UNIVERSITY OF CALIFORNIA
Santa Barbara

Towards Enabling Better Understanding and Performance for Managed Languages

UCSB Tech Report #2012-16, July 2012

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Nagy Mostafa

Committee in Charge:

Professor Chandra Krintz, Chair

Professor Tim Sherwood

Professor Tevfik Bultan

June 2012

The Dissertation of
Nagy Mostafa is approved:

Professor Tim Sherwood

Professor Tevfik Bultan

Professor Chandra Krintz, Committee Chairperson

April 2012

Towards Enabling Better Understanding and Performance for Managed Languages

UCSB Tech Report #2012-16, July 2012

by

Nagy Mostafa

To my parents, my wife, my sister,

and all the brave men and women of Tahrir Square.

Acknowledgements

I am indebted to all the people who contributed in some way to the progress of this work.

I am deeply grateful to Prof. Chandra Krintz for all the support, help, guidance and encouragement that she has provided during the entire process. Chandra's positive attitude and confidence has always been a motivating factor throughout this work.

I would also like to thank Prof. Tim Sherwood and Prof. Tevfik Bultan for serving on my Ph.D. committee and for all the insightful feedback and discussions.

I am deeply grateful to my parents, my wife, my sister and my friends for their continuous support and love. Special thanks to my friends Haytham and Hassan for their encouragement.

Finally, my deepest gratitude to the courageous men and women of Tahrir Square, Egypt, for their rise against injustice and tyranny. To all the martyrs and injured I dedicate this work. You have made us all proud.

Curriculum Vitæ

Nagy Mostafa

Education

2012	Doctor of Philosophy in Computer Science, University of California, Santa Barbara.
2011	Master of Science in Computer Science, University of California, Santa Barbara.
2006	Master of Science in Computer Science, Alexandria University, Egypt.
2003	Bachelor of Science in Computer Science, Alexandria University, Egypt.

Experience

2011 – Present	Performance Engineer, Intel Corp.
2011	Summer Internship, Intel Corp.
2007	Summer Internship, Citrix Online.
2007 – 2011	Research Assistant, University of California, Santa Barbara.
2003 – 2006	Teaching Assistant, Alexandria University, Egypt

Publications

N. Mostafa, C. Krintz, C. Cascaval, D. Edelsohn, P. Nagpurkar, P. Wu. *Understanding the Potential of Interpreter-based Optimizations for Python*. UCSB Technical Report #2010-14, 2010.

N. Mostafa and C. Krintz. *Tracking Performance Across Software Revisions*. Proceedings of the 7th International Conference on Principles and Practice of Programming in Java (PPPJ), pp. 162 - 171, 2009.

N. Chohan, C. Bunch, S. Pang, C. Krintz, N. Mostafa, S. Soman, and R. Wolski. *AppScale: Scalable and Open AppEngine Application Development and Deployment*. International Conference on Cloud Computing (CloudComp'09), 2009.

N. Mostafa *An FPGA-based Chip-Multiprocessor Model*. Unpublished Master's Thesis, Alexandria University, Egypt, 2006.

Abstract

Towards Enabling Better Understanding and Performance for Managed Languages

UCSB Tech Report #2012-16, July 2012

Nagy Mostafa

Computer systems today are ubiquitous and come in variety of forms. On the low end, there are resource-constrained battery-powered handheld devices that are widely used such as smart phones, tablets and netbooks. On the high end, there are powerful highly parallel multi-core devices such as desktops, workstations and servers. However, the diversity of these devices and the the heterogeneity of their platforms complicate software development. Developers must possess knowledge of a variety of architectures, platforms and languages in order to build, optimize, tune, and deploy software efficiently.

Fortunately, there are means to facilitate software development across platforms. Revision Control (RC) systems enable concurrent collaboration between many developers with different languages and platforms expertise. Additionally, automated software deployment is made easier via across-platform software repositories that provide updates, fixes and patches. Finally, advances in managed programming languages (e.g. Java, C#, Python, Ruby, JavaScript) and their managed runtime environments (MREs)

simplify portable software development by abstracting the details of the target platforms.

Despite their benefits, these remedies complicate understanding of software behavior and extracting performance. First, RC systems allow source code changes to be made in isolation with no regard to how different modifications interact to affect behavior and performance. Second, the ease of software deployment leads to different versions of the software being used by millions of users over diverse platforms making it difficult to reason about how the application will be used "in the wild". Third, MREs abstract the hardware, hinder performance understanding, and advanced MREs usually have high startup cost and footprint. They are also complex to build and maintain, particularly for Dynamic Scripting Languages (DSLs).

In this dissertation, we investigate the question of whether we can devise novel profile analysis and collection techniques to address the above drawbacks and enable better understanding and improve performance of managed languages. We answer this question by exploring novel solutions that exploit the use of modern collaboration technologies, open source managed runtime systems, and popular software distribution mechanisms. Our techniques include a performance-aware RC system for Java programs, interpreter-based optimizations and remote compilation framework for DSLs. We describe each of our techniques in detail and present empirical evidence of its efficacy and potential.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Modern computing devices have become an integral part of our daily lives. Ranging from hand-held and embedded devices, such as smart phones, tablets, Internet TVs and in-vehicle entertainment systems, up to desktops, workstations and high-end servers, they have become pervasive, ubiquitous and rapidly growing in numbers. For example, according to the IDC [58], 500 million smart phones were sold worldwide in 2011 with a forecast of 982 million for 2015. Similarly, the number of tablets shipped worldwide climbed from 17 million in 2010 to 63.3 million in 2011.

These numerous computing machines are architecturally heterogeneous and diverse. On one end, hand-held and embedded devices are equipped with relatively simpler and slower processors. Smart phones typically come equipped with RISC-based ARM processors with clock speeds ranging from 600 MHz to 1.5 GHz while net-books are mostly based on simple in-order single- or dual-core Intel Atom processors. These platforms are targeted more towards reduced power and extended battery life than per-

formance. On the other end, desktops and high-end servers are designed to maximize performance with little or no concern for power. The performance of high-end systems stems in part from multi-core and super scalar architectures that exploit and facilitate instruction- and thread-level parallelism for performance. For example, since Intel released Sandybridge-based processors in 2011, quad-cores desktops are increasingly common. Similarly, modern Xeon-based workstations can contain up to 8 processors each with 10 processing cores which, using hyper-threading, provides execution contexts for up to 160 concurrent threads.

Providing software for this diverse and heterogeneous range of platforms is a difficult task as each class of platforms has its own design requirements. Low-end devices demand software with low memory footprint and power consumption. Conversely, high-end computers require more parallelism for better performance. Hence, software developers must provide a software version targeting every class of platforms. Furthermore, despite the consistent and rapid advances in hardware performance, software continues to require additional levels of optimizations to extract high performance. This is because improvement in hardware is coupled with an increase in software complexity to provide more features and richer user experience. However, many of the hardware capabilities are architecture-specific requiring that software be optimized for each new platform released. Matters are complicated as developers employ multiple languages and componentize their software to manage complexity, maintenance, and to increase

programmer productivity. Increasingly, high-performance software development calls for developers with a wide range of skills and expertise both in languages and platforms. All of the above makes software more difficult to build, optimize, analyze, deploy and maintain.

These challenges in building software have led to a number of remedies that address them. The primary ones are:

1. Collaboration between large and diverse developers communities,

2. An increasing reliance on repositories to share, distribute, and automatically update applications and systems, and

3. The use of high-level languages and Managed Runtime Environments (MREs) to facilitate development and portability across platforms.

We first briefly overview these advances and then present our dissertation contributions.

1.1 Addressing the Challenges of Complex Software Development

The first advance that addresses the challenges of complex software development is support for developer collaboration. By easing the process of collaboration and shared

software development, participants can share their knowledge, experience, and expertise and build more interesting, diverse, and complex software systems than is possible for any single participant to do alone. In addition to more complexity, such collaboration also leads to modular software with components written in different programming languages that inter-operate. Contribution of a diverse, distributed and often geographically distant developer community to a code-base is made possible via revision control (RC) systems. RC systems are management tools that allow collaborative and incremental contribution to shared source code. They provide access to source files, tracking of revisions and branches, tagging, merging of modifications, and reporting of conflicts. RC systems can be client-server based such as CVS, RCS and Subversion or distributed such as GIT, Mercurial and Codeville and are widely used for both open-source and proprietary software. For example, Linux and Android are maintained under GIT while FireFox and OpenOffice use Mercurial. Moreover, SourceForge [107], a famous open-source code hosting web-service, supports five RC systems: Subversion, GIT, Mercurial, Bazaar and CVS. As of July 2011, SourceForge hosts more than 300,000 projects and has more than 2 million registered users.

Recent advances in software deployment also eases the burden of complex software maintenance and heterogeneous platform support. Such systems provide efficient and reliable packaging and automatic configuration and updates of software that simplifies the software deployment so that it requires minimal to no user intervention. These

systems provide users with fixes, patches and updates to the application that execute on their computer system. With an effortless deployment process, an application can be easily made available to millions of users and different platforms. Software publishers commonly dedicate resources to host software repositories and provide tools that allow users to browse, download and update their software. For example, Debian-based Linux distributions use the Advanced Package Tool (APT) [8] and RedHat-based distributions use the yum package manager [122]. Similarly, Windows has its own automatic update service to install security patches and software updates. For hand held and embedded systems, Android-based devices get updates from the Android Market while Apple devices use the AppleStore.

Finally, recent advances in programming languages and their managed runtime environments (MREs) (e.g. Java, C#, Python, Ruby, JavaScript, and others) have significantly simplified complex application development by abstracting away the details of the underlying computing systems. These languages are object-oriented, easy to learn and provide many high-level constructs such as threading, exception handling and annotations, among others. They also come with extensive libraries supporting multitude of data structures, algorithms and APIs. They facilitate portability across heterogeneous computer systems through the use of MREs. An MRE, also known as a high-level language virtual machine (VM), is a software abstraction layer that isolates (sandboxes) the application code from the underlying operating system and hardware. Each MRE

is optimized and tuned for every platform so that the applications themselves need not be.

