcal{O}$ at s_1 multiple *replace constructor* operators for $li1$ and $msgs$, and multiple *inject call* operators all the public methods of $li1$ type (e. g., to change the behavior of `Msgs` constructor). Similarly if a variable is of primitive type, the algorithm selects a type specific operator. For example, for a variable of integer type the set $\mathcal{O}$ is $\{plus, minus\}$.

The algorithm continues and concretizes a mutation operator by assigning to each mutation operand a *concrete variable*, or an *abstract variable* that yet does not exist in the program (line 8 and 9). The concept of *abstract variable* adds the support to alter a method's runtime behavior when the behavior is dependent on a specific value for the method argument. For example, the behavior of method `log` in Figure 3.1 depends on the new messages being duplicated. Therefore the need to support to add new messages objects to the list.

The approach continues and it stores the mutation operator type compatible variables in the set o^V. For p_0 at s_2 the set o^V for the mutation operator of `List.add` contains $li1$ and an abstract variable of type `List`, and a concrete and abstract variable of type `Msg`.

The final set of mutations $\mathcal{M}_p$ is updated with a mutation for each combination (i. e., cartesian product) of valid arguments o^{args} taken from o^V (line 13) that are compatible with o^T.

To apply a mutation operator the approach concretizes all abstract variables in o^{args} with an initial value (line 12). Algorithm 4 describes how PerfSyn initialize abstract values of primitive and reference types. First the function checks the type of the variable, if the variable is of primitive type then it assigns to the variable a random value, or a value from a pool of existing constants (line 5). Otherwise if the variable is of a reference type, the function recursively construct fresh objects to initialize the abstract variable, and the constructor arguments (line 17).

The approach taken to initialize an abstract variable by Algorithm 4 is to select the *first* valid sequence of statements. The selected sequence is not guaranteed to increase the probability of finding a bottleneck. However this is not a problem because PerfSyn can apply a mutation operator that initializes a variable using a different constructor and change the arguments to the constructor later on the search process.

The final step to apply a mutation operator is to change a program p into p', by adding or by substituting a sequence of statements in p. A mutation derived from a mutation operator of type *inject call*, *mutate value*, or *mutate field* will add new statements to p before the statement position s_i. The mutation operator *modify constructor* substitutes the appropriate statements in p and adds new statements to p if necessary (e. g., if an argument to the constructor is a freshly initialized variable).

Algorithm 3 Mutations for a program p

```
1:  function MUTATIONS($p \in \mathcal{P}$)
2:      $\mathcal{M}_p \leftarrow \emptyset$
3:      for statement $s_i \leftarrow \mathcal{S}_p$ do
4:          $\mathcal{V} \leftarrow$ get variables visible at $s_i$
5:          $\mathcal{O} \leftarrow$ list of mutation operators for $\mathcal{V}$
6:          for mutation operator $o \leftarrow \mathcal{O}$ do
7:              $o^T \leftarrow$ list of types of $o$ parameters
8:              $\mathcal{V}^{concrete} \leftarrow$ variables in $\mathcal{V}$ with type $o^T$
9:              $\mathcal{V}^{abstract} \leftarrow$ abstract variables for each type in $o^T$
10:             $o^V \leftarrow \mathcal{V}^{concrete} \cup \mathcal{V}^{abstract}$
11:             for $o^{args} \leftarrow o_0^V \times o_1^V \times ... \times o_n^V$ do
12:                 $o^{args} \leftarrow$ concretize all abstract variables in $o^{args}$
13:                 $\mathcal{M}_p \leftarrow \mathcal{M}_p \cup mutation(o, o^{args})$
14:             end for
15:         end for
16:     end for
17:     return $\mathcal{M}_p$
18: end function
```

Algorithm 4 Initialization of an abstract variable

```
1:  function INITABSTRACTVARIABLE(abstract variable $v_a$ of type $T$)
2:      $\mathcal{S}_{init} = [\ \ ]$
3:      if $T$ is primitive then                    ▷ Primitive value initialization
4:          $index \leftarrow nextVariableIndex()$
5:          $value \leftarrow randomOrPoolValue(T)$
6:          add statement $v_{index} = value$ to $\mathcal{S}_{init}$
7:          return $\mathcal{S}_{init}$
8:      else                                    ▷ Recursive ctor arguments setup
9:          $C_T \leftarrow$ constructor($T$)
10:         $C_{args} = \emptyset$
11:         for parameter type $T_{param}$ of $C_T$ do
12:             $v_a \leftarrow$ abstractVar($T_{param}$)
13:             $C_{args} \leftarrow C_{args} \cup v_a$
14:             add statements calling $initAbstractVariable(v_a)$ to $\mathcal{S}_{init}$
15:         end for
16:         $index \leftarrow nextVariableIndex()$
17:         add statement $v_{index} = C_T(C_{args})$ to $\mathcal{S}_{init}$
18:         return $\mathcal{S}_{init}$
19:     end if
20: end function
```

3.2.4 *Gathering Execution Feedback*

To determine whether a program p exposes a bottleneck, PerfSyn executes and dynamically analyzes a program p.

Definition 11 (Execution feedback). *The feedback obtained by executing a program p is a tuple $(f_1, \dots, f_j)$, where each f_i $(1 \leqslant i \leqslant j)$ represents a dynamically measured property of p's execution.*

The tuple of measurements may contain any value of interest to understand the performance of the method under test, such as execution time, memory consumption, or amount of network traffic. In practice these measurements can be any type of data that a dynamic analysis can collect during a program execution.

Based on the execution feedback obtained for each program in a path through the mutation tree, an oracle decides how useful a program is for exposing a bottleneck:

Definition 12 (Performance oracle). *Let $(p_0 \xrightarrow{\mu_1} p_1 \xrightarrow{\mu_2} \dots p_k)$ be a path through the mutation tree. Given a sequence of execution feedback F, where each element represents the feedback for a p_i $(0 \leqslant i \leqslant k)$ on the path, the performance oracle is a function $F \rightarrow \mathbb{R}$. A higher value returned by the oracle indicates that the program p_k is closer to exposing a bottleneck.*

The strict requirement that the oracle must return a discrete value is imposed by the type of search that PerfSyn performs on the mutation tree. I. e., it maximizes the execution feedback from the oracle. The intuition is that maximizing such a value will steer PerfSyn closer to trigger a bottleneck.

In Section 3.2.7 we describe concrete examples of how an oracle uses the execution feedback.

3.2.5 *Learning from Executions*

PerfSyn extends the mutation tree by selecting one of the outgoing edges of the current program p. The main challenge is to decide about the potential effectiveness of a mutation $\mu_{predict}$ before having performed it. This is crucial because performing a mutation costs time: first it requires to modify the current program, second and more importantly it requires to execute the mutated program to collect the execution feedback, and third it requires the oracle to compute the score for the program starting from the execution feedback. To address this challenge, after each iteration the algorithm learns from the previously performed mutations to predict the effectiveness of similar mutations (line 13). At first, the algorithm computes the set $\mathcal{M}_{learn}$ of already performed mutations μ_{learn} that fulfill three conditions:

1. μ_{learn} and $\mu_{predict}$ share the same mutation operator;
2. the operator is applied with the same method, constructor, or type-specific operator;
3. if the mutation refers to existing variables, then both μ_{learn} and $\mu_{predict}$ share the same choice as to whether to use existing or new variables.

Based on this set of already performed mutations to learn from, the algorithm builds a prediction table $\mathcal{L}$ that indicates for each mutation how often it has increased the value returned by the oracle in previous iterations. The mutation tree is an implicit memory of the effects on m_{ut} performance. The prediction table $\mathcal{L}$ is a concrete, simple, and naive form of this memory. We leave the study to build alternative strategies to the prediction table as future work.

3.2.6 *Exploring the Search Space*

The key step of PerfSyn is to explore the mutation tree to find a bottleneck-exposing program. Because of the large number of possible mutations, exhaustively exploring the tree is impractical, even when setting a bound on the exploration depth. In principle, any search algorithm, metaheuristic, or other machine learning technique that solves combinatorial optimization problems can be used to explore the mutation tree. We implement and experimentally compare two algorithms

1. the A* algorithm [66], because it is one of the most popular and widely used graph search algorithms; and
2. Ant Colony Optimization (ACO) [48], which we find to be well-suited for our problem.

In the following sections we explain how we concretize the generic parts of the exploration strategy using these two approaches.

A Search*

A* [66] is a path finding algorithm that heuristically and iteratively builds a solution to reach a goal state. The algorithm selects at each iteration step the most promising solution, i.e., a solution that maximizes the cost function $f(n) = g(n) + h(n)$. Function $g(n)$ represents the cost for a path from the root to node n, and $h(n)$ heuristically estimates the cost of traversing node n to reach the goal.

We adapt the A* algorithm to the problem of finding in the mutation tree a path that exposes a bottleneck. A node n is a program, and the function $f(n)$ is the performance oracle's score for this program. Our version of A* does not use the heuristic function h, but PerfSyn uses only the feedback obtained from the execution $g(n)$. The issue of using the heuristic function $h(n)$ that an upper bound for the performance oracle's score cannot be known a priori. To bound the memory used by the search, we use the SMA* variant of the A* algorithm [118].

Our modified version of mutation selection based on A* is presented in Algorithm 5. The algorithm has a global state that is maintained during the entire search. The global state is an ordered set *openSet* which contains all the nodes of the tree that have to be explored, sorted by the performance oracle score (i. e., the most promising nodes are selected first). In addition A* keeps set *closedSet* that contains all so far explored nodes. The algorithm starts by collecting the list of mutations for the program p. The algorithm then applies each of p's possible mutations to collect the score for each one of them (line 3-8). This step is necessary because the A* algorithm extracts the most promising node from the *openSet* as next node to explore (line 11). The algorithm terminates by reducing the *openSet* size to a maximum user set threshold *bestOpenSize* if the size of *openSet* exceeds *maxOpenSize* value. This last step is necessary to control that the size of *openSet* does not grow too large when the explored depth of the mutation tree increases.

Algorithm 5 A* based mutation selection

```
1:  function PICKMUTATION($p \in \mathcal{P}, \mu \in \mathcal{M}$)
2:      $\mathcal{M}_p \leftarrow mutations(p)$
3:      for each $\mu \in \mathcal{M}_p$ do
4:          $p' \leftarrow mutate(p, \mu)$
5:          $(f_1, .., f_j) \leftarrow execute(p')$
6:          add $(f_1, .., f_j)$ to $F_{p'}$
7:          $score \leftarrow oracle(F_{p'})$
8:          replace $\mu$ with $(\mu, score)$ in $\mathcal{M}_p$
9:      end for
10:     $openSet \leftarrow openSet \cup \mathcal{M}_p$
11:     $\mu_{best} \leftarrow maxByScore(openSet)$
12:     if $|openSet| \geqslant maxOpenSize$ then
13:         reduce size of $openSet$ to $bestOpenSize$
14:     end if
15:     return $\mu_{best}$
16: end function
```

Ant Colony Optimization (ACO)

ACO [48] is an iterative algorithm to find an approximate solution to a combinatorial optimization problem. The intuition of the algorithm, which gave it its name, is that a set of ants traverse the tree while leaving pheromones on edges between components. Pheromones are numeric weights that are obtained by evaluating partial solutions and that evaporate over time. Pheromones encode how close already explored paths are to its goal. Based on the pheromones from previous iterations, ants prioritize paths through the solution space in such a way that the ants steer toward a solution with high pheromone values. This process iteratively improves the explored solutions until meeting some stopping criterion.

We adapted ACO to iteratively explore paths through the mutation tree using a set of ants that independently select the next mutation. During an iteration, each ant traverses the mutation tree and picks the next mutation based on the probability distribution described by *prob* for all outgoing edges:

$$prob(\mu) = \frac{\mathcal{T}(\mu) \cdot \mathcal{L}(\mu)}{\sum_{\mu' \in outgoing(p)} \mathcal{T}(\mu') \cdot \mathcal{L}(\mu')} \tag{3.1}$$

The map $\mathcal{T}(\mu)$ yields a default pheromone value for all not yet performed mutations or it yields the current pheromone for the already explored mutations. In other words, the algorithm initially assigns equal probabilities to all the mutations, and then focuses more and more on promising mutations. The map $\mathcal{L}$ is the prediction table that assigns to each mutation its expected effectiveness in changing the performance. The algorithm updates pheromones values τ after all ants have completed a sequence of nb_{mut} mutations:

$$\tau_{new} = \begin{cases} (1-\varphi) * \tau_{old} + \tau_{old} * score & \text{if on best path} \\ (1-\varphi) * \tau_{old} & \text{otherwise} \end{cases} \tag{3.2}$$

The algorithm reduces each pheromone by a constant factor φ, i.e., the pheromones evaporate. Furthermore, the pheromones of all mutations involved in the best path found so far are increased proportionally to the score returned by the oracle. Moreover, the algorithm bounds pheromone values in such a way that no mutation ever becomes impossible, and that the search continues to explore new mutations even after finding a promising path [131].

Algorithm 6 shows in detail our ACO based mutation selection strategy. The algorithm uses an ε-greedy strategy combined with random proportional strategy to select the next mutation to explore. The algorithm starts by obtaining the list of mutations for the program p, and then it draws a random number (line 3). If the number does not exceed a probability of 25% it then returns a random mutation, favoring exploration of the mutation tree. Otherwise the algorithm proceeds by using a random proportional strategy (line 6-14) described in Equation 3.1.

This strategy first draw a random probability, it then traverses the list of mutations until it finds a mutation with probability that is greater or equal to the random probability. This selection strategy favors mutations that have higher probability, i. e. the mutations with the highest score, but it still allows exploration of the mutation tree.

Algorithm 6 ACO based mutation selection

```
 1: function pickMutation($p \in \mathcal{P}, \mu \in \mathcal{M}$)
 2:     $\mathcal{M}_p \leftarrow mutations(p)$
 3:     $rnd_\varepsilon \leftarrow random()$
 4:     if $rnd_\varepsilon \leqslant 0.25$ then return $pickAtRandom(\mathcal{M}_p)$
 5:     else
 6:         $prob_{select} \leftarrow random()$
 7:         $prob_{sum} \leftarrow 0.0$
 8:         for each $\mu \in \mathcal{M}_p$ do
 9:             if $prob_{sum} \geqslant prob_{select}$ then return $\mu$
10:             else
11:                 $prob_{sum} \leftarrow prob_{sum} + prob(\mu)$
12:             end if
13:         end for
14:         return $pickAtRandom(\mathcal{M}_p)$
15:     end if
16: end function
```

3.2.7 *Feedback and Performance Oracle*

Which of the synthesized programs is the best is determined by the performance oracle (see Definition 12) and depends on the kind of performance problem to search. Instantiating PerfSyn for a specific kind of bottleneck requires to define a feedback function that evaluates to a higher score when the program is closer to reach the goal of exposing a bottleneck. The following presents two feedback functions targeted at specific kinds of bottlenecks.

Exposing Changes in Relative Performance

The first application of our generic framework detects performance bottlenecks fixed or introduced into another functionally equivalent implementation. Prior work to detect performance regressions [31, 50, 153] requires inputs that exercise the analyzed code. In contrast, we focus on how to automatically create inputs that expose performance changes. To use PerfSyn to find bottlenecks due to changes in relative performance, the developer provides a second, assumed to be slower, implementation of the analyzed code, e.g., an earlier version of the same class. We call this alternative implementation the *reference implementation*.

EXECUTION FEEDBACK The feedback for a path from the root program p_0 to a program p_k through the mutation tree is a sequence $F = [(c_0, c_0^{ref}), \ldots, (c_k, c_k^{ref})]$ where each pair (c_j, c_j^{ref}) contains the execution cost for the implementation under analysis and the reference implementation, respectively. During both executions, the approach gathers feedback about the execution cost. Because naively measuring the execution cost as wallclock time is inaccurate for short methods we instead measure the number of evaluated conditional checks [110, 154].

PERFORMANCE ORACLE The oracle computes the score of a program p_j as the difference in execution cost of the implementation under analysis and the reference implementation using the formula:

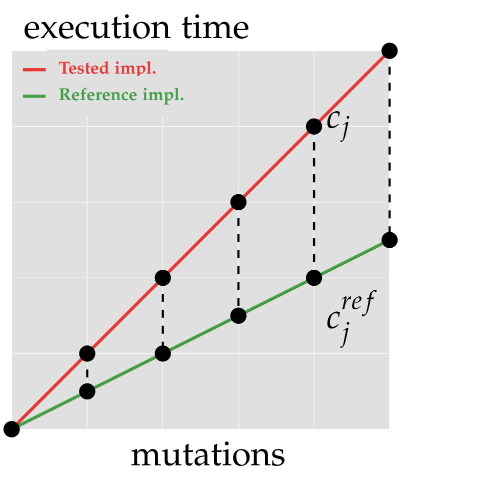
$$score(F) = |c_j^{ref} - c_j| \tag{3.3}$$

That is, the oracle steers the search toward paths that maximize the difference in execution cost between the two versions of the code. This formulation of the score allows to find paths in the mutation tree where the reference implementation is faster (i. e., the new implementation introduces a bottleneck) and where the reference implementation is slower (i. e., the new implementation fixes a bottleneck).

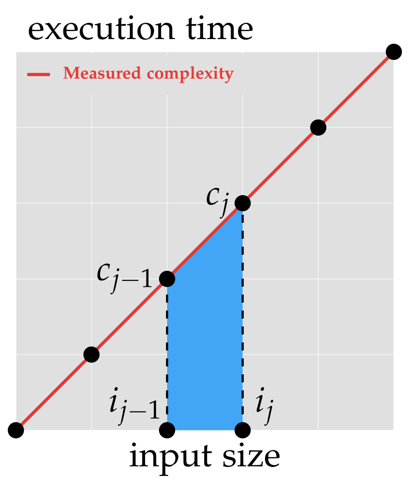
Figure 3.2a depicts graphically the formulation for $score(F)$ where the implementation under test contains a bottleneck, i. e. the reference implementation is faster. Each point in the curve represent a measurement, and the dashed black lines length represent how the score evolves when multiple mutations are applied.

An alternative formulation to Equation 3.4 is to consider the two cases in separated formulas. For example, PerfSyn could search only for programs where the reference implementation is slower using the formula:

$$score(F) = max(0, c_j^{ref} - c_j) \tag{3.4}$$



(a) The dashed black line represents the score for the program p_j after applying j mutations.



(b) The highlighted area below the curve represents one of the trapezoid used to calculate $score(F)$.

Figure 3.2: Representation of the calculation for $score(F)$ for both performance oracles described in Section 3.2.7

Exposing Unexpected Asymptotic Complexity

The second application of the PerfSyn framework focuses on *unexpected asymptotic complexity*, where an implementation requires more resources to handle increasingly larger inputs than expected. Prior work has studied how to detect such problems under the assumption that the inputs that expose the problem are available [39, 59, 157]. In contrast, we focus on how to automatically synthesize inputs that expose an unexpected asymptotic complexity. To use PerfSyn to find such a problem, the developer specifies the expected complexity class, e.g., $\mathcal{O}(1)$, $\mathcal{O}(n)$, $\mathcal{O}(n^2)$, of the method under test.

EXECUTION FEEDBACK The feedback for a path through the mutation tree is a sequence $F = [(i_0, c_0), \ldots, (i_k, c_k)]$, where each pair (i_j, c_j) represents a measurement of the input size and the execution cost of a program p_j in the path. In general, the problem of determining this size is impossible because the notion of "input size" is program-specific and typically not explicitly specified. In this dissertation, we measure the size of the input based on the number of first read memory locations [39, 40], and we measure the execution cost as the number of conditional checks.

PERFORMANCE ORACLE To compute a score that indicates how close a path is to expose a bottleneck the oracle performs polynomial curve fitting using the measured values in F. The curve represents how the execution cost varies depending on the input size. To determine the best-fitting curve, PerfSyn a naive curve matching algorithm.

First the algorithm tries to fit the data points to curves of different degrees and it selects the curve that yields the smallest normalized root mean square error against the selected polynomial. This naive algorithm is effective in practice and it does not require complex numerical techniques that are out of the scope of this dissertation. However the naive algorithm due to its limitations is not able to automatically match polynomials for logarithmic complexities like $\mathcal{O}(log(n))$ and the definition of input size used wrongly estimates the input size for such functions with sub-linear behavior [39].

Second, the oracle computes the score as the area under the curve using the trapezoidal numerical integration over the data points in F:

$$score(F) = \frac{1}{2} \cdot \sum_{(i_j, c_j) \in F} (i_j - i_{j-1}) \cdot (c_j + c_{j-1}) \tag{3.5}$$

The intuition behind the score is that, the larger the area under the curve the more the actual performance deviates from the expected performance because the work per input increases. Figure 3.2b depicts graphically how the area under the curve is calculated. Each black dot on the red linear curve is a measurement. The area in the figure highlighted in blue shows how one portion of the summation is calculated.

3.2.8 *Reporting and Ranking of Methods*

PerfSyn reports to the developer a ranked list of methods that may contain a performance bottleneck, the ranked list reports the most severe bottlenecks first.

The oracle score alone does not provide an intuitive characterization for a developer of the severity of the bottleneck to identify which methods should be inspected first. To this end PerfSyn uses a different ranking system for each of the oracles described in Section 3.2.7:

- *Changes in Relative Performance.* PerfSyn first filters a method if the score of the best bottleneck exposing program p_{best} for the method does not exceed a t_{rel} threshold. PerfSyn then sorts the remaining methods by their $score(p_{best})$, and it then prioritizes methods for which the new implementation is slower than the reference implementation.
- *Unexpected Asymptotic Complexity.* PerfSyn first filters a method if the error of the fitted curve for p_{best} is above a threshold t_{err} and for which $|F_{best}| < t_{size}$. I. e. PerfSyn filters methods that do not have enough data-points to reliably approximate the polynomial curve. The remaining methods are then reported to developer by prioritizing methods with an higher empirically complexity for p_{best}.

3.3 IMPLEMENTATION

We implemented PerfSyn as a tool to analyze Java classes. To gather feedback about program executions, we instrument the analyzed code using the Java bytecode manipulation library ASM [22]. To collect information about the program structure, the class hierarchy, and type information of the initial program, we use the Soot framework [140]. We developed our own framework to make use of Soot information (e. g., querying data-type relationships). The implementation of curve fitting and other mathematical computations builds upon the Apache-Commons Math library [1] . The oracle that exposes changes in relative performance is implemented by executing a synthesized program with two version of the class under test, each running in a separate Java Virtual Machine (JVM). To avoid issues of incompatible APIs across versions, PerfSyn ignores mutations that involve methods that do not exist in both versions or methods that have different signatures.

PerfSyn's implementation is split across two main software components: (i) a program synthesis component that traverses the mutation tree and that generates bottleneck exposing programs, and (ii) a profiling component that executes programs and that collects the execution feedback. The tree search process communicates with one or multiple profiling process using a memory mapped file(s) to minimize the communication overhead. Reduction of the communication overhead is key factor because PerfSyn must spend its time budget searching for bottlenecks.

Search

The input to implementation is the source-code and executable Jar archives of the m_{ut} and its dependencies. The implementation of the search performs four main tasks:

1. It first parses the source-code to extract Java data-types and their generics type information using the Eclipse JDT framework [3]. The extracted information contains the information required by PerfSyn to instantiate the data-types during program generation. For example, PerfSyn requires to know the type-hierarchy to handle sub-classing while searching for compatible types to concretize an abstract variable. Java utility libraries make extensive use of generic types and it is important to handle generic types correctly during program generation. For example, binding the `Object` type to each generic parameter is not profitable for PerfSyn because this causes to make all type instances to be compatible for a method argument. The result is an unnecessary increase in size of the search space.

2. In the second phase the implementation builds an initial root program p_0 concretizing an *inject call* mutation operator for the m_{ut} using the procedure described in Section 3.2.3. The implementation queries the type-hierarchy and searches for compatible data-types to instantiate all the arguments to the m_{ut}.
3. Once p_0 is created the search starts and continues until the specified time budget expires. The implementation runs the profiling instances in parallel and collects the execution feedback using a simple synchronization protocol.
4. The explored mutation tree is saved into a JSON file and then parsed with a Python script that reports the ranked methods to the developer as described in Section 3.2.8.

Profiling

The profiler takes as an input the Jar of the dependencies and which performance metric to measure for all the executed methods. The profiling process is agnostic of the specific m_{ut} that is executing. Using a simple API interface the profiling process is provided to a new program p every time that the search requires the execution feedback measurements. Profiling instruments all the loaded classes in the JVM and each newly received program p. The limitation of this byte-code instrumentation approach is that the instrumentation may miss some of the classes. For example, the `java.lang.Object` or `java.lang.String` classes methods cannot reliably instrumented without crashing the running VM.

Table 3.1: Classes and methods used in the evaluation.

Project	Version	Classes (# of methods under test)
Changes in relative performance:		
Ant	1.9.1 vs. 1.9.4	IdentityStack (6), VectorSet (20)
Commons-Collections	3.2.1 vs. 4.1	TreeList (9), ListOrderedMap (14), ListOrderedSet (11)
Commons-Lang	3.4 vs. 3.5	ArrayUtils (51), CharSetUtils (5), StringUtils (1)
Unexpected asymptotic complexity:		
Lucene	5.5.4	Operations (17)
Commons-Collections	3.2.1 4.1	BoundedFifoBuffer (8) CompositeMap (1)
Commons-Math	3.6	EnumeratedIntegerDistribution (1)
Guava	8ea0f20 6cec4d2	Sets (1) NavigableMap (1)
SunFlow	0.07.2	SunFlowAPI (1)

3.4 EVALUATION

In this section we verify the claim that PerfSyn synthesizes bottleneck triggering programs. We evaluate PerfSyn using the oracles presented in Section 3.2.7. We start by describing our experimental setup and we evaluate PerfSyn with a set of popular Java libraries. We show that PerfSyn is able to trigger known and previously unknown bottlenecks for multiple methods in popular Java libraries. In Section 3.4.2 we describe the cases for which PerfSyn fails to trigger a bottleneck.

3.4.1 *Experimental Setup*

BENCHMARKS We apply PerfSyn to 147 methods from 15 classes in seven popular Java projects [133]. Table 3.2 shows the list of methods we selected for the evaluation. We analyze 17 methods with bottlenecks known from previous work [103] and from publicly available bug reports, all of which have been confirmed by the developers. These methods allow us to evaluate whether PerfSyn detects developer-confirmed problems. The remaining 130 methods used are to evaluate how many additional bottlenecks PerfSyn reports and whether it detects any previously unknown problems.

For the oracle that targets to discover bottlenecks due to changes in relative performance, we compare different versions of the same class. We select the latest two versions of the same class at the of evaluation time. One can imagine to apply PerfSyn for any two pair of versions of a method, or for pairs of commits in a version control system. For the methods with known and fixed bottlenecks, we use the version before and after the fix.

For the oracle that targets to discover unexpected asymptotic complexity, we analyze methods that explicitly annotate the expected complexity in their documentation. To selected these methods we first programmatically parsed the documentation for keywords like `complexity`, `O(*)`, `linear`, `quadratic`, etc. We then manually inspected methods that contained an annotation about the expected complexity and selected the methods from which the complexity was explicitly stated in the documentation.

INSPECTING BOTTLENECKS For each method, PerfSyn yields a set of synthesized programs and their performance score. In practice, we expect developers to inspect only methods with an high score and methods that ranked hight based on the criteria listed in Section 3.2.8. In our evaluation, we set the meta-parameters described in Section 3.2.8 as follows:

- *Changes in Relative Performance.* The difference in performance reported by PerfSyn is larger than $t_{rel} \geqslant 1.2x$.
- *Unexpected Asymptotic Complexity.* The error of the fitted curve is below $t_{err} \leqslant$ 5%, the number of data-points is $|F_{best}| \geqslant 4$, and the fitted complexity exceeds the expected complexity.

In our evaluation we report only methods that pass these two criteria.

PARAMETERS AND HARDWARE We give a 5-minute time budget per method under test, and we limit the maximum exploration depth of the mutation tree $maxMuts = 32$. For A*, we set the maximum number of nodes in the open set to $maxOpenSize = 8{,}192$ when the memory is full, and all but the best $bestOpenSize = 16$ nodes are removed. For ACO, we use 16 ants.

All experiments are done on a 42-core machine with two 2.2 GHz Intel Xeon processors E5-2699, 512GB memory, running 64-bit Ubuntu 16.04.1 LTS with GNU/Linux 4.4, Java 1.6.0_27 using OpenJDK 1.8.0.111, with 256GB of memory assigned to the VM.

Table 3.2: Bottlenecks detected by PerfSyn (Muts=number of mutations, Perf=Outcome of performance oracle in bytecode conditionals (bc), and the execution time in nano-seconds (ns)). The highlighted rows represent previously unknown bottlenecks.

		Best program					
		A*			**ACO**		
ID	Method	Muts	Performance (bc)	(ns)	Muts	Performance (bc)	(ns)
Changes in relative performance:							
1	ArrayUtils.removeAll	1	-1.85x	-1.01x	3	-1.85x	-1.04x
2	ArrayUtils.removeElements	31	+1.68x	+1.59x	29	+1.54x	+1.24x
3	ArrayUtils.indexOf	1	+1.23x	+1.23x	1	+1.23x	+1.37x
4	ArrayUtils.isNotEmpty	1	-1.67x	-	1	-1.67x	-
5	ArrayUtils.isSameLength	1	+1.40x	-	1	+1.40x	-
6	CharSetUtils.squeeze	14	+1.75x	+1.24x	13	+1.25x	-
7	StringUtils.getLevenDist	3	+79.95x	+8.32x	1	+50.60x	+5.16x
8	IdentityStack.containsAll	7	-40.91x	-5.29x	14	-41.03x	-6.92x
9	IdentityStack.removeAll	12	-6360.0x	-772.0x	8	-57.05x	-11.35x
10	IdentityStack.retainAll	12	-6360.0x	-697.0x	14	-3.24x	-3.96x
11	ListOrderedMap.remove	7	-1.44x	-1.16x	6	+2.37x	+1.14x
12	ListOrderedSet.addAll	12	+1.37x	+1.25x	14	+1.37x	+1.18x
13	ListOrderedSet.addAllAtIndex	1	-1.42x	-	3	-1.39x	-1.40x
14	ListOrderedSet.toArray	5	+2.60x	-	2	+2.60x	-
15	ListOrderedSet.remove	-	-	-	23	+1.75x	+1.11x
16	VectorSet.addAll	9	-1.22x	-2.48x	12	-1.26x	-1.24x
17	VectorSet.clone	5	+1.22x	-	2	+1.22x	-
18	TreeList.addAll	1	+6.18x	+1.16x	10	+7.48x	+4.22x
19	TreeList.addAllAtIndex	-	-	-	12	+1.51x	+1.35x
Unexpected asymptotic complexity:							
20	EnumIntDist.probability	32	$O(n)$	-	7	$O(n)$	-
21	NavigableMap.isEmpty	7	$O(n)$	-	6	$O(n)$	-
22	Sets.powerSet	10	$O(n)$	-	4	$O(n)$	-

3.4.2 *Effectiveness in Finding Bottlenecks*

For 22 out of the 147 methods, PerfSyn synthesizes a program that exposes a bottleneck and that we inspect based on the criteria given in Section 3.4.1. The table which shows for each method the best program synthesized with A* and ACO, along with the number of mutations applied to reach this program and the performance effect exposed by the program. For the changes in relative performance, we show speedups and slowdowns with a plus and minus, respectively. For example, for bottleneck 7, the A* search synthesizes a program by applying three mutations, and this program shows the method to be faster than in the previous version. Bottleneck 7 is previously known and the code for the last version (version 3.5 of Apache-Lang) contains a fix to the performance problem. In other words PerfSyn is able to synthesize a program that shows that the new version is faster than the previous. This means, that at least for the synthesized program, that the optimization was beneficial. For unexpected asymptotic complexities, the table reports the complexity class that PerfSyn finds. For all bottlenecks in the table, the methods were expected to be $\mathcal{O}(1)$, but turn out to have linear complexity.

The performance differences reported in Table 3.2 are based on the number of evaluated conditions (bc) and execution time (ns) on the profiling machine. To validate this proxy metric, we execute the synthesized programs and measure their wall-clock execution time, reported in column *Performance (ns)*. To confirm the performance differences for bottlenecks ID 1 to ID 19, we repeatedly measure the execution time (32 times) and check whether the differences are statistically significant (95% confidence) using the JMH framework [55]. To confirm bottlenecks ID 20 to ID 22, we manually inspect the code and verify that the computational complexity reflects what PerfSyn reports. The validation confirms that in general the proxy metric reliably reflects the performance behavior of the synthesized tests when executed on the machine used for the experiments, and that the manually checked code complexity reflects what PerfSyn reports. However for short methods (e.g., like for bottleneck ID 14 and bottleneck ID 17) the proxy metric fails to accurately approximate the performance behavior of the method because the execution time differences for the methods are not significant.

Analysis of Reported Bottlenecks

Eight of the 22 methods reveal a previously unknown performance property and the remaining bottlenecks other previously known bottlenecks. Table 3.2 highlights these bottleneck in grey. Bottleneck ID 1 is a previously unknown slowdown in a method without a previously known problem. For the other highlighted bottlenecks, PerfSyn triggers an unexpected slowdown introduced by a change supposed to optimize the code. For example, for bottleneck ID 16, PerfSyn shows that applying a change supposed to be an optimization is not beneficial for a usage scenario. These cases show that our approach helps developers to find unexpected effects of modifying code. The remaining reports confirm that optimizations applied by developers are indeed beneficial. This kind of report is useful for a developer who wants to verify his/her performance assumptions after a code change.

Analysis of Non Reported Bottlenecks

125 methods out of the 147 are not reported by PerfSyn as containing a bottleneck. Among these methods 19 methods were not reported because PerfSyn failed to generate an initial program p_0 to start the program synthesis. The common cause for PerfSyn to fail creating an initial program p_0 is the lack of semantic knowledge about the method arguments. For example, methods of the class `VectorSet` from Ant expect that the index argument must be already in range of the underlying collection. This issue araises because the approach is not aware of the semantic relationship among method parameters and because primitive values are randomly selected from a pool of existing values. We discuss this limitation in Section 3.6 and how to alleviate this issue in practice.