Using the MRE execution model, programs written in managed languages are compiled into an architecture-independent intermediate representation (typically bytecode) that is lazily loaded and executed by the MRE either via interpretation or a combination of interpretation and dynamic compilation. State-of-the-art MREs employ adaptive profile-guided dynamic optimizations and roll-back mechanisms. They also provide services that additionally boost developer productivity such as dynamic type-checking and automatic memory management (garbage collection). MREs execution engines are commonly stack-based machines (e.g. Java virtual machine, CPython, CRuby) but there are others that employ a register-based bytecode format and engine (e.g. Parrot and Dalvik).

In the recent years, Dynamic Scripting Languages (DSLs) such as Python, Ruby, JavaScript and PHP have gained popularity among developers. Originally, these programming languages were designed and used solely for connecting software components written in other languages, rapid prototyping of ideas, and text processing. They are now used to build fully-fledged, complex, client- and server-side applications. For example, YouTube is almost entirely written in Python and Ruby-on-Rails is widely used for web-development. DSLs are favored over classical managed languages, such as Java and C#, for their succinct, high-level syntax and dynamic features (e.g. dy-

namic typing). Their MREs are generally simpler and are interpreter-based making them light-weight with small memory footprint.

1.2 Dissertation Contributions

Although the above remedies facilitate complex software development, they cause understanding software behavior and extracting high performance to be increasingly challenging. Distributed, collaborative and incremental contribution to source code complicates understanding of performance evolution across software revisions. RC systems allow each developer to modify the code in isolation without regard to others' modifications. Hence, it becomes harder to reason about how different modifications made by different developers to different parts of the code inter-play together to shape overall behavior.

Additionally, making the software available via a repository leads to different versions of the software being used in various ways by millions of users over a diverse range of platforms. This diversity in usage models complicates the reasoning about how the programs will be used "in the wild" and to come up with optimizations that will benefit all users.

Although MREs enable rapid development, portability and adaptive optimizations, they suffer from many disadvantages. First, MRE's abstraction of the hardware com-

plicates performance analysis. This can be seen in simple MREs which are interpreter-based and as such, do not execute any code natively. Moreover, advanced MREs have other components (e.g. dynamic compiler, garbage collector and thread manager) run concurrently with the program and affect its behavior in an unpredictable manner. Hence, it becomes difficult to reason about program behavior running on top of MREs. Second, compilation-based MREs pose performance overhead during startup which can be problematic for short-running programs or those that demand high responsiveness. They also cause significantly higher memory footprint than interpreter-based MREs because of the translation process of compact bytecode to native code. Finally, MREs with adaptive optimizations support profile-collection, heuristics for optimizations selection, dynamic compilation, garbage collection, on-stack-replacement and de-optimization mechanism. All these features make optimizing MREs complex and hard to build, tune and maintain.

Moreover, MREs are particularly harder to build, and thus understand the performance of, for DSLs. This is because the dynamic nature of DSLs requires an advanced specializing compiler to extract any performance benefit from programs. Furthermore, unlike classical languages, DSLs specifications change constantly with every new release. This calls for a flexible, simple and easy to modify implementation of these languages. Hence, most standard DSLs implementations favor simplicity over perfor-

mance and employ interpreter-based execution despite the performance gains that are possible with compilation.

In summary, these and other modern software development trends make it challenging to

1. reason about and understand the behavior of programs across software revisions and when they are deployed in the wild, and

2. extract high performance dynamically and automatically from programs.

As a result, we require new techniques that enable better understanding of the behavior and performance of such programs, to help developers debug and optimize them, and to guide novel runtime techniques that automate performance optimization in this setting.

Hence, with this dissertation we address the question of

> ***how to employ novel profile analysis and collection techniques that exploit the use of modern collaboration technologies, open source managed runtime systems, and popular software distribution mechanisms to enable better understanding and improve performance of managed languages.***

We address this question by designing, implementing, and evaluating new techniques that

- extract and represent the behavior of managed language programs as they evolve across software revisions,

- characterize and optimize interpreter-based dynamic managed language programs, and
- target the "sweet-spot" between interpretation and compilation for dynamic language MREs to achieve the benefits of both without imposing their detriments.

Performance-Aware Revision Control Our first set of contributions in this dissertation target performance-aware revision control. Revision control (RC) systems have been key in enabling collaborative and distributed contribution to source code. Although these tools track source code revisions and automate merging and resolution of conflicts, they do not track how source code modifications impact performance. Tracking performance evolution across revisions is important to enable better understanding of overall program performance and resolving performance bugs and regressions. To enable this, we present a new service for RC systems that provides feedback to developers on how committed source code modifications affect performance. Our Performance-Aware RC System (PARCS) is a Java program profiling and analysis framework that automatically generates a program profile using test inputs when a new source code revision is checked into the RC repository. PARCS is based on annotated Calling Context Trees (CCTs) and code difference information from the RC system. We present an improved incremental CCT differencing algorithm that utilizes code difference information to find topological differences in a meaningful manner. Our algorithm

is also able to attribute each performance difference to the probable code modification causing it. We report each difference back to the developers along with a list of its probable causes. We evaluated PARCS prototype using a popular open-source JVM and set of real-life applications. Our results show that our CCT differencing algorithm combined with source code differences are capable of better CCT comparison than existing algorithms. We also demonstrate, by a case-study, that PARCS is effective in guiding developers to code modifications causing performance differences.

Characterization and Optimization of DSLs The next set of contributions that we make with this dissertation focus on understanding the behavior of and extracting high performance from Dynamic Scripting Languages (DSLs). Despite their increase in popularity, the vast majority of DSLs remain interpreted. The reason is that the dynamic nature of these languages and the frequent changes in their semantics complicates any efficient compilation-based implementation. We investigate ways of enhancing DSLs performance while maintaining the simplicity of their runtimes. We focus on the Python programming languages as a representative of these languages.

Interpreters have the advantages of being simple to develop, flexible to modify, easy to deploy and low memory footprint. They also encourage rapid development by eliminating the need to compile the code before testing it. Given all that, and despite the advances in MRE technology, interpreters have remained the standard implementation

of choice for the majority of DSLs. To enable better interpreter performance, we focus on characterizing and optimizing CPython, the standard interpretation-based Python implementation. We analyze the efficacy of existing interpreter bytecode dispatch optimizations for managed static languages on DSLs. We demonstrate that, while these optimization benefit static languages significantly, they provide no performance gain for DSLs. To explain this, we compare the performance characteristics of CPython and interpreted Java. Our findings show that DSLs interpreters' overhead originate from the runtime and not the bytecode dispatch process. Accordingly, we present a set of interpreter-based optimizations that target the discovered sources of overhead. We evaluate our optimizations using a set of community benchmarks and are able to extract performance gains of up to 28%, and 15% on average.

Remote Compilation Framework. Finally, we contribute a new approach to optimizing DSL programs using the Remote Compilation Framework (RCF). RCF implements a decoupled and distributed MRE that preserves the advantages of interpretation while having the benefits of compilation. RCF is a feedback-based offline compilation solution that is an intermediary point in the runtime design space between interpretation and dynamic compilation. It provides compilation as a remote service. While the program is executed by users in the wild, sampling profilers running on users platforms send samples to RCF server over the network. The server aggregates samples from

various users to form a global profile that it uses to guide optimizations, such as type-specialization, inlining and others, on hot code regions. RCF builds upon the principle of software repositories to send the optimized code as program update back to users. We also add adaptability to RCF via a phase detection and recompilation mechanism. RCF can continuously monitor the global program profile over time and react to significant changes that violate previous optimization assumptions. Our phase detection algorithm includes a set of novel heuristics for comparing profile intervals and detecting stable new phases.

RCF decouples program execution from compilation which enables us to achieve the benefits of feedback-based optimization while maintaining simpler client-side MRE implementation, lower memory footprint and shorter program startup time. We empirically evaluate RCF for Python using a set of community benchmarks as well as real applications. We find that our system effectively improves performance $1.1\times-1.7\times$ for real applications and $1.3\times-3.4\times$ for community microbenchmarks. We also compare RCF to a popular Python compilation-based MRE: PyPy. We find that RCF achieves similar performance with $2.7\times-7\times$ smaller memory footprint.

The outline of the remaining of this dissertation is as follows: In Chapter 2, we present and analyze our performance-aware revision control system framework. Chapter 3 overviews dynamic scripting languages and provides background for the next chapters. In Chapter 4, we present and discuss our Python characterization data and

how it relates to Java. We also explore the effectiveness of dispatch-focused interpreter optimizations. In Chapter 5, we explore the sources of overhead for Python and present and evaluate three interpreter-based optimizations to target them. In Chapter 6, we introduce our remote compilation framework for dynamic scripting languages. We also discuss its compilation model, optimizations and phase detection heuristics. Finally, we conclude our work and discuss future work in Chapter 7.

Chapter 2

Performance-Aware Revision Control

Repository-based revision control systems such as CVS, RCS, Subversion, and GIT, are extremely useful tools that enable software developers to concurrently modify source code, manage conflicting changes, and commit updates as new revisions. Such systems facilitate collaboration with and concurrent contribution to shared source code by large developer bases. In this chapter, we investigate a framework for "performance-aware" repository and revision control for Java programs. Our system automatically tracks behavioral differences across revisions to provide developers with feedback as to how their change impacts performance of the application. It does so as part of the repository commit process by profiling the performance of the program or component, and performing automatic analyses that identify differences in the dynamic behavior or performance between two code revisions.

The text in this chapter is in part a reprint of [70] ©2009 Association for Computing Machinery, Inc. Reprinted by permission. http://dx.doi.org/10.1145/1596655.1596682

We present our system that is based upon and extends prior work on calling context tree (CCT) profiling and performance differencing. Our framework couples the use of precise CCT information annotated with performance metrics and call-site information, with a simple tree comparison technique and novel heuristics that together target the cause of performance differences between code revisions without knowledge of program semantics. We evaluate the efficacy of the framework using a number of open source Java applications and present a case study in which we use the framework to distinguish two revisions of the popular FindBugs application.

2.1 Introduction

Software developers world-wide employ revision control (RC) systems for managing a vast diversity of open-source and proprietary software code bases. RC systems facilitate and support distributed, collaborative, and incremental contribution to shared source code via storage repositories and tools that provide, among other things, access to files, management, tracking, and branching of revisions, automatic resolution of conflicts, and feedback to developers when automatic conflict resolution fails or when events occur by other developers.

Client-server RC systems include CVS, RCS, Subversion, and the Visual Studio Team System (VSTS); popular RC systems that implement distributed local reposito-

ries include GIT, Fossil, Mercurial, and Codeville. Although these RC systems are stand-alone applications (the focus of our work), support for revision control can be and is integrated into other applications such as word processors, spreadsheets, and databases. Some RC systems provide additional services such as automatic testing (Visual Studio Team System (VSTS)) and defect or issue tracking (e.g. Codeville, Fossil, VSTS).

In this work, we are interested in providing a new service for RC systems: tracking of revision performance and dynamic behavior differences. To enable this, we have designed and implemented *Performance-Aware Revision Control Support (PARCS)*, a service that provides feedback to developers as to how a change that they have committed affects the behavior and performance of the overall application. Given the complexity of hardware and software and the popularity of collaborative development, such tools are key to helping developers understand the behavior of large applications and how local and incremental modifications impact overall performance over time.

PARCS is a program profiling and analysis framework that executes a program using test inputs when a new source code revision is checked into an RC repository. PARCS builds upon and extends prior work on calling context tree (CCT) profiling [3, 120, 12, 10, 127] and performance differencing [126], but is unique in that it targets two different revisions of the same program with the same input on the same platform. Prior work has focused on identifying performance differences across two

executions of the same program using different inputs or underlying platforms [126]. PARCS instruments the program and generates a precise CCT during offline (background) execution. PARCS annotates the CCT with a number of different performance metrics and can store CCTs for later comparison across revisions.

To find CCT differences, PARCS first compares the trees using common tree matching that we extend with feedback from changes that developers have made to the code and simple relaxation techniques. As a result, it incrementally identifies topological differences in the CCTs of two revisions. PARCS then classifies these differences into categories that distinguish a likely reason behind the performance differences: method addition/deletion, direct code modification, indirect code modification effect, and non-determinism. PARCS excises all subtrees rooted at nodes where these differences originate.

The CCTs that result after this pruning are topologically identical. PARCS analyzes these trees for differences in the performance metrics that annotate their nodes. For this step, PARCs employs simple weight matching and performs an iterative algorithm to identify pairs of nodes with weight differences that are significant, i.e., that are larger than the differences that are typical of a non-deterministic effect. Finally, PARCS attributes each topological and weight difference to a set of probable code changes and reports its findings back to the developer.

We implement a PARCS prototype for Java programs via extensions to the OpenJDK JVM from Oracle. We empirically evaluate the efficacy of PARCS using a number of open source Java programs and employ PARCS to identify behavioral differences between two revisions of these applications. We describe a detailed case study that we perform to attribute behavioral and performance differences to changes made between revisions using a single input for the popular FindBugs application [42]. Overall, we find that PARCS is easy to use and highly effective at helping to identify the cause of revision-based behavioral and performance differences in Java programs.

We next provide background on the techniques that PARCS builds upon and extends. We then detail the PARCS design and implementation in Section 2.3 and present a case study on our use of PARCS for revisions of the popular FindBugs application in Section 2.4. In Section 2.5, we empirically evaluate PARCS using a number of different open source applications. We then discuss related work in Section 2.6 and present our conclusions and plans for future work in Section 2.7.

2.2 PARCS

PARCS is performance-aware repository control support that identifies dynamic behavioral and performance differences that result from changes to source code from revision to revision. To enable this, PARCS employs a dynamic calling context tree

(CCT) for collection and evaluation of dynamic program behavior. PARCS compares two software revisions by identifying the topological and weight-based differences between the CCTs of the two revisions. We overview the background on CCTs and CCT topological differencing in the subsections that follow. We detail the PARCS implementation of weight-based differencing in Section 2.3.

2.2.1 Performance Profiling and Representation

PARCS collects, manipulates, and compares the dynamic behavior of a program using a data structure called calling context tree (CCT) [3]. Given a method call stack, the calling context is the list of methods that are resident on the stack at any particular time. A CCT captures the calling context of each dynamic method invocation that occurs during program execution. All activations of the same method that execute from the same calling context are aggregated into a single node. An edge X$\rightarrow$Y represents a call from method X to method Y. The calling context of node Y, thus, is captured by the series of nodes from the root of the tree down-to node Y. A CCT edge can be annotated with various execution metrics of the call it represents, such as invocation count, average execution time, ..., etc.

Figure 2.1 illustrates an example of a CCT for a program with methods A through F. Assuming that A is the entry method, (a) shows the CCT for a particular execution of the program. The numbers on the edges are invocation counts. For example, the

invocation count on the edge B→D is 2, which means that D is called twice from the context A→B.

```
public void A(){
  1: B();
       ...
  5: C();
       ...
  9: B();
}
public void B(){
  1: D();
       ...
  5: E();
}
public void C(){
  1: F();
}
public void D(){}
public void E(){}
public void F(){}
```

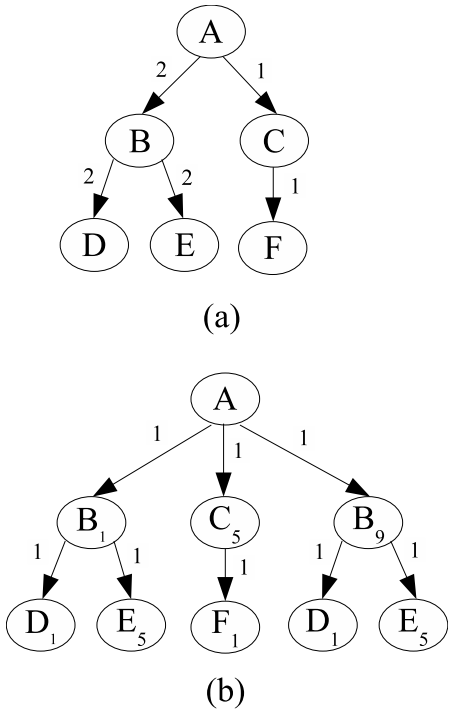


Figure 2.1: Code snippet with the corresponding CCT. (a) The corresponding CCT with no call-site information included. (b) The equivalent tree with call-site information shown as subscripts

PARCS employs CCTs for its profile collection. However, we extend its implementation to distinguish call-sites (prior work considers all calls to a method Y within the same activation of method X to have the same context [126]). In our CCTs, PARCS records a method Y that is called from two different call-sites within method X independently from each other. Figure 2.1 (b) shows the CCT with call-site information

(shown as subscripts). Distinguishing based on call-site information increases the size of the CCT but provides more details about the execution that are useful to developers for identifying behavioral and performance differences across revisions. Call-site information can also be used for measuring code coverage and anomaly detection [22]. We evaluate the trade-off between size and accuracy of employing call-site information for identifying performance differences in Section 2.5.

The call-site CCT serves as a suitable data structure for comparing performance across program revisions as it captures context information which helps programmers better understand performance and correlate it to the program semantics. Context information expressed as stack traces are still the most widely used means of describing program points of failure. Moreover, CCTs provide a good trade-off between size and accuracy compared to dynamic call trees (DCTs) and dynamic call graphs (DCGs) [3].

PARCS instruments each method entry and exit of the program to collect the CCT. Since PARCS is employed by a revision control system off-line (in the background), we do not consider the overhead of exhaustive profiling of the calling contexts. Exhaustive profiling is important for PARCS since it is able to capture all calling context behavior. PARCS annotates the collected CCTs with other profile information such as total and average execution time and invocation count. The PARCS framework is extensible enabling researchers to investigate its efficacy using other performance metrics (e.g. cache misses, branch mispredictions).

2.2.2 Identifying Topological Differences

PARCS compares two CCTs to identify the topological differences between them. In the subsections that follow, we consider two well known topological tree matching algorithms: tree transformation and common tree matching. We then present relaxed common tree matching, the algorithm that PARCS employs to identify topological differences.

Tree Transformation

Shasha et al. [2] propose a tree comparison algorithm for ordered trees; they employ dynamic programming for its implementation. An ordered tree is a tree in which the children of each node have total order. Given two trees, the algorithm finds a sequence of operations that, when applied to one tree, transforms it to the other. The algorithm is proven optimal in the number of transformation operations used (the edit distance) and has a time complexity of $O(|CCT1| \times |CCT2|)$. The transformation operations used are:

1. *Delete X:* delete node X and move its children to its parent Y; the children are inserted at the same position in the child order of Y at which X was positioned.

2. *Insert X, Y, P:* add node X to be a child of node Y at position P in the children order of Y. X gets a consecutive sub-sequence of Y's children.

3. *Rename X, Y:* rename node X to Y.

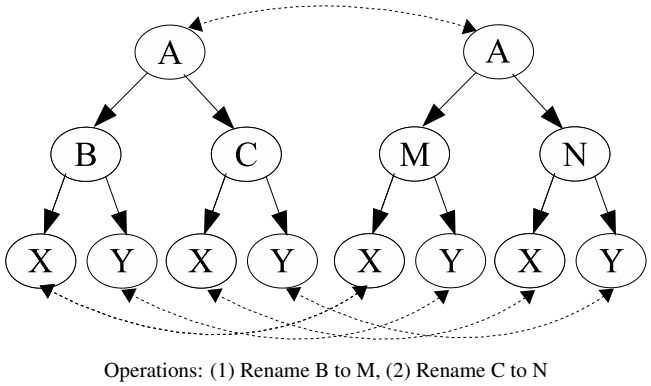


Figure 2.2: Example of tree transformation. Two rename operations needed to transform the left tree to the right one.