The remaining 106 methods are not reported by PerfSyn because they did not pass the filtering criteria. We list some common reasons for which PerfSyn does not report a method:

- *Commons-Lang.* The class `ArrayUtils` is a helper class that performs bulk operations on arrays (e. g., adding and removing array elements). We selected from this class methods that perform the same operation but on arrays of different data-types. It is the case that these methods do not contain any bottleneck. Many of these methods are filtered altogether because they all call the same utility methods to perform the operation on the array.
- *Lucene.* None of the methods of the class `Automaton` is reported to have an higher empirical complexity than the expected complexity. We manually inspected the class source-code and we found that none of the methods in the class exceeds its documented computational complexity as reported by PerfSyn.

We manually inspected all the remaining filtered methods, the methods are not listed because they either fall in the two categories above for the respective oracle type, or because they did not contain a bottleneck.

3.4.3 *Examples of Synthesized Programs*

Figure 3.3 shows three programs synthesized by PerfSyn. The highlighted statements are inserted by PerfSyn on top of the initial program.

The program in Listing 3.3b exposes a bottleneck in `ListOrderedMap.remove`. The implementation of the faster version of `remove` performs a containment check before removing the element to save time when the collection is non-empty and when it does not contain the element. A previous version the method did not contain this optimization. For bottleneck ID 11 PerfSyn we synthesize two different programs, one for each search as indicated in Table 3.2. PerfSyn discovers a path in the mutation tree that adds multiple elements to the map and that passes an argument to the method. For one synthesized program the argument to `remove` is contained in the collection, and for the other program the argument is not part of the collection.

For the program shown in Listing 3.3c, after a few iterations, PerfSyn prioritizes a path through the mutation tree that inserts multiple similar statements. As a result, the execution time of the method under test increases, exposing the linear complexity of the method.

The first example in Listing 3.3a, shows the program synthesized for bottleneck 1. The bottleneck is caused by two unnecessary calls in `ArrayUtils.remove` that copy and sort the second parameter of the method. These calls are only necessary when the second argument is a non-empty array. PerfSyn detects this case and synthesizes a program that triggers the problem, showing that the bottleneck is triggered independently of the content of the first argument.

For bottleneck 9, PerfSyn synthesizes a program that removes from an empty collection. The older version of `IdentityStack.remove` does not execute any code because the collection is empty. However, the newer version of the method first creates a useless instance of a `HashSet` and then needlessly adds to this newly created set multiple elements supposed to be removed from the empty collection.

Overall, these examples show that our approach detects real-world performance bottlenecks in widely used Java classes. By synthesizing a program that demonstrates each problem, developers can easily understand and then fix the bottlenecks, or more easily understand the performance impacts of code changes.

(a) Program that triggers bottleneck ID 1.

```java
public class SynProg {
   public run() {
      String[] v0;
      String v3;
      int v2;
      int[] v1;
      v0 = new String[/*...*/];
      v3 = "str0";
      v0 = Array.add(v0, v3);
      v0 = Array.removeLast(v0);
      ...
      v1 = new int[/*...*/];
      ArrayUtils.removeAll(v0,v1);
   }
}
```

(b) Program that triggers bottleneck ID 11.

```java
public class SynProg {
   public run() {
      ListOrderedMap v0 = new ListOrderedMap();
      v0.put("str0","...");
      v0.put("str1","...");
      ...
      v0.put("strN","...");
      v0.remove("strX");
   }
}
```

(c) Program that triggers bottleneck ID 20.

```java
public class SynProg {
   public run() {
      int[] v0 = new int[/*...*/];
      int v3 = /*...*/;
      Array.add(v0,/*...*/);
      Array.add(v0,/*...*/);
      /*...*/
      Array.add(v0,/*...*/);
      EnumIntDist v1 = new EnumIntDist(v0);
      v1.probability(v3);
   }
}
```

Figure 3.3: PerfSyn synthesized programs from Table 3.2. The light blue statements have been inserted by PerfSyn to p_0 during the program synthesis.

3.4.4 *Efficiency*

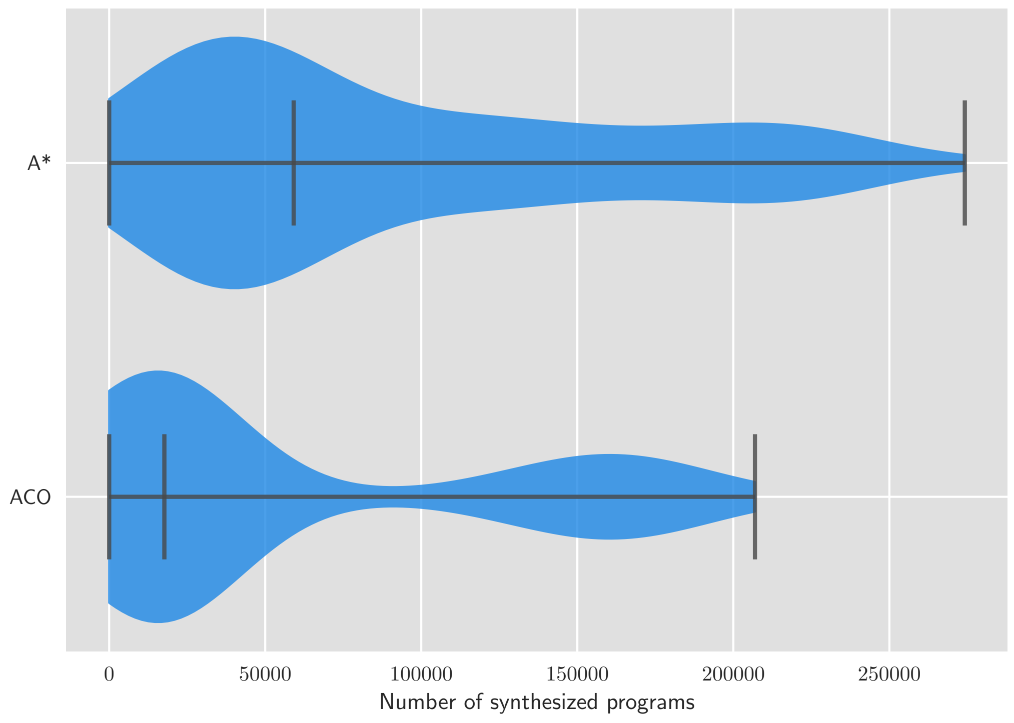
To evaluate the efficiency of PerfSyn we measure the number of explored nodes in the specified budget of time. Figure 3.4 shows the distribution of the number of explored nodes in the mutation tree. The results show that both A* and ACO explore a large number of programs in the given time budget. The relative difference between the two search strategies are mostly due to an implementation detail: ACO probabilistically chooses a single mutation at each step, which sometimes causes ACO to expand longer paths of programs that cause a crash.

These programs take more time to execute because of our handling of exceptions in the profiling VMs (e. g., timeouts to manage infinite loops and recursions).

Figure 3.5 shows the breakdown for each benchmark of the valid generated and executed programs for the two evaluated oracles. A valid generated and executed program is a synthesized program that was correctly executed without throwing an exception, but could be a semantically invalid program (e. g., it does not respect APIs calling protocols). The two graphs confirm our previous hypothesis: ACO generates more failing programs that cause PerfSyn to synthesize, on average, less programs.

Overall, we conclude that both search strategies efficiently explore the mutation tree, which enables them to detect bottlenecks in at most 5 minutes per method.

(a) Unexpected Changes in Relative Performance.



(b) Unexpected Asymptotic Complexity.

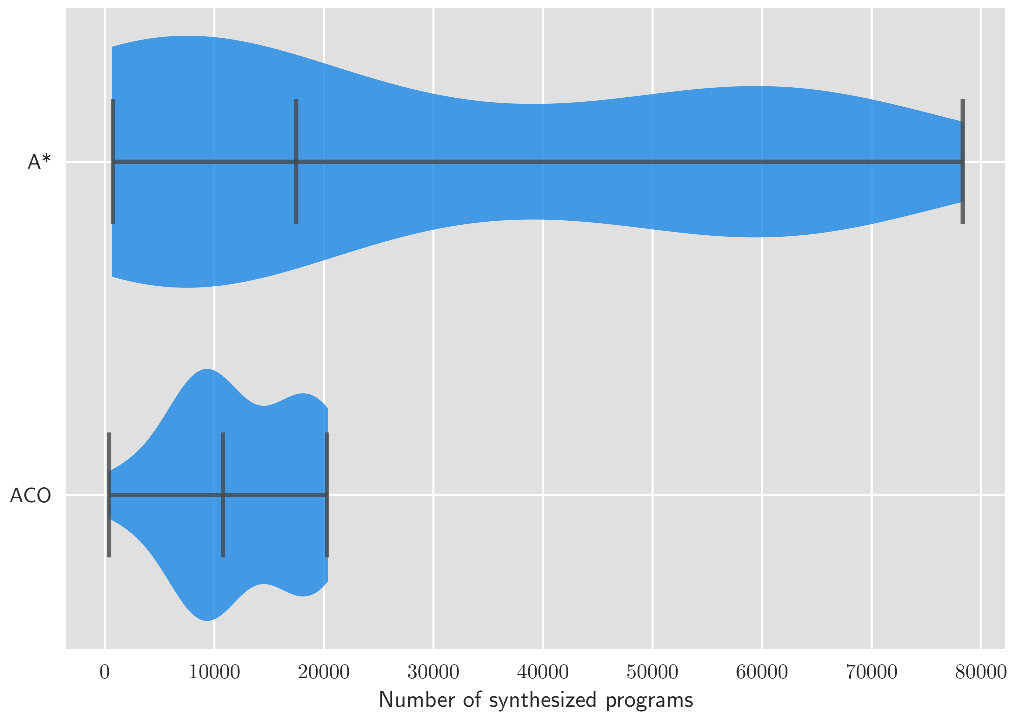
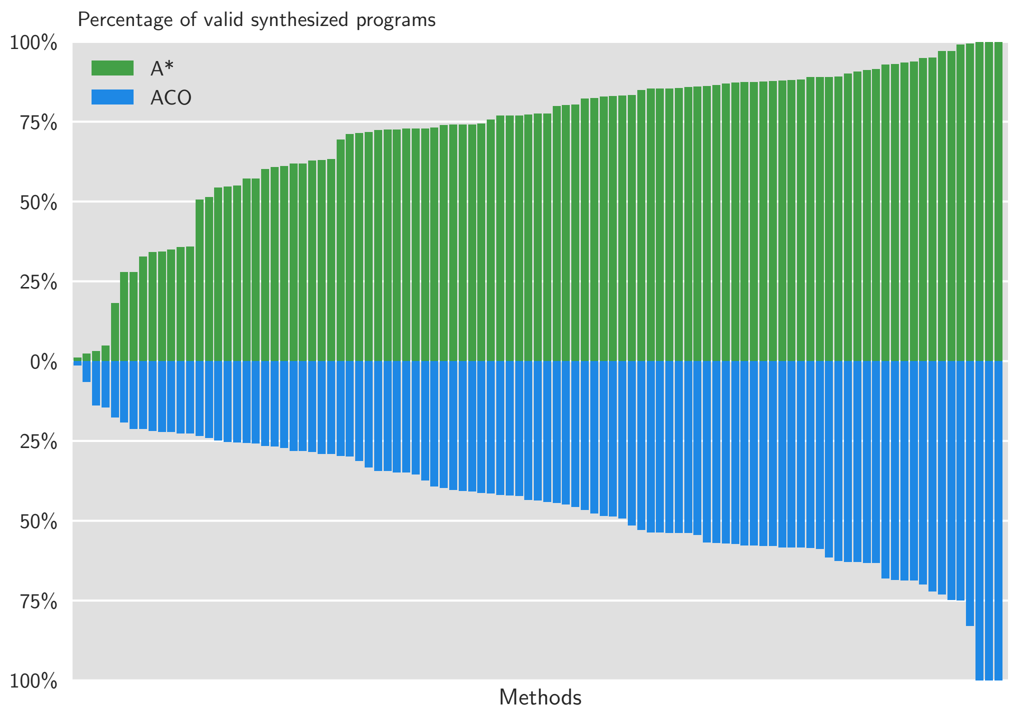


Figure 3.4: Violin plots of the average number of programs synthesized while analyzing a method.

(a) Unexpected Changes in Relative Performance.



(b) Unexpected Asymptotic Complexity.

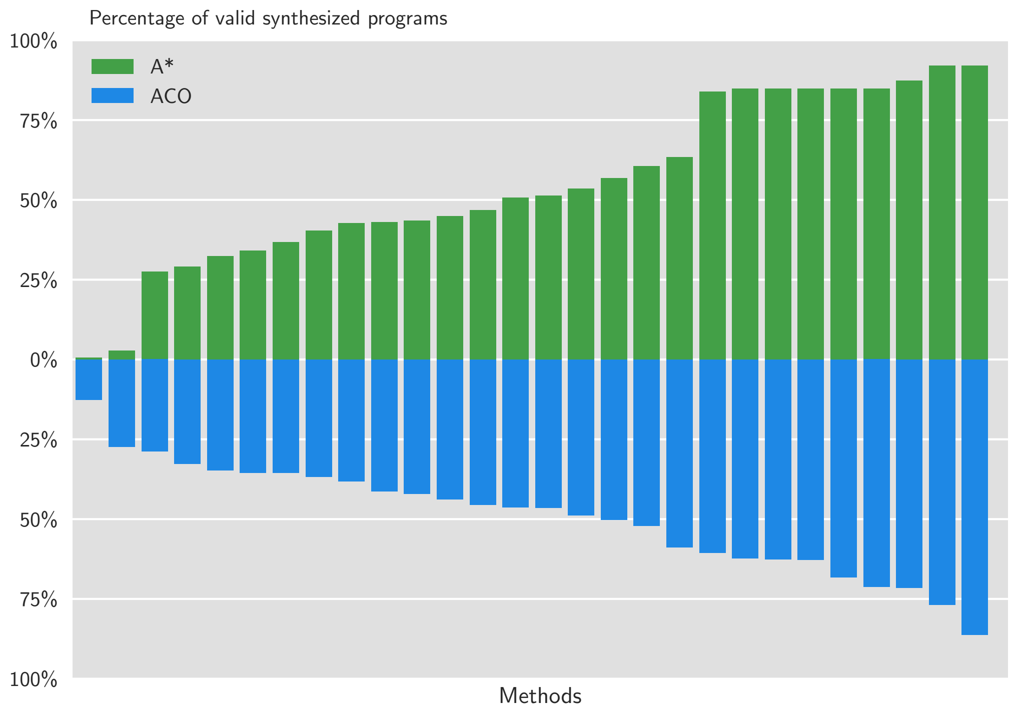


Figure 3.5: Plots showing the percentage of valid synthesized programs for each oracle.

3.5 RELATED WORK

3.5.1 *Test Generation*

Wise [24] steers symbolic execution toward inputs that trigger worst-case complexity. In contrast, PerfSyn uses a blackbox approach. SpeedGun [109] detects performance regressions via test generation, which is a special case of detecting unexpected relative performance. Instead of generating tests at random, PerfSyn uses feedback to steer toward bottleneck-exposing tests.

EventBreak [110] also exploits performance feedback to generate tests, but for UI-level instead of unit-level tests. Dhok et al. [45] propose to generate tests that expose loop inefficiencies. Their work focuses on a particular kind of performance problem and it relies on a manual inspection of generated tests. Travioli [106] dynamically identifies functions that traverse data structures, which can help developers in manually constructing performance tests.

Compared to all of the work above, PerfSyn is the first framework (i) that provides a generic approach to automatically create tests for different kinds of bottlenecks and (ii) to formulate the problem of finding a bottleneck as a graph exploration problem.

Test generation for purposes other than performance has been widely studied. Fuzz testing [56, 58], concolic execution [57], feedback-directed random test generation [105], evolutionary algorithm-based test generation [51], and combinations of static and dynamic analysis [136] share the idea of using feedback from past executions to steer the generation of future tests.

These and other [25] approaches steer toward high code coverage. Instead, PerfSyn exposes bottlenecks, which typically do not require covering all statements or branches but repeatedly trigger specific statements and branches. Approaches that use genetic algorithms [51] may not be directly usable for performance because it is not trivial to understand the performance effects of their program mutations (e. g., mutations that combine two different programs).

3.5.2 *Dynamic Analysis*

PerfSyn relates to dynamically analyzing the asymptotic complexity and scalability of software. Goldsmith et al. [59] pioneered the idea of empirically estimating the computational complexity of a program. Their work requires a developer to specify the input size of the analyzed code – a problem addressed more recently [39, 157]. PerfSyn builds on one of these approaches [39] to estimate the input size for determining unexpected asymptotic complexities. A limitation shared by all of the above is to require inputs that may expose complexity issues; PerfSyn addresses this limitation by automatically generating the inputs.

Instead of analyzing scalability w.r.t. input size, Calotoiu et al.[27] propose a dynamic analysis to model the scalability w.r.t. the number of processors that execute a program. Our work differs by targeting performance problems that are independent of the underlying hardware.