Although this algorithm was originally designed for abstract trees, Zhuang et al. employ it to compare two CCTs for the *same* program that they execute on different platforms or with different inputs [126]. The authors in this prior work use the number of operations required to transform CCT1 to be CCT2, as a *difference metric* with which they compare two trees.

Figure 2.2 shows two CCTs with topological difference and the minimum sequence of operations that transforms the left CCT to the right one. After applying the transformation the two CCTs become identical. All nodes that are not involved in any transformations are matched nodes. The dotted arrows in Figure 2.2 shows the matching.

There are two drawbacks, however, that discourage us from adopting this algorithm for PARCS. First, the way the algorithm matches nodes relies solely on the node label and its post-order in the tree. It ignores the context of the node (path from root to the

node) and hence may match nodes with the same method name but different contexts. For example, in Figure 2.2, unless C was renamed to N as part of code modification, method X called by C has a different semantic than X called by N. Considering the two to be equivalent could be misleading to performance analysts. Since in PARCS we are comparing two revisions of a program, these inaccuracies are more likely to occur more often since the code is different across revisions. Inaccuracies in PARCS lead to incorrect identification of differences and attribution of differences to code changes. Second, using dynamic programming incurs quadratic time and space overhead. While this is tolerable for small CCTs, it becomes hindering for larger ones. Since we rely on call-site CCTs for more accurate differencing, using this algorithm becomes infeasible. For example, the call-site CCT of FindBugs has 185,960 nodes on average. Hence, the algorithm demands a matrix of over 34 billion entries. Empirically, the algorithm runs out of memory for 80% of our test cases. For the above reasons, we investigate an alternative approach to CCT matching that is more semantically aware and suitable for large CCTs.

Common Tree Matching

Common tree matching is a well-known, simple technique for comparing two trees. The algorithm traverses the tree level-by-level, comparing nodes. Each node in the tree

has an order. The order of n ($n.order$) is the position of n amongst its siblings. For example, in Figure 2.3 (a), the order of nodes A, B and F are 1, 1 and 2, respectively.

We define equivalence of two nodes recursively as follows

Definition 1 *Node Equivalence ($\equiv$):*

Given $n_1 \in CCT_1$ and $n_2 \in CCT_2$, $n_1 \equiv n_2$ iff

1. $n_1.method\_name = n_2.method\_name$ *and*
2. $n_1.order$ = $n_2.order$ *and*
3. $n_1.parent \equiv n_2.parent$

This definition implies that equivalent nodes always have the same context. If a node has no equivalence, we consider the subtree rooted at it a topological difference.

Figure 2.3 (a) illustrates a common tree matching example (subscripts are call-sites). First, we compare root nodes, since they are equal, we proceed to the second level (A's children). On the second level, the first node B exists in both trees, thus we consider it on the common tree and will process all of its children once we move to the third level. The second node C in the left tree corresponds to F in the right, which is a mismatch; we report both C and F and their subtrees as a topological difference. We do the same thing (apply a mismatch) for node C in the right tree. We proceed similarly for the last level. The shaded nodes constitute the resulting common-tree.

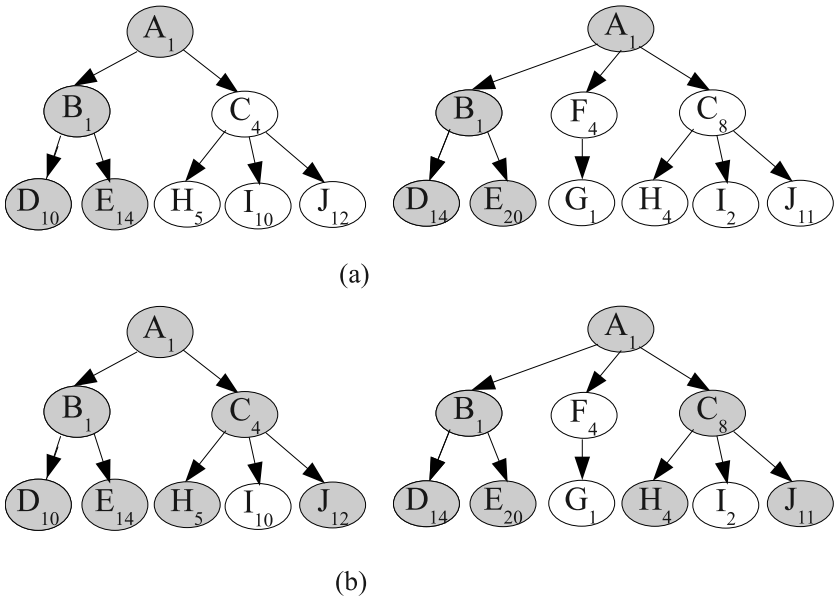


Figure 2.3: Common tree matching examples. (a) Strict common tree matching. (b) Relaxed common tree matching. Subscripts are call-sites. Shaded nodes form the common tree found in each case

The problem with common-tree matching is that it follows a very conservative definition of equivalence. In the right tree of Figure 2.3 (a), method A seems to have been modified to call method F before it calls C, shifting C one step to the right. If this is the case, we should report C as part of the common-tree. Because of the definition of equivalence, we report C in the right tree as a mismatch. To overcome this limitation and to capture such changes to source code, we relax the definition above to use the relative ordering among matched nodes instead. Our definition of equivalence then becomes

Definition 2 *Relaxed Node Equivalence($\equiv_R$):*

Given $n_1, p_1 \in CCT_1$ *and* $n_2, p_2 \in CCT_2$.

Let L_1 *and* L_2 *be the set of left siblings of* n_1 *and* n_2*, respectively.*

Let (p_1,p_2) *be the last pair of equivalent nodes found (if any).*

Then $n_1 \equiv_R n_2$ *iff*

1. $n_1.method\_name = n_2.method\_name$ *and*
2. $n_1.parent \equiv_R n_2.parent$ *and*
3. $p_1 \in L_1 \Leftrightarrow p_2 \in L_2$ *and*
4. $p_1.call\_site < n_1.call\_site \Leftrightarrow p_2.call\_site < n_2.call\_site$

We refer to the version of the algorithm that employs this definition of equivalence as *relaxed common-tree matching*. Using this algorithm, equivalent nodes still have the same context (Rule 2). The difference is that they do not have the exact child order. This is relaxed by Rules 3 and 4. Rule 3 means that for each pair of equivalent nodes, a common subsequence of nodes is found from the two sequences representing their ordered children. This takes care of cases where extra calls shift nodes among their siblings. Rule 4 adds the constraint that the relationship ($< or >$) between two consecutive children call-sites in the two sequences must be identical (note that no two different nodes can have the same parent and equal $call\_site$). This ensures that actual

shifting of nodes has taken place and that we are not matching to a wrong node that happens to have the same method name.

Figure 2.3 (b) illustrates how the relaxed common tree matching works. Despite having the same sibling order, nodes I_{10} and I_2 are not matched merely because the call site of I_{10} is greater than that of H_5 which is not the case for I_2 and H_4. This means that although the subtree at C has been shifted due to code modification (the call to F), the two invocations of I are different. On the other hand, node C is matched despite the different sibling order and the subtree rooted at F is reported as a difference. We employ relaxed common-tree matching within PARCS to identify topological differences between two CCTs (program revisions). We will quantify in Section 2.5 the improvement in matching accuracy that relaxed common tree matching achieves over conventional common tree matching.

2.3 PARCS Implementation

Figure 2.4 overviews the PARCS process. We employ PARCS for revisions of Java programs in our current prototype. We start by checking out the source code of the two revisions of interest from a code repository (e.g. CVS). We then compile the source code to bytecode. Next, we run the two revisions using the same test input via a modified Java Virtual Machine that builds CCTs from the execution. We can generate the

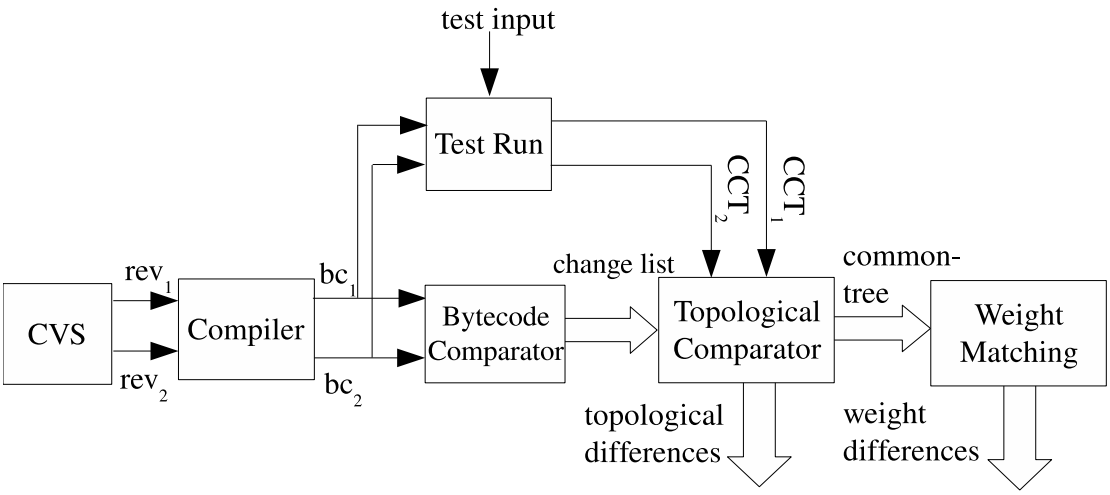


Figure 2.4: PARCS process. rev_1 and rev_2 refer to old and new revisions, respectively. bc_1 and bc_2 are the corresponding bytecodes.