Beyond asymptotic complexity issues, dynamic analysis is widely used to detect various other kinds of bottlenecks. General purpose profilers highlight functions where most CPU time is spent [61] or performance-critical paths [114], extract a model that summarizes performance properties of a program [23], or highlight code locations that, if optimized, will speed up the execution [44].

Other profilers target particular classes of performance problems, such as JIT-unfriendly code [60, 129], unnecessarily repeated behavior [103], inefficient use of object-oriented language features [93, 101, 145–148, 150], and UI delays [76]. PerfDiff [160] compares the performance of a single implementation in different environments; in contrast, one of our oracles compares the performance of different implementations.

The effectiveness of all these approaches depends in test inputs that trigger interesting behavior. PerfSyn contributes an automated way to generate such test inputs.

To help developers understand and localize observed bottlenecks, several dynamic analyses have been proposed. They help understand idle times of servers [11], associate bottlenecks with particular configuration options [14], and reveal impact relations between code locations [155]. Some approaches combine runtime data from multiple users to pinpoint the root cause of bottlenecks [65, 127]. SyncProf [154] analyzes waits-for relationships between critical sections to localize synchronization-related bottlenecks. A recent survey [71] summarizes work on performance anomalies.

3.5.3 *Static Analysis*

Several static analyses have been proposed that detect performance bottlenecks, e. g., redundant traversals of data structures [104] and other unnecessary loop iterations [102]. Dufour et al. [49] propose a technique that combines static and dynamic analysis to find excessive uses of temporal objects.

3.5.4 *Studies of Performance Issues*

Studies of performance issues in real-world code show that developers spend non-negligible amounts of time dealing with bottlenecks [75, 88, 120]. One of them [88] reports that many bottlenecks only manifest with specific input data and with input data of specific sizes, a problem addressed here. A recent study over large body of open-source projects shows that performance tests is still lacking in the large [85].

3.6 LIMITATIONS

STRUCTURED INPUT DATA. In our evaluation we focused on libraries that build generic collections (e. g., arrays, trees, or lists). However PerfSyn lacks support targeted specifically to programs (e. g., parsers) that use domain-specific data-structures governed by a language grammar and that also carry semantic information (e. g., HTML and XML file formats). To correctly synthesize programs that use these data-structures PerfSyn can implement tailored mutation operators for the data-types, but at the cost of loosing generality. We plan to study and to evaluate an extension of PerfSyn with specific support for these data-structures in future work.

EFFECTIVENESS OF INITIAL PROGRAM. The effectiveness of PerfSyn in reporting performance bottlenecks is influenced by the initial program p_0 because the program may be far from a bottleneck triggering state. Therefore finding a sequence of mutations that yields a better solution in the time budget depends on p_0. Our current prototype implementation generates a root program p_0 using the same procedures used to generate a sequence of statements to initialize an abstract variable (see Section 3.2.3). However this can lead to sub-optimal results because only a fraction of the possible initial programs p_0 that exercise m_{ut} is explored. An optimal solution to the problem of finding a "good" initial program is unfeasible to find in practice because of the requirement to explored a large number of programs.

Other approaches [53] combine a randomized search with systematic exploration of suitable primitive values to generate an initial program to start the search.

WHITE-BOX APPROACH. PerfSyn is efficient in finding performance bottlenecks because it does not execute expensive program analyses. However PerfSyn may benefit from a white-box approach, i. e., an approach that analyzes a program's source-code and that uses facts about the program under analysis to drive the search for bottlenecks. For example, PerfSyn could use information about the predicates that condition a method's control flow branches to drive the program generation. PerfSyn could generate programs that setup the state to evaluate these branches towards a specific, possibly bottleneck triggering, method path. Currently PerfSyn is not aware of path predicates and finding a predicate-triggering state may be difficult because it may require long sequences of mutations. Consequently PerfSyn may miss performance bottlenecks that are triggered when a method's control flow depends on a specific argument's value (e. g., specific enumerative type value).

3.7 SUMMARY

In this chapter we presented the first general framework for synthesizing inputs to expose performance bottlenecks in a given method under test. This new approach addresses the understudied problem of creating bottleneck-exposing inputs and it expands the application of existing profiling techniques, helping software developers in their efforts to reduce performance bottlenecks. The approach synthesizes a program that calls the method under test, and during the synthesis process systematically increase the amount of work done by the method. A key insight enabling PerfSyn is that the problem of finding a sequence of mutations that leads to a bottleneck-exposing program can be effectively and efficiently addressed as an optimization problem. This type of problem can be solved using state-of-the-art algorithms. Based on this insight, the approach uses feedback from executions of the synthesized programs to steer the synthesis towards the most promising sequences of mutations. The presented approach is applicable to different kinds of performance bottlenecks, including unexpected asymptotic complexity and unexpected performance relative to another implementation. We evaluate PerfSyn by applying it to widely used Java classes, in which PerfSyn reveals 22 bottlenecks in only five minutes of profiling time per bottleneck.

4 MINING TEST INPUTS

The main contribution of this chapter is to tackle a major limitation that affects PerfSyn and state-of-the-art automatic test generators [51, 52, 105]. I. e. PerfSyn is effective in synthesizing bottlenecks exposing programs for generic classes, such as collections, but it is ineffective in generating tests for domain-specific software.

PerfSyn fails to synthesize programs for domain specific classes because the setup code for the inputs to the method under test creates an invalid state for the method (e. g., it always crashes the method's execution), or because the input to the method does not allow repeated executions of the same portion of the code (e. g., there is no crash but the method does not execute relevant program paths). To hit a performance bottleneck, the input to the method may be required to have specific properties. For example, a method may require an in-memory data-structure that the method traverses to have a specific shape and size, achievable only towards a specific sequence of API calls (e. g., a graph free of cycles). Reaching a performance triggering state may be easier for generic classes because their limited set of operations is shared among a common interface (e. g., `java.util.Collection`) and each operation has always the same semantic. However, creating a performance triggering program for a method that requires well-formed inputs (e. g., inputs in the language of a grammar) is not trivial. In this case the number of operations that can be applied to the input can be large and each operation may have strong, type-specific, semantics attached.

Unfortunately, PerfSyn is not aware of these requirements and the program synthesis is limited by design because of its *black-boxing* nature. A possible solution to this limitation is to manually provide type specific operations but at a loss of generality of the approach, or to use a *white-boxing* approach but at the cost of executing prohibitively expensive and limited program analyses [24, 57, 121]. Yet, even if the input data-structure respects all the preconditions required by a method, the method may still require that the fields values of the data-structure to be valid domain-specific identifiers (e. g., HTML tags in a web-page, or JSON objects delimiters). Also, in this case, PerfSyn fails to synthesize valid programs because it assigns primitive values randomly or by extracting them from a pool of commonly used constant values.

These limitations are not unique to PerfSyn but they are also shared with automatic test generators, a powerful and widely used approach to create inputs for exercising a software under test with minimal human effort. Compared to PerfSyn a test generator focuses on maximizing the execution coverage of a method to reveal correctness bugs, while PerfSyn maximizes the time spent in a piece of the code to expose performance bottlenecks. Existing test-generators use a wide range of techniques, ranging from feedback-directed random test generation [12, 105, 108], over search-based approaches [34, 51], to symbolic reasoning-based test generators [25, 57, 121, 136]. Despite all successes, test generation still suffers from non-trivial limitations. For example, a study reports that only 55.7% of bugs in a well known collection of existing faults are revealed by the test suites generated by three state-of-the-art test generators [124].

A cause of the limited success of revealing bugs is that test generators often fail to cover a buggy statement because the inputs provided in the test do not enable the code to bypass sanity checks that reject invalid inputs, a limitation also shared with PerfSyn. In particular, creating bug-revealing inputs often requires suitable strings, but creating such strings requires domain knowledge about the software under test, which existing test generators and PerfSyn do not have. For example, consider testing a class responsible for parsing SQL statements. Testing the class with a randomly generated string is highly unlikely to reach deeply into the code because the invalid input is discarded quickly due to a parse error.

In this chapter we attack the limitation described above, namely to find appropriate literal values for a domain-specific context. We study this problem in the context of a test generator and we compare against a state-of-the-art test generator [105] using a well-known suite of existing faults that provide a clear metric for success [124]. In this chapter we present TestMiner, an approach to predict input values suitable for a given method under test:

- The first contribution of this chapter is an information retrieval technique for test inputs. We are the first to exploit the knowledge hidden in large amounts of existing code to address the problem of finding suitable input values for testing.
- The second contribution of this chapter is to devise a scalable and efficient prediction technique of domain-specific values. We show how to integrate the technique into an existing state-of-the-art test generator [105]. One of the major advantages of TestMiner is that it does not require any pre-processing of data and it can be used out-of-the-box.

- The last contribution is an empirical evaluation of TestMiner. We show that our novel approach substantially increases the branch coverage of Randoop [105] for benchmark classes that are usually difficult to handle by a test generator.

```
package org.pkg;

class SqlParser {
   /* Transforms provided string into an
    * abstract syntax tree.
    * Quickly rejects invalid strings.
    */
   Tree parse(String stmt) {
      SqlLexer lexer = new SqlBaseLexer(/*...*/);
      SqlBaseParser parser = new SqlBaseParser(/*...*/);
      /* Other setup code ... */
      Tree result = null;
      try {
         /* Try parsing mode (i) */
         tree = parser.apply(lexer, stmt);
      } catch (ParseException ex) {
         /* Failed with parsing mode (i), trying mode (ii) */
         lexer.reset();
         parser.reset();
         tree = parseFunction.apply(lexer, stmt, /*...*/);
      } catch (Exception e) {
         throw new /*...*/;
      }
      return tree;
}

class SqlsOptimizer {
   /* Transforms the given tree
    * into another tree.
    * Quickly rejects invalid values,
    * e.g., null.
    */
   Tree optimizeRanges(Tree sqlTree) {/*...*/}
}
```

Listing 4.1: Example of a class under test that requires a set of suitable string values for effective testing.

4.1 TESTMINER BY ILLUSTRATION

As an example, consider testing a class responsible for parsing SQL statements, such as the `SqlParser` class shown in Listing 4.1. The example class is a simplified version of a real-world SQL parsing class that can be found in the Defects4J suite [124]. Testing the `parse` method with randomly generated strings is highly likely to not reach deeply into the method's code because invalid inputs are discarded quickly. Moreover, suppose to generate tests for a larger code base where some code requires instances of `Tree`, i. e., values returned by `parse`, such as the `SqlOptimizer` class in the example. Without passing suitable strings to the parser, the test generator cannot create non-trivial `Tree` objects and will also fail to effectively test other parts of the code.

State-of-the-art test generators obtain input data, such as strings, in various ways. First, most generators use randomly generated values or values from a fixed pool which are cheap to obtain but unlikely to match domain-specific data formats. For example, Randoop [105] often uses the value "hi!" as a string value. Clearly this value will not pass the `SqlParser` string sanity checks.

Second, some test generators extract constants, e. g., stored in fields of the class under test, and return values of previously called methods and use these values as inputs. This approach is effective if suitable constants and methods are available, but fails otherwise.

Third, some test generators symbolically reason about expected values [58], e. g., based on a constraint solver able to reason about strings [158]. While effective, this approach often suffers from scalability issues and may not provide the best cost-benefit ratio which is crucial for testing [20].

Fourth, some test generators for UIs testing use automated monkey testing together with human knowledge to manually collect input data, which is time consuming and expensive for a large code-base [87].

Finally, some test generators rely on a grammar that describes expected string values [56, 152]. However, including grammars for all, or even most, domain-specific input formats into a general-purpose test generators is impractical.

Contrary to existing input generation approaches, TestMiner is a scalable and a general approach that exploits the wealth of information available in existing code bases. TestMiner extracts the domain-specific information encoded in existing tests, using an information retrieval-inspired mining technique that predicts input values suitable for testing a particular method.

TestMiner relates to approaches for mining method call sequences to be used during test generation [135], but is orthogonal because we mine test input values. Another existing approach obtains values via a search engine [94]. In contrast to TestMiner, it relies on web pages that list example values for a particular domain and on an external search service whose implementation is neither known nor under the control of the test generator. Recent work applies advanced machine learning techniques on large code bases to address software engineering problems, e. g., by suggesting API usages [63], detecting plagiarism [68], and completing partial code [115].

To achieve generality of the approach we must define the program context associated with values found in existing tests [87]. The context definition must allow a test generator to easily query TestMiner for later values retrieval. In this dissertation we define the context to be the fully qualified signature of methods where a literal appears to be an argument of a call to this method. We found a method signature to be a natural mapping for the context first, because method calls are the main execution unit in object-orient programs. Second because method calls are the main unit of composition of unit-tests built by test generators [105], and finally because method signatures carry semantic information in the form of natural language [67].

To achieve scalability we must address the challenge to efficiently search across thousands of values given their associated context (i. e., method signature). To this end TestMiner applies state-of-the-art information retrieval techniques that are used daily in search engines achieving low response query times for very large data-sets [29].

4.1.1 *Challenge 1. Effective Indexing of Literals*

At first, TestMiner extracts literals from the source code of existing tests and indexes them for quick retrieval. The indexing is designed in such a way that the approach can predict suitable values for method signatures not seen during the mining, e. g., because the natural language terms used in the signature overlap with an already seen signature.

The assumption we make for TestMiner is that the multitude of analyzed projects contain classes that share common domain-specific knowledge, and that domain-specific literals are spread across projects test-suites, to test a project components. For example, Listing 4.1 shows two SQL classes. In the multitude of projects we analyzed, there exist classes that use an SQL-database as a back-end for data storage. These classes implement or call other SQL utility classes using string literals as arguments for SQL queries. The key idea of TestMiner is to mine calls in existing tests to re-use the mined literals to test classes like `SqlParser`. The challenge is to correctly map the signatures in a generated test to semantically similar signatures in the mined corpus.

Take for example the partially generated test in Figure 4.1 for the method `parse` of the class `SqlParser`. In the example the test generator must select a value for the first method argument in the partial test (highlighted and indicated with `???`). In the corpus we analyzed we found multiple calls to SQL related classes like `java.sql.ResultSet.getClob`. These calls have string literal arguments such as `"SELECT * FROM javaSavedTable"`. and `"create table TEST(COL varchar(128))"`. For the signature `org.pkg.SqlParser.parse` in the example, it is clear that the signature contains two semantically important keywords (tokens) "parse" and "sql". These keywords should be prioritized when matching signatures but they still must be compared to the more commonly used tokens like "org" used in package names. A naive solution to filter these common tokens could be to manually filter them. However this solution is impractical because it will require endless manual work. A practical and more effective solution to avoid this issue is to assign a weight to each token. Intuitively tokens that carry a semantically relevant meaning are assigned an higher weight, and less important tokens are assigned a possibly lower weight. This is a well known problem that is addressed by state-of-the-art information retrieval techniques. To assign weights to tokens, TestMiner uses the *tf-idf* function, see Section 4.2 for more details. This weighting function guarantees to assign an higher weight to non-recurring tokens such as "sql", but it assigns a lower weight to common tokens like "org". In the example in Figure 4.1 the *tf-idf* function assigns for the method signature `org.pkg.SqlParser.parse` the following weights to tokens:

$$\{org : 5, pkg : 10, sql : 45, parse : 40\}$$

Such assignment satisfies our requirement to increase importance of semantically relevant keywords like "'sql".