CCTs of earlier revisions on-the-fly, in parallel, or store them in the repository. Developers can specify the input that PARCS uses to generate CCTs; PARCS can evaluate multiple inputs and CCT pairs concurrently.

In addition, PARCS performs a fast, static bytecode comparison on the revisions to extract method-level changes. PARCS feeds the CCTs and the bytecode change-list into the incremental topological comparator. After this component removes all topological differences from the CCTs, PARCS performs weight matching on the resulting trees to identify nodes with the largest differences in performance metrics. We detail each of these steps in the following subsections.

2.3.1 CCT Collection

The PARCS system generates CCTs by exhaustively recording all application methods calls and returns. For this study, we record only application methods and ignore calls to the Java runtime and library code to keep CCT sizes small and CCT processing fast. We can easily extend PARCS to include library calls, if necessary. Our CCTs, as described earlier, distinguish contexts for each call-site invoked. We annotate each CCT node with the invocation count, and the average and standard deviation of the method's execution time. We store all CCTs in a relational database for future analysis.

2.3.2 Method-level Bytecode Comparison

We perform bytecode comparison to generate a list of all added, deleted, modified, and renamed methods. The process starts by compiling source files from each revision code base to get the set of class files. The class files therefore belong to either the application or any local Java modules it uses. We do not consider class files that are dynamically downloaded over a network or created at runtime.

We have chosen to implement this comparison on the Java virtual machine intermediate representation (bytecode) rather than source code because of its compact and readily available format as opposed to manipulating the diff files of a repository. Also, some source code changes are useless to PARCS since they have no effect on the program semantics (e.g. variable declaration relocation within a method, variable renam-

ing, replacing a for-loop with a while-loop, ... etc.). Most of these changes are not reflected on the bytecode and hence are automatically ignored. The same argument holds for any other virtual machine intermediate form.

We match the class files from the two revisions according to their package and class names. For each matched pair of class files, we generate a list of methods that each class file contains. By comparing the two lists, we build the following method sets:

1. *Added Methods:* methods present in the new revision but not in the old one.

2. *Deleted Methods:* methods present in the old revision but not in the new one.

3. *Modified Methods:* methods present in both revisions with everything identical except for the code body.

4. *Renamed Methods:* methods present in both revision with everything identical except for the method name.

We compare methods by their fully qualified names and code bodies. A fully qualified method name consists of the full package name, class name, and method signature. The method signature consists of method name, number and type of parameters, and return type. We consider a method modified, if only its code body has been changed. Renamed methods have only changed method names.

2.3.3 Incremental Topological Comparison

Any CCT topological difference between revisions is caused by one of the following four reasons:

- **Reason 1** *Addition/Deletion of Methods:* any calls to such methods introduces a topological difference.
- **Reason 2** *Direct Modification:* code modification that explicitly enables/disables or adds/deletes a call to a method existing in both revisions.
- **Reason 3** *Indirect Modification:* a change in the program that has a side effect. For example, a global variable update or a configuration file change that affects which methods are called. Also, some modifications may have hidden effects on another method execution time, such as the effect of cache thrashing.
- **Reason 4** *Non-determinism:* Any randomness in execution.

Using the code change information that we obtain from the bytecode comparison, we label CCT nodes as "added", "deleted", "modified" or "renamed". This mapping of code change to the dynamic CCT enables topological differencing to proceed incrementally. Using these reason categories, we apply the relaxed common-tree matching technique that we describe in Section 2.2.2, incrementally in three stages:

Stage 1. We excise all subtrees rooted at added and deleted nodes and log each change for later attribution (Reason 1).

Stage 2. We identify topological differences that are most likely due to direct modification (Reason 2). Given the set of modified nodes, we identify the modified nodes that are highest dominators in the tree. X is a dominator of Y, if the path from the CCT root to Y contains X. A highest dominator is a modified node with no modified dominators (i.e. highest in the tree). Using this definition, we match highest dominators across the two CCTs by method signatures and contexts. We ignore unmatched dominators for now as we handle them in Stage 3. For each pair of matched highest dominators, we perform relaxed common-tree matching locally on the subtrees rooted at them and excise the subtrees afterwards as they become identical. The intuition behind this step is that modified nodes will likely have direct effect on the shape of their subtrees. Therefore, we perform common-tree matching locally on those subtrees to detect those effects. This heuristic improves matching accuracy by ensuring that a node match always belongs to the corresponding modified subtree which is most likely to be the correct match. If we were to perform global common-tree matching instead (on the whole CCTs), nodes under a modified subtree would not necessarily be matched to nodes under the corresponding subtree.

We report all differences found as potentially resulting from direct modification, since they are dominated by at least one modified node. Although this is the most likely cause (and is the most common in our experience), it is possible that the differences we identify result from side-effects or non-determinism.

Stage 3. Finally, we conduct a global topological comparison for what remains of the two CCTs and excise unmatched subtrees. These subtrees are present either due to side effects or non-determinism (Reasons 3 and 4) as they are not dominated by any modified nodes. In other words, none of their callers is a modified method, so the reason they are present in one revision and not the other is either they were enabled by an indirect effect of code modification or due to randomness of execution. We excise (and report) all unmatched subtrees.

2.3.4 Identifying Weight Differences

With all topological differences reported and omitted from the two CCTs, the parts remaining are identical in topology but they may vary in performance metrics. PARCS performs weight matching to identify the differences in weights across CCTs. Weight differencing is a key component of PARCS, since it identifies differences that are due to changes to the code made by the developers that change functionality without changing the method call behavior. In addition, weight matching identifies behavioral and performance differences due to modification side effects (and non-determinism).

The PARCS weight matching algorithm quantifies the degree of similarity between the two trees in terms of their annotated performance data using an overlap metric

defined and used in prior work [41, 11, 12, 126]. We define overlap in our setting as:

$$overlap(CCT_1, CCT_2) = \sum_{\substack{n_i \in CCT_1 \\ n_j \in CCT_2 \\ n_i \equiv_R n_j}} min(pweight(n_i, CCT_1), pweight(n_j, CCT_2))$$

where $n_i \equiv_R n_j$ means that n_i in CCT_1 is equivalent, under relaxed equivalence definition, to n_j in CCT_2 (the two nodes match). We define $pweight(n, CCT)$ as the percentage that the weight of node n constitutes out of the total weight of all nodes in CCT. The degree of overlap ranges from 0% to 100% and indicates how much of the performance of CCT_1 is similar to that of CCT_2, i.e. how much of CCT_2's performance is covered by CCT_1. 100% overlap indicates perfectly identical CCTs. Note that since there is non-determinism and noise in performance data, it is likely that two CCTs generated by two different runs of the same program on the same platform with the same input, do not have 100% overlap. For example, the latest revision of FindBugs application, has a 99.3% overlap in execution time between two identical runs. Figure 2.5 illustrates the common-trees from Figure 2.3 that PARCS has annotated with absolute node weights and $pweights$ (shown in parenthesis). The overlap of the two CCTs is 76%.

To identify the pairs of nodes that constitute the most significant performance difference, we employ this overlap metric as part of an iterative weight matching algorithm based upon that employed in [126]. The algorithm performs weight adjustments

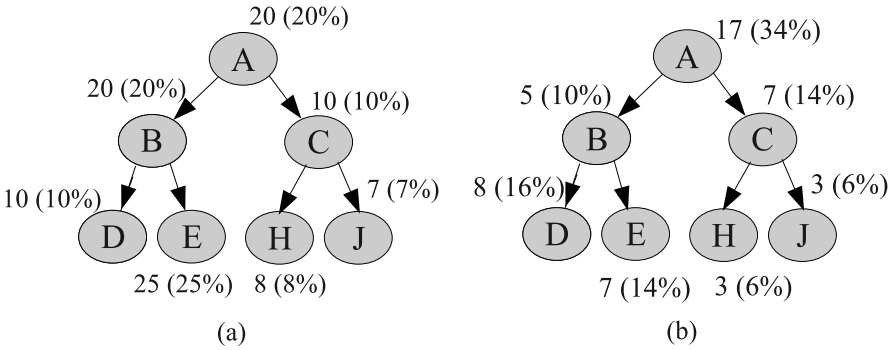


Figure 2.5: Weight matching example. Common trees with identical topology and different weights.

to improve the overlap up to a pre-defined threshold; the nodes adjusted are the ones with most significant weight difference. The algorithm is parameterized by an overlap threshold and/or number of nodes of interest. We automate generation of the overlap threshold value by computing the overlap percentage of two CCTs for the latest revision – the same program, on the same platform, using the same input, that we execute twice. This overlap value captures the difference that we expect from noise and non-determinism. Developers can set this threshold to a different value, to investigate other weight difference pairs, if so desired. Alternatively, developers can specify the number of nodes they are interested in investigating. The nodes that PARCS returns are the methods responsible for the greatest contribution to the overall weight difference between the two revisions.

2.3.5 Attributing Differences

In our current prototype of PARCS, we report each difference with an ordered list of methods that most likely contain the code change(s) that caused the difference. We also report supporting evidence and data for each method (context, performance metrics, etc.). In addition, PARCS presents the context information (annotated subtrees, complete CCTs with highlighted node differences, ...etc.) to developers in graphical format for easy viewing and investigation. The exact attribution of a difference to a specific change proceeds by hand – however with PARCS support (described below). We walk through an example of this process in the next section for two revisions of the FindBugs application.

To identify the most likely methods causing each difference that PARCS identifies, we employ a simple heuristic. For Stage 1 differences, we report the parents of the excised subtrees (callers to added/deleted methods). These parent nodes represent modified methods that either contain additional calls to added methods or calls to deleted methods removed from them. For each subtree excised in Stage 2, we report the list of modified nodes on the path from the subtree root to the CCT root starting from the closest modified dominator upwards. For Stage 3 and weight matching, we report the differences along with their context.

2.4 Usage Example: FindBugs

In this section, we demonstrate by example how we apply these heuristics to identify the reason for topological and weight differences. To enable this, we compare the CCTs of two revisions of FindBugs [42], a Java tool to find bugs statically in Java code.

First, we execute Stage 1 of the algorithm to remove all subtrees rooted at added and deleted nodes and Stage 2 to find differences dominated by modified nodes. Figure 2.6 visualizes a subset of the CCT from the latest FindBugs revision. We only show node ID's for convenience and we draw the modified methods as rectangular nodes. The nodes in gray are those that Stage 2 identifies as a difference from the CCT of the earlier revision. Those are the nodes that Stage 2 removes. Stage 2 returns the list of all modified nodes between the subtree root and the CCT root for all excised subtrees. PARCS orders the list from the modified node nearest to the subtree to the farthest. For example, for the subtree rooted at node 29748, PARCS returns the list (29744, 19913). For this case study, we first investigate the reason behind the topological differences in the three subtrees rooted at 29745, 29747 and 29748 which correspond to methods: *Item.init()*, *Item.makeCrossMethod()* and *Item.equals()*, in the FindBugs application, respectively. As we described earlier, there are three potential reasons behind these differences. The first and most likely reason is direct code modification that introduced/enabled these calls. In such cases, the modified method nodes will be one of

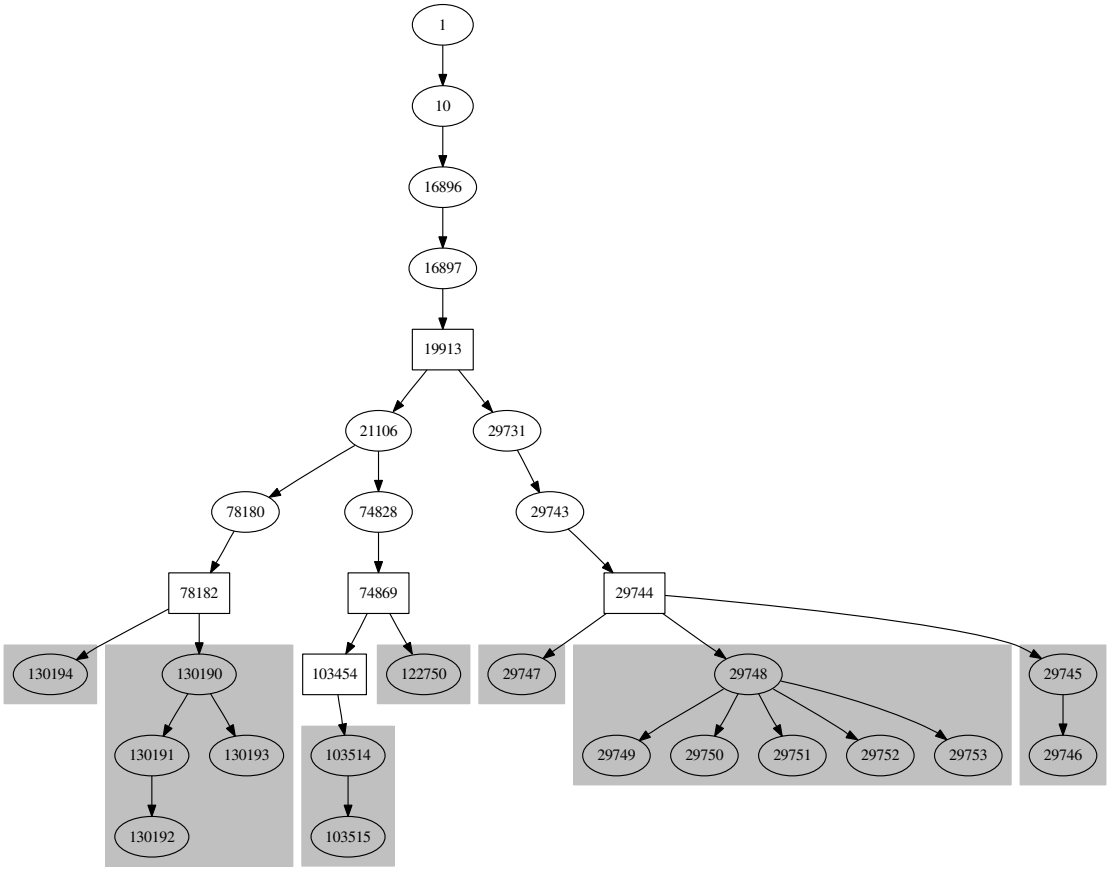


Figure 2.6: Visualization of topological differences between two Findbugs revisions. The shaded subtrees are a subset of those subtrees removed by Stage 2. Rectangular nodes represent modified methods.

the ancestors of the subtrees roots (in this case nodes 29744 and 19913) that Stage 2 returns. The second reason is a side effect of some modification that indirectly induces the subtrees. Finally, the reason may be non-determinism during execution.

We begin by investigating the methods that correspond to the nodes that Stage 2 reports in order which are (29744) *FieldSummary.setComplete()* then (19913) *FindBugs2.analyzeApplication*. Using the differences in the source code of these methods reported by the source code repository (or our bytecode analysis tool), we find that the modified method *FieldSummary.setComplete()*, inserts these three calls in the latest revision but not in the former.

We repeat the same procedure to find the cause for the different subtrees rooted at nodes 130194 and 130190. The ordered list of candidate methods that Stage 2 reports is (78182, 19913). Again, we start by the node closest to the subtree root which corresponds to method *FindUnrelatedTypesInGenericContainer.analyzeMethod()* which contains a source code change that inserts the two calls.

After removing all topological difference during Stage one and two, the only topological differences remaining, if any, will be due to either indirect modification or non-determinism of execution. By running stage three of the algorithm, we find one tree removed from each CCT both rooted at method *JavaVersion.clinit* (not shown in the figure). This method is the class initializer for the class *JavaVersion*. Analyzing the method's caller, we find that non-determinism is the reason. In particular, the use of the

Java data structure *HashSet* makes no guarantee to the iteration order of the set. The order by which the items in the *HashSet* are processed dictates when the class initializer of *JavaVersion* is invoked to cause topological difference.

Finally, we investigate the reason for the total execution time difference between the two CCTs (with topological differences excised). We find that the node with the highest difference in $pweight$ is of the method *PreorderVisitor.visitCode()*. PARCS reports that both the execution time and invocation count have changed in the new revision. The invocation count drops from 8469 in the old revision to 5427 in the new one. PARCS reports that this method invokes a call to *OpCodeStackDetector.visitCode()* that was removed in the new revision and added to the caller of *PreorderVisitor.visitCode()* instead. This change causes a drop in the invocation count of that method which decreases its total execution time.

2.5 Experimental Evaluation

Our experimental platform is a dual-core Intel Core 2 Duo machine clocked at 2.4 GHz with 4M of L2 cache and 2GB of main memory running Linux-2.6.24. The Java virtual machine we use is HotSpot version 13.0-b02 within OpenJDK 1.7.0 with our extensions for collection of performance statistics and context profiles.

App. Name	Description	Revisions
checkstyle	Code style checker for Java.	SVN revisions 2090 and 2100
doctorj	Javadoc analysis tool.	Versions 5.1.1 and 5.1.2
findbugs	Bug detector for java.	CVS commits on 21st and 25th Aug. 2008
jaranalyzer	Jar files dependency analyzer.	Versions 1.1. and 1.2
java2html	Java to Html converter.	Versions 4.1 and 4.2
jruby	Java implementation of Ruby.	Versions 1.1.2 and 1.1.3
jython	Java implementation of Python.	SVN revisions 4899 and 4981
pmd	Java code checker.	SVN revisions 6399 and 6421

Table 2.1: Description of applications studied.

Table 2.1 describes the eight open-source Java applications that we use to evaluate PARCS empirically along with the revisions/versions used. For each application, we use PARCS to compare the dynamic behavior of two close revisions of the code running with the same test input. Revisions were chosen to be no more than ten days apart. For four applications (doctorj, jaranalyzer, java2html and jruby) we compare releases instead of revisions because we did not have access to the revision control system. We exercise each application with five inputs. We use doctorj, findbugs, jaranalyzer, java2html and PARCS Java source code as inputs for checkstyle, java2html and pmd. For jruby and jython, we use five microbenchmarks as inputs: binarytrees, nsievebits, fannkuch, mandelbrot and nsieve from The Computer Language Benchmarks Game [103]. For findbugs, we use checkstyle, java2html, PARCS and two other Java projects. Finally, for jaranalyzer, we use checkstyle, findbugs, jruby, jython and a devised test input.

Table 2.2 shows the average CCT size, in number of nodes, and time-weighted average stack depth of the old and new revisions for each application over the five inputs. The numbers are close indicating the high similarity of the revisions.

App. Name	Average Node Count		Average Depth	
	old	new	old	new
checkstyle	1448937.6	1449183.6	31.76	31.81
doctorj	390814.2	390816.4	30.87	29.94
findbugs	185960.2	183552.2	19.85	19.74
jaranalyzer	569.6	565	16.58	16.28
java2html	2886.6	1075.4	10.73	10.29
jruby	321365.2	304357.6	34.54	31.95
jython	145700.4	145896.2	21.3	21.26
pmd	676247.4	676248.4	53.5	53.28

Table 2.2: Applications average CCT sizes and average stack depth for two revisions.