4.1.2 *Challenge 2. Fast Prediction of Values*

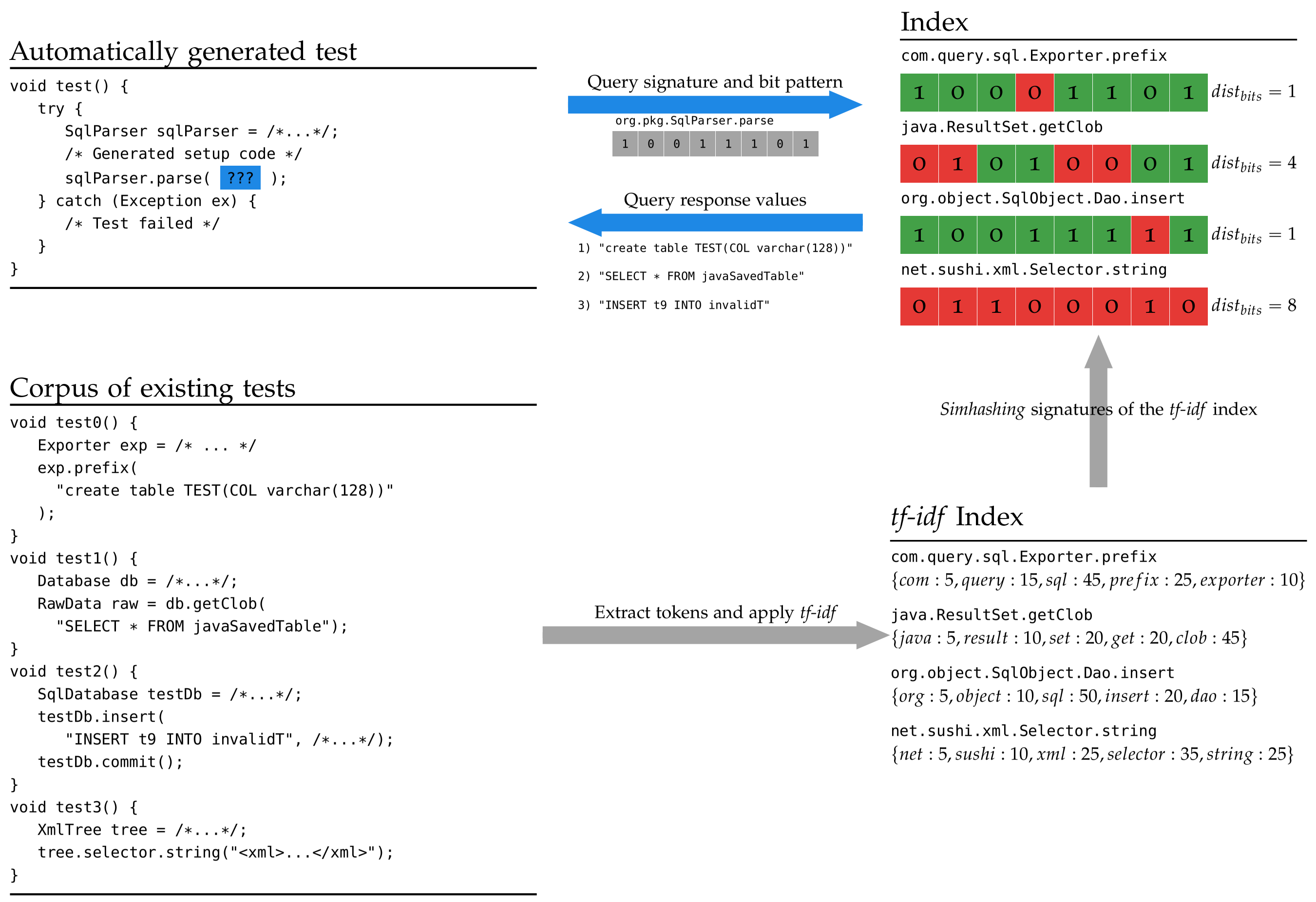
The mining and indexing part of TestMiner is executed only once, independently of the specific software under test and the test generator, and the same indexed data can serve inputs for various queries. Then, a test generator queries the mined indexed data for suitable values for a given class under test, and the predicted values are then used as test inputs during test generation.

When testing the `SqlParser.parse` method in Listing 4.1, TestMiner suggests string values that relate to the domain encoded in the signature of the method under test. Using these values during test generation enables the generated tests to cover code not touched with randomly generated inputs, which in turn also enables the approach to effectively test other classes in the code base, such as `SqlOptimizer`.

To efficiently retrieve the indexed data TestMiner matches tokens from the query signature to tokens in the indexed data. To this purpose TestMiner adopts the locality-sensitive hash function Simhashing [29] to transform a sequence of tokens with their respective weights into a bit vector representation. TestMiner groups similar vector representations in the index and then performs a bit-wise comparison of the vector representations in the index to match vectors that have the lowest bit distance.

Figure 4.1 shows this process graphically. In the example TestMiner uses a 8 bit-vector representation for the index, which results in the represented bit patterns. Assume that TestMiner only indexed the four signatures found in existing tests represented in the example and that the query signature is `org.pkg.SqlParser.parse`. Figure 4.1 shows that the bit-vector representation for the signatures `com.query.sql.Exporter.prefix` and `org.object.SqlObject.Dao.insert` differ of only one bit ($dist_{bits} = 1$) from the query signature. As a reply to the query TestMiner collects values associated with the two signatures in the index and returns them to the test generator.

Since retrieving values from indexed data is cheap, the technique is more efficient than existing heavyweight techniques, such as symbolic reasoning. At the same time, the technique is more effective than existing efficient techniques, such as using random values, because TestMiner returns values suitable for the domain of the tested code.


Automatically generated test
Query signature and bit pattern
org.pkg.SqlParser.parse
1 0 0 1 1 1 0 1
Query response values
1) "create table TEST(COL varchar(128))"
2) "SELECT * FROM javaSavedTable"
3) "INSERT t9 INTO invalidT"
Index
com.query.sql.Exporter.prefix
1 0 0 0 1 1 0 1 $dist_{bits} = 1$
java.ResultSet.getClob
0 1 0 1 0 0 0 1 $dist_{bits} = 4$
org.object.SqlObject.Dao.insert
1 0 0 1 1 1 1 1 $dist_{bits} = 1$
net.sushi.xml.Selector.string
0 1 1 0 0 0 1 0 $dist_{bits} = 8$
Simhashing signatures of the *tf-idf* index
Corpus of existing tests
Extract tokens and apply *tf-idf*
tf-idf Index
com.query.sql.Exporter.prefix
$\{com : 5, query : 15, sql : 45, prefix : 25, exporter : 10\}$
java.ResultSet.getClob
$\{java : 5, result : 10, set : 20, get : 20, clob : 45\}$
org.object.SqlObject.Dao.insert
$\{org : 5, object : 10, sql : 50, insert : 20, dao : 15\}$
net.sushi.xml.Selector.string
$\{net : 5, sushi : 10, xml : 25, selector : 35, string : 25\}$

```
void test() {
   try {
      SqlParser sqlParser = /*...*/;
      /* Generated setup code */
      sqlParser.parse( ??? );
   } catch (Exception ex) {
      /* Test failed */
   }
}
```

```
void test0() {
   Exporter exp = /* ... */
   exp.prefix(
     "create table TEST(COL varchar(128))"
   );
}
void test1() {
   Database db = /*...*/;
   RawData raw = db.getClob(
     "SELECT * FROM javaSavedTable");
}
void test2() {
   SqlDatabase testDb = /*...*/;
   testDb.insert(
      "INSERT t9 INTO invalidT", /*...*/);
   testDb.commit();
}
void test3() {
   XmlTree tree = /*...*/;
   tree.selector.string("<xml>...</xml>");
}
```

Figure 4.1: TestMiner generates inputs data by minining existing tests. TestMiner matches similar signatures by first tokenizing the mined signatures and then it assigns weights to the method signature tokens by applying the *tf-idf* function. Finally it indexes the token signatures using the similarity hashing function Simhashing to create an index a bit-vector representation for the signatures. To query TestMiner a previously unseen signature bit-vector is matched against signatures in the index. In the figure matching bits are marked in green, and the non matching bits are marked in red. In this example only two signatures closely match the query signature because they share the "sql" keyword.

4.2 APPROACH

This section presents TestMiner in detail. The input to TestMiner is a corpus of projects with existing test-suites. A test-suite is a collection of unit-tests run to verify the correctness of a program's implementation. Due to the prevalence of unit-tests in most real-world projects, there are thousands of suitable test-suites available. Nevertheless TestMiner can also benefit from the source-code of the project itself.

The first step of the approach is to statically analyze the test-suites to extract test input values. The analysis yields pairs of values and a summary of the context where a value occurs.

The second step of the approach is to summarize and index these values into a representation that enables quick retrieval of the values. TestMiner exploits techniques popular in information retrieval, such as a locality sensitive hashing function called Simhashing [124].

Finally, the third and last step is to enable an automated test generator to retrieve values when creating new test-cases. The test generator queries TestMiner, which in turn provides suitable test input values.

The approach is designed to provide three properties that are essential to achieve the overall goal of generating more effective tests:

1. *Scalable*. The huge space of available code and tests provides input values for various domains. To benefit from this knowledge, the approach must scale well to a large number of corpus projects and input values.
2. *Efficient*. The overall success of a test generator is determined by the number of bugs it reveals in a given time budget. To generate tests efficiently, TestMiner should provide input values quickly yet targeted at the code under test.
3. *Generalizing*. TestMiner aims at testing code that is not in the analyzed corpus. To achieve this goal, the approach should generalize from the context in which an input value occurs and suggest this value in a different, yet similar context during test generation. For example, as illustrated in Figure 4.1, TestMiner may learn from tests of one SQL parser what values to use for testing another SQL-related software.

4.2.1 *Static Analysis of Test Suites*

This first step extracts input values from the program corpus and associates each value with a context. The context will be later used in the retrieval phase when the test generator requests values. The input to the static analysis is the source of the tests in the corpus. The analysis extracts values that appear as literals in the code. The output of the analysis is a set of context-value pairs.

Definition 13 (Context-value pair)**.** *A context-value pair* $(\mathcal{C}, v)$ *consists of a context* $\mathcal{C} : \mathcal{S} \to \mathbb{N}$*, represented as a bag of words, and a value* $v \in \mathcal{V}$ *in some value domain. The set* $\mathcal{S}$ *refers to the set of all strings.*

The context $\mathcal{C}$ may be anything that can be represented as a bag of words. There are various options for defining the context, such as the calling context for the method under test, the type hierarchy of the program, or the signature of a method that is tested.

In this thesis, we compute the context of a value from the statically typed signature of the method that receives the value as an argument. For example, suppose a test case calls `sqlParser.parse("SELECT x FROM y")`, then the context for the value `"SELECT x FROM y"` is computed from the fully qualified signature of the `parse` method. This notion of context works well because fully qualified method signatures often contain rich semantic information [67].

To extract values, the static analysis first transforms each test's source into an abstract syntax tree. The analysis traverses the tree to collect call sites that contain literal arguments. At each call site the analysis collects the method signature, and it annotates each argument with its static type. The analysis does not perform any constant propagation, because we observed that it provides little benefit in practice. The analysis returns for each test a set of call site tuples.

Definition 14 (Call site tuple)**.** *A call site tuple* $S_c = (\mathcal{T}_c, m_c, \mathcal{V}_c)$ *consists of a set* $\mathcal{T}_c$ *of fully qualified type names in which the method* m_c *is defined, the method name* m_c*, and a set* $\mathcal{V}_c$ *of values passed as arguments at the call site.*

For example, suppose a test case calls `sqlParser.parse("SELECT x FROM y")`. Then the static analysis extracts the following call site tuple:

$$(\{org.pkg.SqlParser\}, parse, \{\text{"SELECT x FROM y"}\})$$

The set $\mathcal{T}_c$ may, in principle, contain multiple type names because a call site may resolve to multiple methods (e. g., because of overriding of the method in sub-classing). Our static analysis considers only the statically declared type of the base object of a call, i.e., $|\mathcal{T}_c| = 1$. Completely resolving all possible

Algorithm 7 Summarize and index context-value pairs.

```
Input: Set $\mathcal{P}$ of context-value pairs
Output: Index-to-values map $\mathcal{M}$
1: $\mathcal{M} \leftarrow$ empty map
2: for each $(\mathcal{C}, v) \in \mathcal{P}$ do
3:     $\mathcal{C}_{weighted} \leftarrow normalize(tfidf(\mathcal{C}))$
4:     $h \leftarrow simHash(\mathcal{C}_{weighted})$
5:     update $\mathcal{M}(h)$ with $v$
6: end for
7: return $\mathcal{M}$
```

call targets would require a whole-program analysis, which does not scale to a large corpus, and which interferes with our goal to analyze many projects in a scalable way. For the set $\mathcal{V}_c$ of values, we focus on string values in this thesis. Applying the idea to another value domain, e.g., integers, is straightforward. The reason is that finding suitable strings is a major obstacle for state-of-the-art test generators [124].

Finally, the analysis transforms the tuples into context-value pairs. For this purpose, the approach tokenizes the type names in $\mathcal{T}_c$ and the method name at dot-delimiters, splitting strings based on the camelCase and snake_case conventions, and it normalizes the remaining tokens into lower case. The resulting strings are then represented as a bag of words, which represents the context $\mathcal{C}$. For the above example, the analysis yields this context-value pair:

$$(\{org \mapsto 1, sql \mapsto 1, pkg \mapsto 1, parser \mapsto 2\}, \text{"}SELECT\ x\ FROM\ y\text{"})$$

4.2.2 *Summarizing and Indexing of Inputs Values*

Based on the set of context-value pairs extracted by the static analysis, the next step is to summarize and index these data for a later retrieval. The idea is to associate each input value with one or more hash values that summarize the context in which the input value occurs. The resulting hash map then serves as the basis for retrieval of values suitable for a particular context. Algorithm 7 summarizes the main steps of the approach, which we explain in detail in the following.

Assigning Weights to Context Words

The context of a value, as provided by the static analysis, is a bag of words. Some of these words (or terms) convey useful information about the domain of the tested code, whereas others are highly redundant. For example, consider all test input values extracted from calls to the method `org.pkg.SqlParser.parse`. The string value passed to this method is supposed to be a SQL query, i. e., a string formatted according to the SQL grammar. Therefore, the term "sql" is crucial for describing the context where the values passed to the method are typically used. In contrast, words such as "org" are very frequent and occur across various different contexts.

To enable TestMiner to focus on the most relevant words in a context, we compute a weight for each word (line 3 in Algorithm 7). To this end, we compute the *tf-idf* value of each term, i. e., the *term frequency-inverse document frequency*. This measure is commonly used for information retrieval. Intuitively, it represents how important a term is to a document in a corpus of documents. The document d here is the context $\mathcal{C}$, a term t is a context word in $\mathcal{C}$, and the corpus D is the set of all contexts gathered by the static analysis. Formally, we compute *tf-idf* as

$$tfidf(t, d, D) = f_{t,d} \cdot log\Big(\frac{|D|}{|\{d \in D : t \in d\}| + 1}\Big) \tag{4.1}$$

where document d is the context $\mathcal{C}$, a term t is a context word in $\mathcal{C}$, the corpus D is the set of all contexts, and where $f_{t,d}$ is the frequency of term t in document d. The weight assigned to a word is then normalized over all the *tf-idf* values in $\mathcal{C}$. Other normalization strategies are possible, e.g., log scale. For the above example the normalized weights from Equation 4.1 result in the following weights assignment:

$$\{org : 5, pkg : 10, sql : 45, parse : 40\}$$

In the example a low weight is assigned to "org" because this word occurs frequently in the corpus, and it assigns a relatively high weight to "sql" because this word is relatively uncommon but appears twice in the context. As a result, the approach champions the informative parts of the signature and penalizes the less informative ones.

Indexing with Locality-Sensitive Hashing

After assigning weights to context words, TestMiner indexes the values for efficient retrieval. A naive approach would be to build a hash map from bags of context words to values. To retrieve a value for a particular context, TestMiner checks if the same context has been observed in the corpus of projects and if it does, returns the matching values. This naive approach has several problems. The first problem is the limited scalability because the approach will scan the entire corpus for every query. The second, more important, problem is the lack of generalization of the naive approach. Specifically, the approach could not return any value for contexts that have not been observed in the corpus. As the overall goal of TestMiner is to support the generation of tests for projects beyond the corpus, the naive approach is unsatisfactory.

We address the challenge of generalizing beyond the contexts observed in the corpus using locality-sensitive hashing. This class of hash functions is designed to preserve similarities of values in their hash values. A locality-sensitive hash function assigns to very similar values the same hash value with high probability, preserving the value similarity also in the hash space.

This goal is fundamentally different from cryptographic hash functions, which aim at producing hashes that minimize collisions and that do not allow for inferring any similarities of the hashed values.

TestMiner uses the locality-sensitive hash function Simhashing [29]. Given a bag of words with weights assigned to each word, this function returns a bit vector that represents the hash.

We exploit locality-sensitive hashing by computing a hash value for each context (line 4 of Algorithm 7) and by storing values indexed by their hash value (line 5).

The final result of the indexing step of TestMiner is a map that assigns hash values to test input values:

Definition 15 (Index-to-values map)**.** *The index-to-values map* $\mathcal{M} : Boolean^k \rightarrow (\mathcal{V} \rightarrow \mathbb{N})$ *assigns a boolean vector of length k to a bag of values. The bag of values is itself a map that assigns each value to the number of occurrences of the value.*

This representation summarizes the context-value pairs extracted from different projects by grouping together all values with the same context. For example, there may be multiple input values for a context (e. g., multiple values for the same method in different call-sites). The index-to-values map will contain a single hash value for this context and map it to a bag of test input values.

4.2.3 *Retrieval of Values for Test Generation*

TestMiner provides values to a test generator. When the test generator retrieves values, e. g., to pass them to a constructor or method call, it is crucial to query in a timely manner because the time budget allocated for test generation is limited.

Integration into Test Generator

As a proof-of-concept, we integrate TestMiner into the state-of-the-art feedback-directed random test generator Randoop [105]. When Randoop requires a string value, e.g., to pass it as an argument to a method under test, we override its default behavior so that it queries TestMiner for V_{query} input values with a P_{query} probability. Randoop default behavior is to use the literal `"hi!"`, a value observed at runtime or one extracted from the byte-code constant pool.

Algorithm 8 Retrieve values from index-to-values map.

```
Input: context $\mathcal{C}_q$ and index-to-values map $\mathcal{M}$
Output: Probability distribution $\mathcal{V}_{result}$ of values
1: $\mathcal{C}_{q,weighted} \leftarrow tfidfWeightsForQuery(\mathcal{C}_q)$
2: $\mathcal{R} = searchSimhash(\mathcal{C}_{q,weighted}, \mathcal{M}, dist_{bits})$
3: return $probDistribution(\mathcal{R}, \mathcal{M})$
```

Retrieval of Values

Algorithm 8 summarizes how TestMiner retrieves test input values for a given query with a context $\mathcal{C}_q$. At first, it assigns weights to the query context words in $\mathcal{C}_q$ using the following weighting function:

$$(0.5 + 0.5\frac{f_{t,q}}{max_t f_{t,q}}) \cdot log\Big(\frac{|D|}{|\{d \in D : t \in d\}| + 1}\Big) \tag{4.2}$$

where $f_{t,q}$ is the frequency of a context word in the query and all the other terms have the same meaning as in Equation 4.1.