2.5.1 Bytecode Comparison

We use Apache Byte Code Engineering Library (BCEL) [15] to perform method-level bytecode comparison of revisions. We quantify the results of the comparison in Table 2.3. Columns two and three (OF and NF) show the number of class files from each application old revision and new revision, respectively. Columns four and five (OM and NM) show the number of methods in the old and new revisions, respectively. Columns six to nine contain the difference in terms of methods deleted (DM), added (AM), modified (MM) and renamed (RM). The highest numbers belong to jaranalyzer, java2html and jruby, for which we use releases instead of revisions. PARCS finds no renamed methods for any of the applications. This is because of the strict definition

of a renamed method that we adopt in which only the method name should change. During our tests, we have found that a method name change is always accompanied by a change in the signature or the containing class, which we classify as a method removal then addition (Section 2.3.2).

App. Name	OF	NF	OM	NM	DM	AM	MM	RM
checkstyle	1386	1386	11948	11953	3	8	11	0
doctorj	226	226	3934	3937	2	5	4	0
findbugs	3570	3569	27424	27415	12	3	11	0
jaranalyzer	413	423	3397	3486	26	115	587	0
java2html	121	132	819	873	138	192	302	0
jruby	4156	4259	25514	26653	773	1912	1592	0
jython	1819	1820	15487	15520	0	33	39	0
pmd	923	923	11336	11336	4	4	6	0

Table 2.3: Parameters and results of bytecode comparison. OF:old files, NF:new files, OM:old methods, NM:new methods, DM:deleted methods, AM:added methods, MM:modified methods, RM:renamed methods

2.5.2 Topological Difference

To evaluate the common-tree matching algorithm that PARCS employs, we quantify the total number of subtrees and nodes that PARCS removes from both trees at each stage. We also measure the size of the common-tree obtained for each application for each input.

Table 2.4 shows the results for each application over its five inputs. The third column is the common-tree size as percentage of the CCT size of the old revision. Six of the eight applications show high common tree coverage (above 85%). Pmd shows

App.	common	common	deleted		added		modified		side effects	
Name	tree size	tree size (%)	ST	ND	ST	ND	ST	ND	ST	ND
checkstyle	1105139	99.9	157	994	159	1217	31	83	18	148
	1633395	99.86	248	2172	266	2639	59	157	2	2
	1856117	99.9	308	1615	318	2014	56	139	16	104
	630763	99.97	137	137	137	137	0	0	16	68
	2009773	99.8	473	3745	509	4483	86	222	18	156
doctorj	393342	99.99	1	13	7	20	5	15	0	0
	609205	100	0	0	2	2	4	14	0	0
	108601	99.99	0	0	3	3	6	21	0	0
	207495	100	0	0	0	0	0	0	0	0
	635365	100	1	14	6	20	5	15	0	0
findbugs	135149	86.96	19	19856	6	17970	38	436	0	0
	129333	87.55	18	18079	6	16235	38	350	0	0
	187604	86.86	20	27824	6	25779	40	587	0	0
	189830	87.88	20	25610	6	23565	38	596	1	1
	168807	86.71	20	25039	6	23339	36	850	0	0
jaranalyzer	507	97.31	0	0	11	14	11	14	0	0
	567	97.09	0	0	11	14	11	14	2	6
	558	97.55	0	0	11	14	11	14	0	0
	586	97.5	0	0	11	14	12	15	0	0
	534	93.68	0	0	11	14	11	14	8	22
java2html	216	7.48	9	2671	11	853	3	7	0	0
	218	7.55	9	2671	11	853	3	7	0	0
	224	7.74	9	2671	11	853	3	7	0	0
	216	7.48	9	2672	11	853	3	7	0	0
	203	7.06	9	2671	11	853	3	7	0	0
jruby	100385	7.21	37189	884560	81821	941162	69716	661326	0	0
	9884	18.75	282	34619	578	42306	1752	10816	0	0
	9221	17.08	220	41268	459	46603	1706	4823	0	0
	9616	18.92	227	35403	453	41565	1794	7380	0	0
	9982	17.39	223	42545	486	49768	1779	6294	0	0
jython	486326	98.54	0	0	760	6830	1512	7615	10	192
	40951	85.08	0	0	755	6799	1510	7557	6	163
	54603	88.38	0	0	756	6799	1512	7613	6	163
	53554	88.08	0	0	764	6866	1511	7562	14	224
	56948	88.63	0	0	771	6923	1522	7673	16	240
pmd	526553	100	0	0	1	1	0	0	0	0
	931798	100	0	0	1	1	0	0	0	0
	826271	100	0	0	1	1	0	0	0	0
	817248	100	0	0	1	1	0	0	0	0
	279367	100	0	0	1	1	0	0	0	0

Table 2.4: Subtrees (**ST**) and nodes (**ND**) removed at each stage of topological differencing. For each application, the results for each of the five test inputs is shown.

the highest common tree ratio as only one node is reported as a topological difference. As we mentioned previously, we compare releases for jaranalyzer, java2html and jruby. As expected, java2html and jruby show low common tree coverage, while jaranalyzer surprisingly shows high coverage between its releases.

The other columns show the number of subtrees and the equivalent number of nodes that PARCS removes at each stage. The columns titled "added" and "deleted" contain data about subtrees removed due to being rooted at added or deleted nodes (Section 2.3.2). The one titled "modified" contains trees that have at least one modified node as a dominant node. "Side effects" are unmatched subtrees that cannot be classified as any of the above.

Figure 2.7 shows the amount of overlap obtained by PARCS for each test case using two overlap metrics: average execution time (a) and invocation count (b). The results are all high for both (above 82%). As expected, the overlap is less using average execution time due to noise in measurement. High overlap, however, does not mean a good match. For example, java2html shows nearly perfect overlap for both metrics yet, as Table 2.4 shows, it yields around 7% common tree. Hence, overlap alone can be misleading as it tends to be high if the common tree is small.

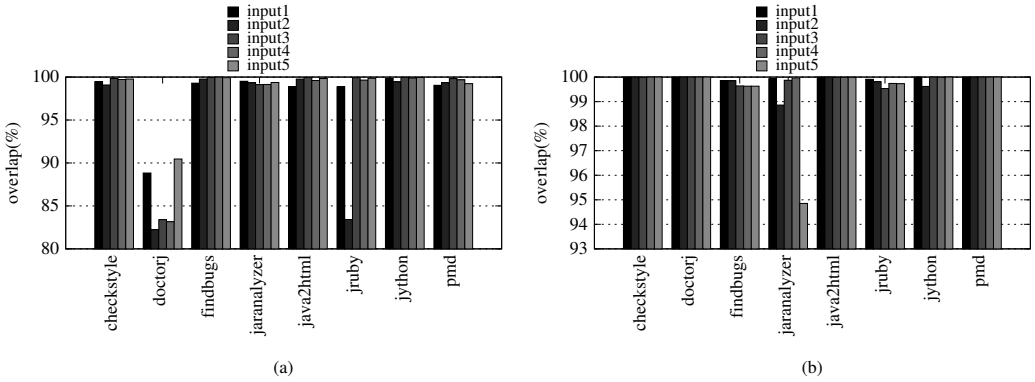


Figure 2.7: Overlap for two metrics: (a) Average execution time. (b) Invocation count.

To better measure the matching accuracy, we define the following matching score metric:

$$score(CCT_1, CCT_2) = (\prod_{o_i \in O} o_i(CCT_1, CCT_2)) \times \frac{|CT|}{|CCT_1|} \times \frac{|CT|}{|CCT_2|} \times 100$$

where $|CT|$ is the common tree size and O is the set of overlap metrics used. In our case, we use three overlap metrics: invocation count, average execution time, and total execution time. Informally, a match with high score is a match whose common tree covers high percentage of the CCT and has high overlaps as well. Figure 2.8 shows the scores for all applications and all inputs. 6 out of 8 apps exhibit very high score (above 70%). Since we are comparing versions for jruby and java2html, they show very poor matching scores (below 4%). This demonstrates that PARCS is highly tailored for close revisions comparison where code modifications are incremental. Nevertheless, for jaranalyzer the score is surprisingly high even though we are comparing releases.

We next compare matching scores of relaxed common tree matching against strict matching. Figure 2.9 shows the average scores over all inputs for each application. Relaxed common tree matching shows significant improvement over strict matching for jaranalyzer, jython, findbugs and jruby. For the remaining cases, there is still slight improvement.

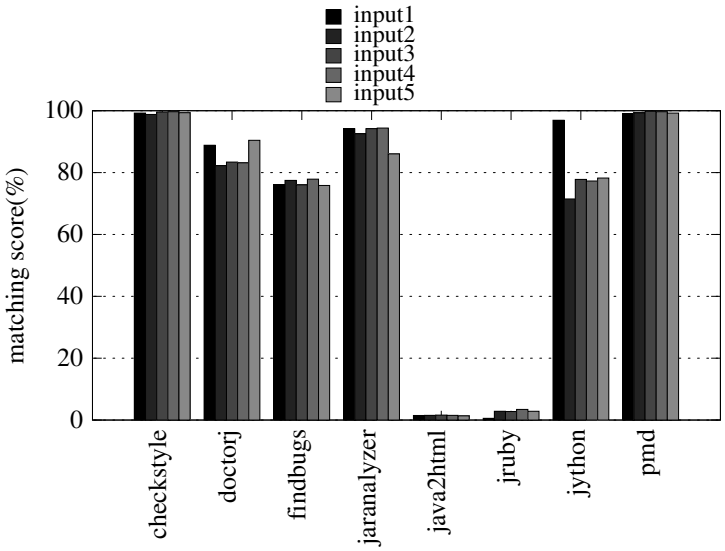


Figure 2.8: PARCS matching scores

We also study the benefit of using CCTs with call-site information. To assess the additional differences revealed via call-site CCTs, we have compared the number of nodes removed as topological differences using both types of CCTs. Higher number of nodes removed means more differences that PARCS discovers. Table 2.5 shows the average number of nodes removed for each application over five inputs. For most applications, the difference is significant. For example, checkstyle has 2404 nodes removed

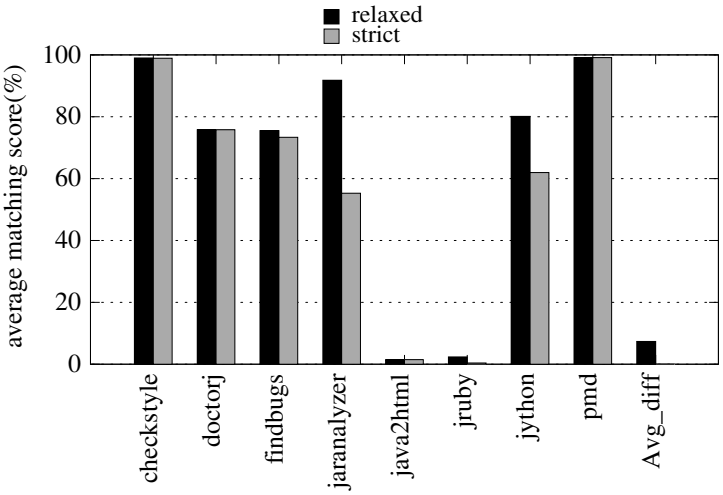


Figure 2.9: Comparison of matching scores between relaxed and strict common tree matching

which gets nearly doubled when using call-site CCT. jruby shows drastic increase in node count due to its recursive nature.