These weights give equivalent importance to a token frequency in the query and to its inverse document frequency in the corpus, effectively prioritizing uncommon tokens. The different weighting function prevents bias towards longer signatures that may contain multiple similar terms. Once the weights are normalized as in Algorithm 7, TestMiner matches the query context $\mathcal{C}_q$ against the contexts that have similar bit patterns using the search algorithm presented in [91]. The search algorithm function *searchSimhash* returns a map

$\mathcal{R} : String \rightarrow \mathbb{N}$) of indexed input values. This function selects input values from hash indices that differ at most in $dist_{bits}$ bits from the hashed query context. The threshold $dist_{bits}$ effectively controls the number of input values returned to the test generator. A high value for $dist_{bits}$ will match against many contexts in the corpus, resulting in a high latency query and possibly unrelated values. In contrast, a low value for $dist_{bits}$ will provide a faster query but may return only few input values. Finally, the algorithm returns a map that represents a probability distribution across suggested values where the probability of a value is proportional to its frequency across all context-value pairs. This map is further passed to the test generator as a set of alternative string values for the current method call. The final decision as to which actual value from the map to be used is left to the test generator, which may decide to use the probability distribution we provide or to randomly pick one of the values.

4.3 IMPLEMENTATION

We integrate TestMiner into Randoop 3.0.8, a state-of-the-art feedback-directed, random test generator [105]. We modify Randoop to probabilistically request literal values from TestMiner whenever the test generator requires a string value, e.g., as an argument for a method call or a constructor call. The modified test generator queries TestMiner with a probability of P_{query} and otherwise executes its default behavior. We implemented the retrieval part of our approach as a self-contained Java library, making it easy to integrate it into other test generators. The static analyses part of TestMiner is implemented on top of the Eclipse JDT framework [3]. Integrating TestMiner into Randoop required only to change about 100 lines in the `ForwardGenerator` class and to add about 70 lines of new code to communicate with TestMiner.

To query TestMiner for values to be passed to a method m, the modified Randoop performs multiple queries with different contexts C_q:

- the package, the class and the method name, and
- the class and the method name,
- the method name.

Querying with multiple contexts allows the test generator to retrieve more diverse values than with a single query because different queries emphasize different domain specific terms. For example, for a method `org.foo.Util.parseDate` we perform a query with the full signature, one with `Util.parseDate`, and one with `parseDate`.

We then combine all the returned values obtained from the three different queries into one set. From this set, the implementation randomly selects V_{query} values and provides them to Randoop as possible input values for the method.

As an optimization to reduce querying time, our implementation caches the results of queries. The motivation is that Randoop often requests values for the same method multiple times during one test generation run. This simple optimization significantly reduces the time spent for retrieving values.

4.4 EVALUATION

We evaluate TestMiner with a large corpus of Java projects and apply a TestMiner-enhanced test generator to a series of benchmark classes that are difficult for a test generator. This section reports to what extent TestMiner improves the effectiveness of generated tests (Section 4.4.2), presents examples to illustrate the strengths and weaknesses of the approach (Section 4.4.3), and evaluates the performance of TestMiner (Section 4.4.4).

4.4.1 *Experimental Setup*

DATA TO LEARN FROM To learn about values suitable in a particular context, we apply TestMiner to 3,601 Java projects from the the Maven Central Repository [6]. We use all projects with source code of tests, which yields 263,276 string values used in 37,821 different contexts.

CLASSES UNDER TEST To evaluate the effectiveness of tests generated with TestMiner, we use 40 classes from 18 open source Java projects. 20 of these classes have been previously used for evaluating test generators [54, 94, 122, 123]. We further augment the existing benchmark classes with string manipulation classes from Defects4J [77] and with parsers of different text formats. Because the overall goal of TestMiner is to suggest values for classes beyond the corpus that the approach learns from, we remove from the corpus all the call site tuples that contain a type name in the projects of the classes under test.

TEST GENERATION For each class under test, we generate tests using both the default version of Randoop and the TestMiner-enhanced version. We use a time budget of 5 minutes per class and repeat each experiment 10 times to account for the random-based nature of Randoop, where each experiment uses a different random seed. Similar parameters were used in previous work [124]. We run the tests using JUnit and measure test coverage using the JaCoCo library. The coverage is the ratio of executed branches and all branches in the source code of a class.

All experiments are done on a 24-core machine with two 2.20 GHz Intel Xeon processors E5-2650 v4, 64GB memory, running 64-bit Ubuntu 16.04 LTS from the Docker image `ubuntu:16.04`, using the latest Oracle Java 8 HotSpot VM (1.8.0.151 at the time of writing), with default values for the memory assigned to the VM.

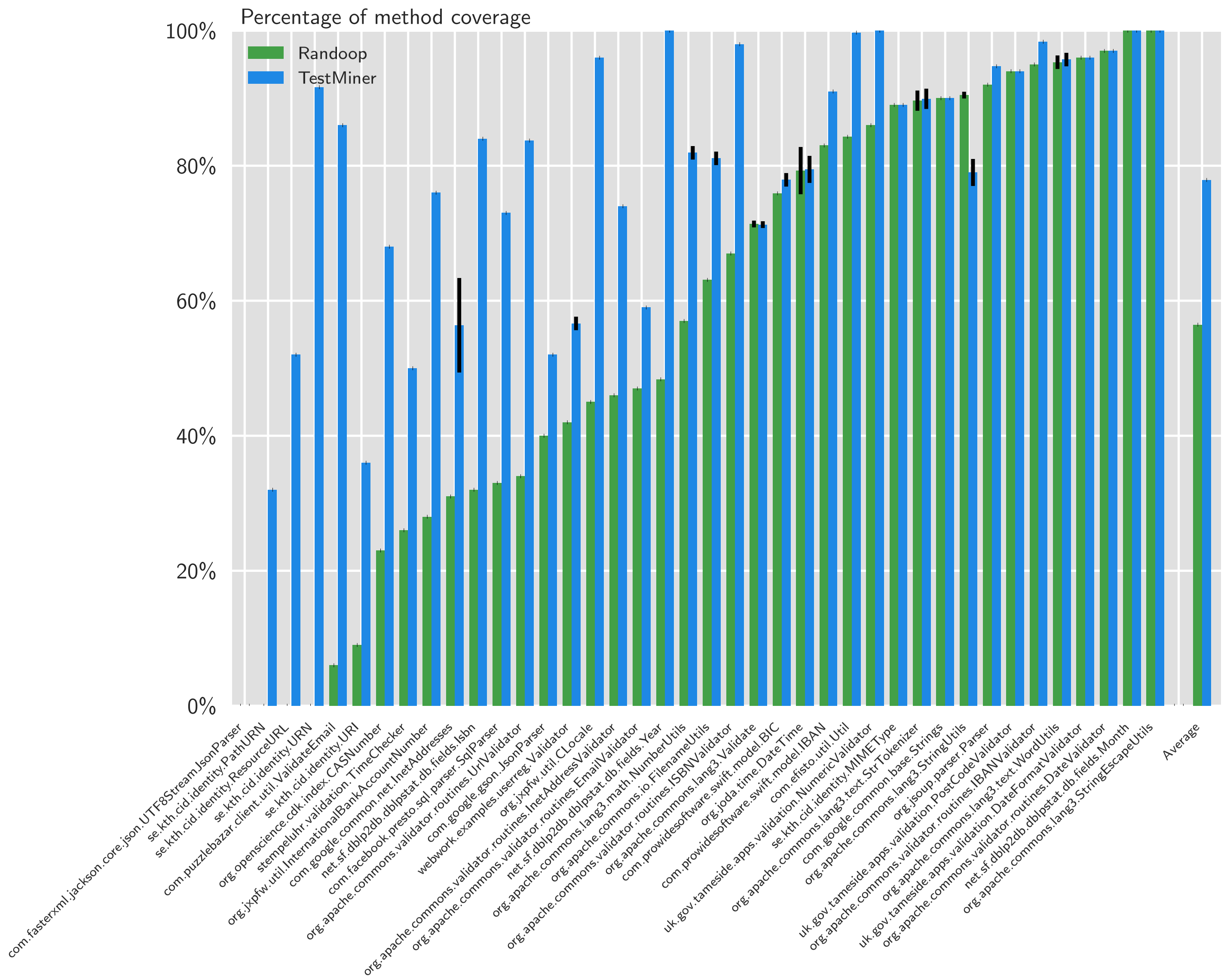


Figure 4.2: Coverage without and with TestMiner. The colored bars represent the average coverage value over 10 runs of the test generator. The size of the error bars shows the difference between the first and the third quartile.

4.4.2 *Effectiveness of Generated Tests*

To assess to what extent TestMiner increases the effectiveness of generated tests, we measure the branch coverage in each class under test. Figure 4.2 shows the average coverage over 10 test suites generated for each class. Overall, TestMiner improves the coverage for 30 classes and decreases it for 2 classes. On average over all classes, the relative improvement is 21%, increasing coverage from 57% to 78%. A Wilcoxon signed rank test shows this increase to be statistically significant ($p < 2.7 * 10^{-6}$). Cohen's d effect size, which is a statistical measure of the strength of the increase, is 0.87, which is considered large.

INCREASED COVERAGE TestMiner increases coverage from 0% to more than 30% for the classes `PathURN`, `ResourceURL`, and `URN` because Randoop is unable to instantiate these classes. In contrast, by using domain-specific strings, TestMiner helps instantiate these classes, enabling the test generator to explore their behavior. However, for the `UTF8StreamJsonParser` class, TestMiner also fails to instantiate the class. Manual inspection shows that the class requires a complicated `IOContext` object that reads JSON files and assumes such files to exist. Providing such files is out of reach for both Randoop and TestMiner but may be addressed, e.g., by symbolic testing techniques [25].

DECREASED COVERAGE TestMiner decreases the branch coverage for 2 classes. A manual inspection of the produced test suites for `StringEscapeUtils` shows that the constant retrieval fails due to an implementation error in our prototype caused by a non-escaped unicode character. For `DateTime` and `StrTokenizer`, the decrease is on average, but achieves higher coverage than Randoop for some test suites, as shown by the error bars. For the `DateTime` class, TestMiner is effective in predicting related data, but the returned values are not compatible with the class constructors and methods because most arguments are not strings. Adding support for primitive types to TestMiner can alleviate this issue. Similarly, for the class `StrTokenizer`, TestMiner fails to select correct values for the class constructor. TestMiner succeed to pass a string to be tokenized but fails to pass a valid delimiter possibly contained in the string. As a consequence the code that performs the string splitting is not covered during execution. To alleviate this issue TestMiner could consider the method parameters name, which are ignored in the current implementation.

Besides a few classes, most of the results do not show a significant difference in coverage across the 10 repetitions, i. e. the error bars are moderately small. We believe that this is due to our relatively large timeout, which allows the test generator to reach saturation no matter what the initial random seed is.

For some of the classes, like the two mentioned earlier, an even higher timeout would probably allow TestMiner to achieve better coverage.

Overall our results show that TestMiner significantly improves coverage for most classes under test.

4.4.3 *Query Result Examples*

TestMiner provides to the test generator semantically rich and diverse values. The following values are examples from tests suites generated during our experiments:

1. *IBAN:* `SCBL0000001123456702`
2. *SQL:* `'abc' LIKE '_'`
3. *Network address:* `fe80::226:8ff:fefa:d1e3`
4. *E-mail:* `test@example.org`

Many values returned by TestMiner are semantically rich values that follow a particular format, such as SQL expressions, email addresses, or international bank account numbers (IBAN), as the ones listed above. However, there are also strings that do not help in testing a specific method, such as "foo" or "metadata". Fortunately, due to the nature of feedback-directed test generation, these values are likely to be ignored in later stages of the generation process. For example, Randoop filters already seen values that did not trigger any errors during execution.

For many queries, we observe high diversity among the suggested values. TestMiner does not only produce valid IBANs like previous work [94], but it also produces almost correct values, such as a BBAN which is the last part of an IBAN. It also produces values printed in various formats, e.g., with and without spaces. Moreover, TestMiner produces IBANs of various lengths, i.e., valid in different countries. Most of these values, mined from existing tests, are given in the IBAN ISO standard documents as examples for valid or invalid values. For example, TestMiner returns the following list of values when queried with the signature `org.apache.commons.validator.routines.IBANValidator.isValid`

- `SC18SSCB11010000000000001497USD`
- `0999000000001001901229114`
- `TR330006100519786457841326`
- `SCBL0000001123456702`
- `IS14 0159 2600 7654 5510 7303 39`

```
public void test() {

SqlParser parser0 = new SqlParser();
Expression expr0 = parser0.createExpression("xx");
Expression expr1 = parser0.createExpression(
      "bound_integer in (2, 4, 3, 5)"
);
/* The exception was thrown in
   test generation */
try {
   Statement stmt0 = parser0.createStatement(
      "SELECT * FROM BOGUS_TABLE_DEF_DOESNT_EXIST"
   );
   Assert.fail("Expected exception")
} catch (NoSuchMethodError e) {
   /* Expected exception.*/
}

/* Regression assertion
   (captures the current behavior of the code)
*/
Assert.assertNotNull(expr0);

/* Regression assertion
   (captures the current behavior of the code)
*/
Assert.assertNotNull(expr1);

}
```

Listing 4.2: Example JUnit test case generated by TestMiner, which exercises code paths that Randoop alone is unable to cover. The highlighted value is provided by TestMiner.

To illustrate how the test suites produced by TestMiner differ from the ones Randoop produces, we present the test case in Listing 4.2. This is a test case generated by TestMiner for the class `com.facebook.presto.sql.parser.SqlParser`. By taking this test and adding it to the suite produced by Randoop, we are able to increase the coverage, showing that it covers a path that Randoop is not able to cover. The crucial difference, highlighted in the figure, is the valid SQL command passed to the `createStatement` method. Randoop alone is not able to create such a complex string as an input value, but TestMiner is able to make use of the knowledge mined from existing tests.

4.4.4 *Performance*

ANALYSIS AND INDEXING The static analysis takes several hours to process the entire corpus but finishes within a single day. Indexing the context-value pairs takes about 20 seconds. However downloading the large corpus of Java projects requires several days due the timeouts imposed by repositories servers.

RETRIEVAL Retrieving values from TestMiner takes longer than using Randoop's hard-coded constants. To measure slow down of test generation, we compare the size of the test suites generated by Randoop and TestMiner. In the 5 minutes time budget, Randoop generates 545,895 tests, whereas the TestMiner-enhanced test generator creates only 243,494 tests, i. e., a 55% reduction. During our evaluation, millions of string values are requested by the test generator, but the number of unique queries is only 481, allowing our implementation to make extensive use of caching to keep the runtime overhead low. Overall, TestMiner slows down the test generation, but the increased runtime cost pays off because the tests generated with TestMiner are significantly more effective.

4.4.5 *Influence of Parameters*

TestMiner has three meta-parameters, which we set experimentally to maximize coverage increase:

MAXIMUM TOLERATED BIT DISTANCE $dist_{bits}$. We report results for experiments were we set $dist_{bits} = 16$. Running TestMiner with $dist_{bits} = 4$, 8 significantly reduces branch coverage because fewer strings are returned to Randoop, which often defaults to its built-in strings. Setting $dist_{bits} = 32$ drastically increases query time and reduces the number of generated test in the time budget.

QUERY PROBABILITY P_{query}. We report results for experiments were we set $P_{query} = 0.5$, such as a coin-toss. Running TestMiner with $P_{query} = 0.25$, 0.75 provides a lower branch coverage. Using a lower probability will increase the number of times TestMiner uses the standard Randoop behavior. Using an higher probability will increase the amount of time spent in querying TestMiner, therefore achieving lower coverage because less tests are generated.

NUMBER OF VALUES V_{query} RETURNED BY TESTMINER. In our experiments we report results for $V_{query} = 10$. Running TestMiner with $V_{query} = 5$, 15 provides no significant difference in branch coverage. TestMiner returns the values related to a query sorted by the number of times they appear in the mined corpus. Therefore this parameter does not have a large influence on the covered branches because the most common values in the mined corpus are used with an higher probability.

4.5 RELATED WORK

4.5.1 *Test Generation*

There are various approach for automatically generating test cases: symbolic [25, 80] and concolic [57, 121] execution [26, 136, 144], random-based test generation [43, 105], and search-based testing [51]. Beyond unit tests, automated testing has been applied, e. g., to concurrent software [108, 109] and to graphical user interfaces [36, 37, 62, 108]. TestMiner is orthogonal and could be integrated into many of these approaches.

4.5.2 *Learning From Existing Code to Improve Test Generation*

Liu et al. train a neural network to suggest textual inputs for mobile apps [87]. Similar to TestMiner, they learn from existing tests how to create test inputs. TestMiner differs by using information retrieval instead of a neural network, by learning from already existing tests written by developers instead of writing tests specifically for learning, and by generating unit tests instead of UI-level tests.

Testilizer [97] mines UI tests for web applications by collecting input values for generating tests. In contrast, TestMiner statically collects input values from tests written for a different application. The "equivalence modulo input" (EMI) approach [84] tests compilers by modifying existing tests, i. e., programs, into new test. TestMiner learns how to create new tests from existing tests but applies to various applications domains beyond compilers. Several approaches improve test generation by learning from existing code which methods to call [72, 134, 135]. TestMiner differs from these techniques by improving the selection of input values instead of the selection of calls.

4.5.3 *Domain Knowledge for Test Generation*

Studies show that providing domain knowledge to test generators improves testing effectiveness [53, 117]. Several approaches obtain domain-specific inputs, e. g., by querying web sites [94, 122], web services [21], or semantic knowledge graphs [92]. All these techniques require querying the internet for retrieving values. To the best of our knowledge, TestMiner is the first offline technique to suggest domain-specific input values.

4.5.4 *Learning from Existing Source Code*

Existing work exploits natural language information in source code, e. g., to detect programming errors [86, 107] and suboptimal identifier names [10, 67], to cluster software systems [41], to infer specifications [159], and to find inconsistencies between code and comments [132]. TestMiner also exploits domain knowledge encoded in natural language, specifically in identifier names, to improve testing. Other work on learning from existing code includes learning a statistical language model for code completion [116] and applying information retrieval to the problem of bug localization [74, 83, 112, 113, 161].

4.6 LIMITATIONS

OUT OF VOCABULARY TOKENS. TestMiner does not handle out of vocabulary tokens. In other words, if a token part of the query signature is not in the indexed corpus, TestMiner returns a probability distribution set $\mathcal{V}_{result}$ with possibly unrelated content to the query signature. This behavior depends on the hashing function that the Simhashing algorithm uses to build the bit pattern of the signature bit vector. This limitation is shared among other natural language processing approaches that rely on dictionaries of fixed size. However in our case, if the result set is empty or it contains unrelated values, TestMiner will default to the standard behavior of the test generator. To alleviate this issue other approaches adopted `Word2Vec` [96] to match semantically similar tokens [87]. However, this approach needs to be trained with a dataset compatible with TestMiner (e. g., code documentation) because it was originally designed for natural languages in mind and not for a programming language setting.

NAIVE CONTEXT DEFINITION. In TestMiner we defined the context to be the statically resolved method signature. This limitation can cause TestMiner to loose potential matching opportunities due to dynamic method resolution. Other work successfully used different definitions of context, for example, the variable and argument declarations in a method body [10].

METHOD SEQUENCES FOR OBJECT INITIALIZATION. In this chapter we presented how to efficiently mine literal values from existing tests. However test generators fail to trigger faults also because of the missing knowledge how to correctly initialize objects [124]. TestMiner still suffers from this limitation. The large body of test suites we used in our evaluation can be mined to collect object initialization sequences using similar approach to [135].

4.7 SUMMARY

Test generation is a challenging problem, and finding suitable input values is an important part of this challenge. In this chapter we presented TestMiner, a new approach that learns from a large corpus of existing tests which input values to use in newly generated tests based on the domain of the tested software. The approach combines static analysis and information retrieval to extract input values, to index them based on the context in which they occur, and to provide values suitable for a specific domain to a test generator. The approach scales to thousands of analyzed projects, efficiently responds to queries for input values, and generalizes beyond the software analyzed as part of the corpus. The TestMiner approach provides a simple querying interface that enables existing test generators to benefit from domain-specific input values with little effort. We evaluated TestMiner and we show that TestMiner improves test coverage from 57% to 78%, on average, over a set of 40 classes that challenge state-of-the-art test generators.

5 CONCLUSIONS

In this dissertation we presented a series of novel program analyses to help developers find and fix performance issues. We introduced three novel techniques that advance the current state-of-the-art of performance issues detection via runtime program analysis.

In Chapter 2 we present an automatic analysis that discovers memoization opportunities, a common way to optimize programs that perform redundant work. This program analysis is novel because it takes an unsound but effective approach compared to previous work that uses conservative analyses based on method purity. We show that the novel analysis is effective in finding previously unknown memoization opportunities in real-world Java libraries. We reported the memoization opportunities that the analysis discovers to the developers of the libraries and they fixed and integrated our suggestions in the libraries code bases.

In Chapter 3 we introduce the first approach to automatically generate bottleneck exposing programs for methods. We show that our novel technique is effective and efficient in generating programs that expose previously known and unknown bottlenecks. The generated programs are short such that a developer can easily inspect them and can use them to evaluate an optimization with little manual effort.

Finally, in Chapter 4, we present an approach to predict domain-specific inputs for methods. This approach leverages the large amount of existing test-cases in open-source repositories to build an index of existing values that is fast to query. We show that this approach is able to provide domain-related values to an existing test generator, and that the approach enables the test generator to achieve higher branch-coverage over a set of difficult-to-test classes.

Given our listed contribution we state that we successfully reached our initial goal to reduce the developer's manual effort in performing the tasks required in debugging a program's performance. Achieving our initial goal was possible because the presented novel approaches increase the level of automation during the performance debugging tasks that a developer manually performs.

However, we learned a valuable lesson while facing the challenges to automate these manually executed debugging tasks. We learned that the developer must still be involved in the performance debugging loop.

Automatically fixing performance problems is an attractive property for a performance debugging tool. Nevertheless, our practical experiences and the limitations of current approaches show that only a very confined set of performance problems can be automatically fixed. In the one hand, the limitations of an automatic approach are due the shallow understanding of the program semantics that a tool possesses and to the properties that a tool can automatically deduce by analyzing the program's behavior. On the other hand, a human is capable of quickly assessing (with more precision if correctly guided by a tool) if a program transformation may improve the program's performance, while maintaining the semantics.

For example, our last contribution of this dissertation presents a practical approach that reports to a developer caching suggestions for methods that suffers from redundant computations. The approach processes the large amount of runtime information collected during a program's execution. Hidden in this information there may exists interesting facts about the program's behavior (i. e., redundant computations) that a developer may have overlooked. These facts, if properly exposed, can guide a developer to implement a profitable performance fix in short time. A developer can use the provided hints to deduce effective fixes to the program with his/her knowledge about the program's behavior.

In this dissertation we did not develop a fully fledged conservative automatic program analysis that may miss these important optimization opportunities. Instead we decided to pay a price in terms soundness to gain in exchange additional valuable information (otherwise discarded) that directed developers to profitable performance gains. As a positive example, some of the suggested fixes by MemoizeIt were accepted by developers even if integrating the changes required additional manual effort to modify the original source-code. Unfortunately, for some of our suggested fixes the developer required additional guarantees (e. g., the implemented cache to be thread-safe) or the developer rejected the changes because they were covering an uncommon use-case of the library (e. g., the cache introduced an overhead).

Despite the decision by the developers to reject some of our proposed changes because they targeted uncommon use-cases of the library, in this dissertation we show that manually crafting inputs that cover only common (performance) cases can be a disadvantage. In general it is good practice to focus on commonly used inputs because they are relevant. However, a manual approach to input generation is not always attractive because a human may forget to evaluate specific (performance) corner-cases. These corner-cases may not be initially considered relevant but they may occur in practice (e. g., by a not yet known or executed workload) causing sub-optimal performance, as shown with the programs generated by PerfSyn. Our tool synthesized programs that trigger such, considered uncommon, corner-cases and that could be fixed with little effort.

Sadly, also fully automating input generation is difficult because of the limited program's semantic knowledge available that causes to miss optimization opportunities. To alleviate this problem, a human could provide mutation primitives to PerfSyn that may strengthen the effectiveness of the approach. This work can be performed once by the developer for the future benefit of avoiding to manually create new inputs after each important application change.

To conclude this dissertation we encourage to focus future research efforts towards better assisting developers with new tools for performance debugging. These tools should guide developers with suggestions that can reduce their efforts but these tools should also consider, request, and exploit the developer's knowledge about the program under analysis.

BIBLIOGRAPHY

[1] Apache-Commons Math. `http://commons.apache.org/math`.

[2] Apache POI. `http://poi.apache.org`.

[3] Eclipse JDT. `http://www.eclipse.org/jdt`.

[4] Google guava. `https://github.com/google/guava`.

[5] Hibernate. `http://hibernate.org`.

[6] Maven Central. `https://search.maven.org`.

[7] NetBeans. `http://www.netbeans.org`.

[8] OProfile. `http://oprofile.sourceforge.net`.

[9] perf. `https://perf.wiki.kernel.org`.

[10] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. Suggesting accurate method and class names. In *European Software Engineering Conference and International Symposium Foundations of Software Engineering (ESEC/FSE)*, pages 38–49, 2015.

[11] Erik R. Altman, Matthew Arnold, Stephen Fink, and Nick Mitchell. Performance analysis of idle programs. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 739–753, 2010.

[12] Shay Artzi, Julian Dolby, Simon Holm Jensen, Anders Møller, and Frank Tip. A framework for automated testing of JavaScript web applications. In *International Conference on Software Engineering (ICSE)*, pages 571–580, 2011.

[13] Shay Artzi, Sunghun Kim, and Michael D. Ernst. ReCrash: Making software failures reproducible by preserving object states. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 542–565, 2008.

[14] Mona Attariyan, Michael Chow, and Jason Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 307–320, 2012.

[15] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, 2015.

[16] Fabian Beck, Oliver Moseler, Stephan Diehl, and Günter Daniel Rey. In situ understanding of performance bottlenecks through visually augmented code. In *International Conference on Program Comprehension (ICPC)*, pages 63–72, 2013.

[17] Swarnendu Biswas, Jipeng Huang, Aritra Sengupta, and Michael D. Bond. DoubleChecker: Efficient sound and precise atomicity checking. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 28–39, 2014.

[18] Stephen M. Blackburn et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 169–190, 2006.

[19] Peter Bodik, Moises Goldszmidt, Armando Fox, Dawn B Woodard, and Hans Andersen. Fingerprinting the datacenter: Automated classification of performance crises. In *European Conference on Computer Systems (EuroSys)*, pages 111–124, 2010.

[20] Marcel Böhme and Soumya Paul. On the efficiency of automated testing. In *European Software Engineering Conference and International Symposium Foundations of Software Engineering (ESEC/FSE)*, pages 632–642, 2014.

[21] Mustafa Bozkurt and Mark Harman. Automatically generating realistic test input from web services. In *International Symposium on Service Oriented System Engineering (SOSE)*, pages 13–24, 2011.

[22] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. ASM: A code manipulation tool to implement adaptable systems. In *Adaptable and extensible component systems*, 2002.

[23] Marc Brünink and David S. Rosenblum. Mining performance specifications. In *European Software Engineering Conference and International Symposium Foundations of Software Engineering (ESEC/FSE)*, pages 39–49, 2016.

[24] Jacob Burnim, Sudeep Juvekar, and Koushik Sen. WISE: Automated test generation for worst-case complexity. In *International Conference on Software Engineering (ICSE)*, pages 463–473, 2009.

[25] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 209–224, 2008.

[26] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: Three decades later. *Communications of the ACM*, 56(2):82–90, February 2013.

[27] Alexandru Calotoiu, Torsten Hoefler, Marius Poke, and Felix Wolf. Using automated performance modeling to find scalability bugs in complex codes. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 45:1–45:12, 2013.

[28] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. In *USENIX Annual Technical Conference (ATEC)*, pages 15—-28, 2004.

[29] Moses Charikar. Similarity estimation techniques from rounding algorithms. In *Symposium on Theory of Computing (STOC)*, pages 380–388, 2002.

[30] Bihuan Chen, Yang Liu, and Wei Le. Generating performance distributions via probabilistic symbolic execution. In *International Conference on Software Engineering (ICSE)*, pages 49–60, 2016.

[31] Tim Chen, Leonid I. Ananiev, and Alexander V. Tikhonov. Keeping kernel performance from regressions. In *Linux Symposium*, 2007.

[32] Tse-Hsun Chen, Weiyi Shang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. CacheOptimizer: Helping developers configure caching frameworks for hibernate-based database-centric web applications. In *European Software Engineering Conference and International Symposium Foundations of Software Engineering (ESEC/FSE)*, pages 666–677, 2016.

[33] Tse-Hsun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. Detecting performance anti-patterns for applications developed using object-relational mapping. In *International Conference on Software Engineering (ICSE)*, pages 1001–1012, 2014.

[34] Wei-Fan Chiang, Ganesh Gopalakrishnan, Zvonimir Rakamaric, and Alexey Solovyev. Efficient search for inputs causing high floating-point errors. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 43–52, 2014.

[35] Hyoun Kyu Cho, Tipp Moseley, Richard Hank, Derek Bruening, and Scott Mahlke. Instant profiling: Instrumentation sampling for profiling datacenter applications. In *International Symposium on Code Generation and Optimization (CGO)*, pages 1–10, 2013.

[36] Wontae Choi, George Necula, and Koushik Sen. Guided GUI testing of Android apps with minimal restart and approximate learning. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 623–640, 2013.

[37] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. Automated test input generation for Android: Are we there yet? (E). In *International Conference on Automated Software Engineering (ASE)*, pages 429–440, 2015.

[38] Holger Cleve and Andreas Zeller. Locating causes of program failures. In *International Conference on Software Engineering (ICSE)*, pages 342–351, 2005.

[39] Emilio Coppa, Camil Demetrescu, and Irene Finocchi. Input-sensitive profiling. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 89–98, 2012.

[40] Emilio Coppa, Camil Demetrescu, Irene Finocchi, and Romolo Marotta. Estimating the empirical cost function of routines with dynamic workloads. In *International Symposium on Code Generation and Optimization (CGO)*, page 230, 2014.

[41] Anna Corazza, Sergio Di Martino, Valerio Maggio, and Giuseppe Scanniello. Investigating the use of lexical information for software system clustering. In *European Conference on Software Maintenance and Reengineering (CSMR)*, pages 35–44, 2011.

[42] Igor Costa, Péricles Alves, Henrique Nazare Santos, and Fernando Magno Quintão Pereira. Just-in-time value specialization. In *Symposium on Code Generation and Optimization (CGO)*, pages 1–11, 2013.

[43] Christoph Csallner and Yannis Smaragdakis. JCrasher: An automatic robustness tester for Java. *Software Practice and Experience*, 34(11):1025–1050, 2004.

[44] Charlie Curtsinger and Emery D. Berger. Coz: Finding code that counts with causal profiling. In *Symposium on Operating Systems Principles (SOSP)*, pages 184–197, 2015.

[45] Monika Dhok and Murali Krishna Ramanathan. Directed test generation to detect loop inefficiencies. In *European Software Engineering Conference and International Symposium Foundations of Software Engineering (ESEC/FSE)*, pages 895–907, 2016.

[46] Yonghua Ding and Zhiyuan Li. A compiler scheme for reusing intermediate computation results. In *Symposium on Code Generation and Optimization (CGO)*, pages 279–290, 2004.

[47] Mikhail Dmitriev. Design of JFluid: A profiling technology and tool based on dynamic bytecode instrumentation. Technical Report SMLI TR-2003-125, Sun Microsystems, Inc., 2003.

[48] M. Dorigo, M. Birattari, and T. Stutzle. Ant colony optimization. *IEEE Computational Intelligence Magazine*, 1(4):28–39, Nov 2006.

[49] Bruno Dufour, Barbara G. Ryder, and Gary Sevitsky. Blended analysis for performance understanding of framework-based applications. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 118–128, 2007.

[50] King Chun Foo, Zhen Ming Jiang, Bram Adams, Ahmed E. Hassan, Ying Zou, and Parminder Flora. Mining performance regression testing repositories for automated performance analysis. In *International Conference on Quality Software (QSIC)*, pages 32–41, 2010.

[51] Gordon Fraser and Andrea Arcuri. EvoSuite: Automatic test suite generation for object-oriented software. In *European Software Engineering Conference and International Symposium Foundations of Software Engineering (ESEC/FSE)*, pages 416–419, 2011.

[52] Gordon Fraser and Andrea Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(8):1054–1068, 2013.

[53] Gordon Fraser and Andreas Zeller. Mutation-driven generation of unit tests and oracles. *IEEE Transactions on Software Engineering*, 2012.

[54] Juan Pablo Galeotti, Gordon Fraser, and Andrea Arcuri. Improving search-based test suite generation with dynamic symbolic execution. In *International Symposium on Software Reliability Engineering (ISSRE)*, pages 360–369, 2013.

[55] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous Java performance evaluation. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 57–76, 2007.

[56] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based whitebox fuzzing. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 206–215, 2008.

[57] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 213–223, 2005.

[58] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Automated whitebox fuzz testing. In *Network and Distributed System Security Symposium (NDSS)*, 2008.

[59] Simon Goldsmith, Alex Aiken, and Daniel Shawcross Wilkerson. Measuring empirical computational complexity. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 395–404, 2007.

[60] Liang Gong, Michael Pradel, and Koushik Sen. JITProf: Pinpointing JIT-unfriendly JavaScript code. In *European Software Engineering Conference and International Symposium Foundations of Software Engineering (ESEC/FSE)*, pages 357–368, 2015.

[61] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. Gprof: A call graph execution profiler. In *Symposium on Compiler Construction (CC)*, pages 120–126, 1982.

[62] Florian Gross, Gordon Fraser, and Andreas Zeller. Search-based system testing: High coverage, no false alarms. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 67–77, 2012.

[63] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. Deep API learning. In *European Software Engineering Conference and International Symposium Foundations of Software Engineering (ESEC/FSE)*, pages 631–642, 2016.

[64] Philip J. Guo and Dawson Engler. Using automatic persistent memoization to facilitate data analysis scripting. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 287–297, 2011.

[65] Shi Han, Yingnong Dang, Song Ge, Dongmei Zhang, and Tao Xie. Performance debugging in the large via mining millions of stack traces. In *International Conference on Software Engineering (ICSE)*, pages 145–155, 2012.

[66] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, July 1968.

[67] Einar W. Høst and Bjarte M. Østvold. Debugging method names. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 294–317, 2009.

[68] Chun-Hung Hsiao, Michael J. Cafarella, and Satish Narayanasamy. Using web corpus statistics for program analysis. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 49–65, 2014.

[69] Peng Huang, Xiao Ma, Dongcai Shen, and Yuanyuan Zhou. Performance regression testing target prioritization via performance risk analysis. In *International Conference on Software Engineering (ICSE)*, pages 60–71, 2014.

[70] Xianglong Huang, Stephen M. Blackburn, and Kathryn S. McKinley. The garbage collection advantage: Improving program locality. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 69–80, 2004.

[71] Olumuyiwa Ibidunmoye, Francisco Hernández-Rodriguez, and Erik Elmroth. Performance anomaly detection and bottleneck identification. *ACM Computing Surveys (CSUR)*, 48(1):4, 2015.

[72] Hojun Jaygarl, Sunghun Kim, Tao Xie, and Carl K. Chang. OCAT: Object capture-based automated testing. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 159–170, 2010.

[73] Xiangyang Jia, Carlo Ghezzi, and Shi Ying. Enhancing reuse of constraint solutions to improve symbolic execution. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 177–187, 2015.

[74] Hsinyi Jiang, Tien N. Nguyen, Ing-Xiang Chen, Hojun Jaygarl, and Carl K. Chang. Incremental latent semantic indexing for automatic traceability link evolution management. In *International Conference on Automated Software Engineering (ASE)*, pages 59–68, 2008.

[75] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 77–88, 2012.

[76] Milan Jovic, Andrea Adamoli, and Matthias Hauswirth. Catch me if you can: Performance bug detection in the wild. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 155–170, 2011.

[77] René Just, Darioush Jalali, and Michael D. Ernst. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 437–440, 2014.

[78] Jonathan Kaldor, Jonathan Mace, Michal Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, Vinod Venkataraman, Kaushik Veeraraghavan, and Yee Jiun Song. Canopy: An end-to-end performance tracing and analysis system. In *Symposium on Operating Systems Principles (SOSP)*, pages 34–50, 2017.

[79] Svilen Kanev, Juan Pablo Darago, Kim M. Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David M. Brooks. Profiling a warehouse-scale computer. In *International Symposium on Computer Architecture (ISCA)*, pages 158–169, 2015.

[80] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[81] Andrew J. Ko and Brad A. Myers. Debugging reinvented: Asking and answering why and why not questions about program behavior. In *International Conference on Software Engineering (ICSE)*, pages 301–310, 2008.

[82] Richard E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, pages 97–109, 1985.

[83] An Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. Combining deep learning with information retrieval to localize buggy files for bug reports. In *International Conference on Automated Software Engineering (ASE)*, pages 476–481, 2015.

[84] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 216–226, 2014.

[85] Philipp Leitner and Cor-Paul Bezemer. An exploratory study of the state of practice of performance testing in Java-based open source projects. In *International Conference on Performance Engineering (ICPE)*, pages 373–384, 2017.

[86] Hui Liu, Qiurong Liu, Cristian-Alexandru Staicu, Michael Pradel, and Yue Luo. Nomen est omen: Exploring and exploiting similarities between argument and parameter names. In *International Conference on Software Engineering (ICSE)*, pages 1063–1073, 2016.

[87] Peng Liu, Xiangyu Zhang, Marco Pistoia, Yunhui Zheng, Manoel Marques, and Lingfei Zeng. Automatic text input generation for mobile testing. In *International Conference on Software Engineering (ICSE)*, pages 643–653, 2017.

[88] Yepang Liu, Chang Xu, and S.C. Cheung. Characterizing and detecting performance bugs for smartphone applications. In *International Conference on Software Engineering (ICSE)*, pages 1013–1024, 2014.

[89] Fan Long and Martin Rinard. Staged program repair with condition synthesis. In *European Software Engineering Conference and International Symposium Foundations of Software Engineering (ESEC/FSE)*, pages 166–178, 2015.

[90] Yun Ma, Xuanzhe Liu, Shuhui Zhang, Ruirui Xiang, Yunxin Liu, and Tao Xie. Measurement and analysis of mobile web cache performance. In *International Conference on World Wide Web (WWW)*, pages 691–701, 2015.

[91] Gurmeet Singh Manku, Arvind Jain, and Anish Das Sarma. Detecting near-duplicates for web crawling. In *International Conference on World Wide Web (WWW)*, pages 141–150, 2007.

[92] Leonardo Mariani, Mauro Pezzè, Oliviero Riganelli, and Mauro Santoro. Link: Exploiting the web of data to generate test inputs. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 373–384, 2014.

[93] Darko Marinov and Robert O'Callahan. Object equality profiling. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 313–325, 2003.

[94] Phil McMinn, Muzammil Shahbaz, and Mark Stevenson. Search-based test input generation for string data types using the results of web queries. In *International Conference on Software Testing, Verification and Validation (ICST)*, pages 141–150, 2012.

[95] Jhonny Mertz and Ingrid Nunes. A qualitative study of application-level caching. *IEEE Transactions on Software Engineering*, 43(9):798–816, 2017.

[96] Tomas Mikolov, Chen Kai, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *CoRR*, 2013.

[97] Amin Milani Fard, Mehdi Mirzaaghaei, and Ali Mesbah. Leveraging existing tests in automated test generation for web applications. In *International Conference on Automated Software Engineering (ASE)*, pages 67–78, 2014.

[98] Nick Mitchell, Gary Sevitsky, and Harini Srinivasan. Modeling runtime behavior in framework-based applications. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 429–451, 2006.

[99] Martin Monperrus. Automatic software repair: A bibliography. 2017.

[100] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Evaluating the accuracy of Java profilers. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 187–197, 2010.

[101] Khanh Nguyen and Guoqing Xu. Cachetor: Detecting cacheable data to remove bloat. In *European Software Engineering Conference and International Symposium Foundations of Software Engineering (ESEC/FSE)*, pages 268–278, 2013.

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

[103] Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. Toddler: Detecting performance problems via similar memory-access patterns. In *International Conference on Software Engineering (ICSE)*, pages 562–571, 2013.

[104] Oswaldo Olivo, Isil Dillig, and Calvin Lin. Static detection of asymptotic performance bugs in collection traversals. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 369–378, 2015.

[105] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *International Conference on Software Engineering (ICSE)*, pages 75–84, 2007.

[106] Rohan Padhye and Koushik Sen. Travioli: A dynamic analysis for detecting data-structure traversals. In *International Conference on Software Engineering (ICSE)*, pages 1–11, 2017.

[107] Michael Pradel and Thomas R. Gross. Detecting anomalies in the order of equally-typed method arguments. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 232–242, 2011.

[108] Michael Pradel and Thomas R. Gross. Fully automatic and precise detection of thread safety violations. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 521–530, 2012.

[109] Michael Pradel, Markus Huggler, and Thomas R. Gross. Performance regression testing of concurrent classes. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 13–25, 2014.

[110] Michael Pradel, Parker Schuh, George Necula, and Koushik Sen. EventBreak: Analyzing the responsiveness of user interfaces through performance-guided test generation. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 33–47, 2014.

[111] William Pugh and Tim Teitelbaum. Incremental computation via function caching. In *Symposium on Principles of Programming Languages (POPL)*, pages 315–328, 1989.

[112] Mohammad Masudur Rahman and Chanchal K. Roy. QUICKAR: Automatic query reformulation for concept location using crowdsourced knowledge. In *International Conference on Automated Software Engineering (ASE)*, pages 220–225, 2016.

[113] Shivani Rao and Avinash Kak. Retrieval from software libraries for bug localization: A comparative study of generic and composite text models. In *Conference on Mining Software Repositories (MSR)*, pages 43–52, 2011.

[114] Lenin Ravindranath, Jitendra Padhye, Sharad Agarwal, Ratul Mahajan, Ian Obermiller, and Shahin Shayandeh. AppInsight: Mobile app performance monitoring in the wild. In *Conference on Operating Systems Design and Implementation (OSDI)*, pages 107–120, 2012.

[115] Veselin Raychev, Pavol Bielik, and Martin Vechev. Probabilistic model for code with decision trees. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 731–747, 2016.

[116] Veselin Raychev, Martin T. Vechev, and Eran Yahav. Code completion with statistical language models. In *Conference on Programming Language Design and Implementation (PLDI)*, page 44, 2014.

[117] José Miguel Rojas, Gordon Fraser, and Andrea Arcuri. Seeding strategies in search-based unit test generation. *Software Testing, Verification & Reliability*, 26(5):366–401, 2016.

[118] Stuart Russell. Efficient memory-bounded search methods. In *European Conference on Artificial Intelligence (ECAI)*, pages 1–5, 1992.

[119] Jennifer B. Sartor, Subramaniam Venkiteswaran, Kathryn S. McKinley, and Zhenlin Wang. Cooperative caching with keep-me and evict-me. In *Workshop on Interaction Between Compilers and Computer Architectures (INTERACT)*, pages 46–57, 2005.

[120] Marija Selakovic and Michael Pradel. Performance issues and optimizations in JavaScript: An empirical study. In *International Conference on Software Engineering (ICSE)*, pages 61–72, 2016.

[121] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for C. In *European Software Engineering Conference and International Symposium Foundations of Software Engineering (ESEC/FSE)*, pages 263–272, 2005.

[122] Muzammil Shahbaz, Phil McMinn, and Mark Stevenson. Automated discovery of valid test strings from the web using dynamic regular expressions collation and natural language processing. In *International Conference on Quality Software (ICQS)*, pages 79–88, 2012.

[123] Ali Shahbazi and James Miller. Black-box string test case generation through a multi-objective optimization. *IEEE Transactions on Software Engineering*, 42(4):361–378, 2016.

[124] Sina Shamshiri, René Just, José Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges. In *International Conference on Automated Software Engineering (ASE)*, pages 201–211, 2015.

[125] Ajeet Shankar, Matthew Arnold, and Rastislav Bodík. Jolt: Lightweight dynamic analysis and removal of object churn. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 127–142, 2008.

[126] Ajeet Shankar, S. Subramanya Sastry, Rastislav Bodík, and James E. Smith. Runtime specialization with optimistic heap analysis. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 327–343, 2005.

[127] Linhai Song and Shan Lu. Statistical debugging for real-world performance problems. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 561–578, 2014.

[128] Kavitha Srinivas and Harini Srinivasan. Summarizing application performance from a components perspective. In *European Software Engineering Conference and International Symposium Foundations of Software Engineering (ESEC/FSE)*, pages 136–145, 2005.

[129] Vincent St-Amour, Sam Tobin-Hochstadt, and Matthias Felleisen. Optimization coaching: Optimizers learn to communicate with programmers. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 163–178, 2012.

[130] C. Stewart, Kai Shen, A. Iyengar, and Jian Yin. EntomoModel: Understanding and avoiding performance anomaly manifestations. *International Symposium of Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 3–13, 2010.

[131] Thomas Stützle and Holger H. Hoos. MAX-MIN ant system. *Future Generation Computer Systems*, 16(8):889–914, 2000.

[132] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. /*icomment: Bugs or bad comments?*/. In *Symposium on Operating Systems Principles (SOSP)*, pages 145–158, 2007.

[133] Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. Qualitas corpus: A curated collection of Java code for empirical studies. In *Asia Pacific Software Engineering Conference (APSEC)*, pages 336–345, 2010.

[134] Suresh Thummalapenta, Jonathan de Halleux, Nikolai Tillmann, and Scott Wadsworth. DyGen: Automatic generation of high-coverage tests via mining gigabytes of dynamic traces. In *Tests and Proofs (TAP)*, pages 77–93. Springer Berlin Heidelberg, 2010.

[135] Suresh Thummalapenta, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. MSeqGen: Object-oriented unit-test generation via mining source code. In *European Software Engineering Conference and International Symposium Foundations of Software Engineering (ESEC/FSE)*, pages 193–202, 2009.

[136] Suresh Thummalapenta, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Zhendong Su. Synthesizing method sequences for high-coverage testing. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 189–206, 2011.

[137] Luca Della Toffola, Michael Pradel, and Thomas R. Gross. Performance problems you can fix: A dynamic analysis of memoization opportunities. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 607–622, 2015.

[138] Luca Della Toffola, Michael Pradel, and Thomas R. Gross. Synthesizing programs that expose performance bottlenecks. In *International Symposium on Code Generation and Optimization (CGO)*, pages 314–326, 2018.

[139] Luca Della Toffola, Cristian-Alexandru Staicu, and Michael Pradel. Saying "hi!" is not enough: Mining inputs for effective test generation. In *International Conference on Automated Software Engineering (ASE)*, pages 44–49, 2017.

[140] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a Java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, pages 125–135. IBM, 1999.

[141] Willem Visser, Jaco Geldenhuys, and Matthew B. Dwyer. Green: Reducing, reusing and recycling constraints in program analysis. In *European Software Engineering Conference and International Symposium Foundations of Software Engineering (ESEC/FSE)*, pages 58:1–58:11, 2012.

[142] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65, April 2009.

[143] Xusheng Xiao, Shi Han, Dongmei Zhang, and Tao Xie. Context-sensitive delta inference for identifying workload-dependent performance bottlenecks. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 90–100, 2013.

[144] Tao Xie, Darko Marinov, Wolfram Schulte, and David Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 365–381, 2005.

[145] Guoqing Xu. Finding reusable data structures. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 1017–1034, 2012.

[146] Guoqing Xu, Matthew Arnold, Nick Mitchell, Atanas Rountev, and Gary Sevitsky. Go with the flow: Profiling copies to find runtime bloat. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 419–430, 2009.

[147] Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, Edith Schonberg, and Gary Sevitsky. Finding low-utility data structures. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 174–186, 2010.

[148] Guoqing Xu and Atanas Rountev. Detecting inefficiently-used containers to avoid bloat. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 160–173, 2010.

[149] Haiying Xu, Christopher J. F. Pickett, and Clark Verbrugge. Dynamic purity analysis for Java programs. In *Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 75–82, 2007.

[150] Dacong Yan, Guoqing Xu, and Atanas Rountev. Uncovering performance problems in Java applications with reference propagation profiling. In *International Conference on Software Engineering (ICSE)*, pages 134–144, 2012.

[151] Guowei Yang, Corina S. Puasuareanu, and Sarfraz Khurshid. Memoized symbolic execution. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 144–154, 2012.

[152] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 283–294, 2011.

[153] Cemal Yilmaz, Arvind S. Krishna, Atif M. Memon, Adam A. Porter, Douglas C. Schmidt, Aniruddha S. Gokhale, and Balachandran Natarajan. Main effects screening: A distributed continuous quality assurance process for monitoring performance degradation in evolving software systems. In *International Conference on Software Engineering (ICSE)*, pages 293–302, 2005.

[154] Tingting Yu and Michael Pradel. SyncProf: Detecting, localizing, and optimizing synchronization bottlenecks. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 389–400, 2016.

[155] Xiao Yu, Shi Han, Dongmei Zhang, and Tao Xie. Comprehending performance from real-world execution traces: A device-driver case. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 193–206, 2014.

[156] Shahed Zaman, Bram Adams, and Ahmed E. Hassan. A qualitative study on performance bugs. In *Conference on Mining Software Repositories (MSR)*, pages 199–208, 2012.

[157] Dmitrijs Zaparanuks and Matthias Hauswirth. Algorithmic profiling. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 67–76, 2012.

[158] Yunhui Zheng and Xiangyu Zhang. Path sensitive static analysis of web applications for remote code execution vulnerability detection. In *International Conference on Software Engineering (ICSE)*, pages 652–661, 2013.

[159] Hao Zhong, Lu Zhang, Tao Xie, and Hong Mei. Inferring resource specifications from natural language API documentation. In *International Conference on Automated Software Engineering (ASE)*, pages 307–318, 2009.

[160] Xiaotong Zhuang, Suhyun Kim, Mauricio J. Serrano, and Jong-Deok Choi. Perfdiff: A framework for performance difference analysis in a virtual machine environment. In *Symposium on Code Generation and Optimization (CGO)*, pages 4–13, 2008.

[161] Yanzhen Zou, Ting Ye, Yangyang Lu, John Mylopoulos, and Lu Zhang. Learning to rank for question-oriented software text retrieval. In *International Conference on Automated Software Engineering (ASE)*, pages 1–11, 2015.