App. Name	nodes removed w call-site	nodes removed w/o call-site
checkstyle	4046.4	2404.4
doctorj	27.4	26.4
findbugs	45223.2	23007.8
jaranalyzer	33.8	22.6
java2html	3531.2	440.4
jruby	570087.6	46527.2
jython	14643.8	2621
pmd	1	1

Table 2.5: Average total nodes excised using CCTs with and without call-site information. Numbers are averaged for each application over five inputs.

The trade-off that we make for this increase in detail (and thus understanding of program behavior) is in the CCT size. In Table 2.6, we quantify this overhead for both revisions of our applications. Columns two and three show the CCT size as the average

App. Name	nodes w call-site	nodes w/o call-site	difference (%)
checkstyle_old	1448937.6	385136.8	276.21
checkstyle_new	1449183.6	385294.0	276.12
doctorj_old	390814.2	273867.6	42.70
doctorj_new	390816.4	273868.8	42.70
findbugs_old	185960.2	120527.6	54.29
findbugs_new	183552.2	118694.6	54.64
jaranalyzer_old	569.6	462.0	23.29
jaranalyzer_new	565.0	459.4	22.99
java2html_old	2886.6	344.8	737.18
java2html_new	1075.4	265.2	305.51
jruby_old	321365.2	25902.6	1140.67
jruby_new	304357.6	32369.4	840.26
jython_old	145700.4	28075.6	418.96
jython_new	145896.2	27695.4	426.79
pmd_old	676247.4	527235.6	28.26
pmd_new	676248.4	527236.6	28.26
			Avg = 294.93%

Table 2.6: Comparison of CCT sizes with and without call-site information.

number of nodes over five inputs for each application, with and without call-site information. The fourth column is the percent increase in CCT size due to using call-site information. The old revision of jruby shows the highest increase while jaranalyzer's new revision shows the lowest. The average increase is nearly 300%. From our experience, recursion is the primary reason for CCT size explosion when call-site information is included. To reduce the CCT size, a threshold can be placed on its depth which can be tuned based on the amount of tree comparison detail required.

2.6 Related Work

In [7, 60, 65, 71], algorithms for syntactical, semantic, and structural comparison of software versions are proposed. All of these approaches, however, operate statically. This is different from our approach, since we rely on dynamic profile (CCT) generated by test runs of the application. Relying on dynamic profile can expose unforeseen effects of code modifications that are hard to identify using only static analysis. Our approach thus complements these efforts.

Zhang et al. propose a technique to match entire execution histories of two program versions running with the same input [124]. The execution history contains control flow taken, values produced, addresses referenced and data dependences. This is different from our technique since these prior works assume semantically equivalent versions (e.g. optimized and unoptimized) while we compare different revisions of a program that can include functional upgrades.

The work most similar to ours is described by Zhuang et al. in [126]. They have developed a framework for comparing CCTs of the same program when running on different platforms or with different inputs. The CCTs they used, however, do not include call-sites which keeps the CCT size reasonable and enables them to use the tree transformation algorithm proposed in [2] to perform the comparison efficiently. While this approach is useful to quantify the difference in execution on different platforms

or when using different inputs, it is not suitable for comparing functionally different versions of the program as information gets blurred in the CCT. Furthermore, due to the nature of the tree transformation technique they adopt, the nodes matched from both trees are not necessarily semantically equivalent. We have discussed this limitation further in Section 2.2.2.

Our work is the first, to our knowledge, to focus on revision-based dynamic behavior and performance differences with support of source code repository systems.

2.7 Conclusion

In this chapter, we present PARCS, an offline analysis tool that automatically identifies differences between the execution behavior of two revisions of an application. PARCS collects program behavior and performance characteristics via profiling and generation of calling context trees (CCTs). We annotate CCTs with call-site information and performance metrics to facilitate identification of differences in CCT topology (changes in the calling patterns of the program) and in overall program performance (via weight differencing). We overview our techniques for identifying differences in CCTs and demonstrate how we use PARCs to attribute differences in execution behavior and performance to specific changes in the source code.

We have presented an empirical evaluation of PARCS using a number of well-known Java applications. We present what supports the use of call-site information to expose additional topological differences than conventional CCTs. We also quantify topological and weight differences between two revisions of each application. Moreover, we developed a scoring metric to assess matching accuracy and have shown that PARCS is best applied to revisions comparison to track and gain a better understanding of how software updates impact overall behavior and performance.

Overall, we find that PARCS is most effective for incremental changes such as those common to revisions. As such PARCS has the potential for facilitating improved understanding of the behavior and performance of complex software systems and their evolution over time.

Chapter 3

Dynamic Scripting Languages

In the previous chapter, we studied performance understanding across revisions with focus on Java programs. In the following chapters, we investigate techniques to improve performance of a popular class of managed languages: dynamic scripting languages. This chapter provides background on these languages, their sources of dynamism and how they are different from Java. It surveys and contrasts possible implementations and discuss the shortcomings of each. It also overviews recent advances in virtual machines for these languages and state-of-the-art optimizations.

3.1 Introduction

Dynamic scripting languages (DSLs) such as Python, Ruby, PHP, and JavaScript have experienced rapid uptake in the commercial sector for software development and general widespread use in recent years. The popularity of these languages stems from

two factors: 1) rapid prototyping through high level syntax, language flexibility and dynamism, and a large collection of frameworks and libraries, thus enhancing programmers' productivity, and 2) cross-platform portability – interpreters and runtime systems are available on a large number of systems. These features make it easy for non-experts and experts alike to employ these languages quickly for a wide variety of tasks and to deploy fairly complex applications on a variety of platforms.

Many of these languages were originally designed for "scripting", e.g., for writing short programs for text processing or as "glue code" between, or embedded within, modules or components written in other languages. Recently, they are increasingly employed for more complex, general-purpose, and self-contained applications. For example, Python is used in Linux for software packaging and distribution, for peer-to-peer sharing [20], computer aided design [45, 89], cloud computing [52, 27], as well as computationally intensive tasks [95]. PHP and JavaScript are commonly used for server-side and embedded client-side scripting, respectively; Google uses JavaScript for their desktop applications, including GMail and Google Docs, while PHP is the most commonly used language for writing wiki software [121]. Development frameworks (e.g. Rails for Ruby and PHP (TRAX), Django for Python, etc.), high level syntax, and language dynamism all play a key role in the use of dynamic languages for increasingly complex applications.

In the rest of this chapter, we overview the dynamic features that distinguish DSLs from other languages. We also survey and contrast methods of implementing them in addition to recent optimizations to improve their performance.

3.2 Dynamic Features

Although statically-typed languages (e.g. C, C++ and Java) provide some dynamic features, such as switch statements, function pointers and virtual functions, these features are either explicit (declared by users) or limited in scope. For example, function pointers and virtual functions in C++ are explicitly declared by the user (e.g. the `virtual` keyword in C++). In Java, although all methods are implicitly virtual, the set of possible targets for any method call is limited to object type and super-types only. This strict enabling of dynamism is essential for efficient execution of these languages in two ways. First, it enables the compiler to generate statically bound code based on static analysis and/or profiling. For example, in Java, a compiler can deduce that a virtual call has only one possible target by inspecting the inheritance chain. Second, even if static binding is not possible, the generated generic code is of low overhead. For example, the use of virtual method tables in Java enables efficient dynamic dispatch of methods.

On the other hand, DSLs dynamic features are pervasive and implicit throughout the language. Although this makes these languages quite appealing to developers for their brevity and high productivity, it makes them quite challenging to optimize. In this section, we overview these dynamic features.

3.2.1 Dynamic Typing

DSLs are most commonly known for being dynamically typed. Dynamic typing means that types are associated with values while variables have no type. Therefore, at runtime, any variable can hold a value of any type. In other words, a variable is merely a pointer that can point to data objects of arbitrary types at runtime. This feature is the building block for other capabilities of DSLs. It makes it possible to generate new types and associate methods to them at runtime. In fact, in Python, all classes are built on the fly. Furthermore, unlike Java and C++, it permits dynamic dispatching of methods across all types in the system, not just types belonging to the same inheritance chain. This makes DSLs less susceptible to static analysis and optimizations in the following manners.

- **Type Safety**

  A language is type safe if it guarantees that types are accessed and used correctly during execution. In statically-typed languages, this is mostly established statically at compile-time with few type checks performed at runtime. Static

type-checking helps avoid type-errors early before running the program. Establishing static type-safety for DSLs relies on type inference [125, 47, 84] which is the deduction of variable types at certain points in the program based on how they are used. Type inference is quite complex to achieve for DSLs because of the dynamic features and meta-programming constructs [6, 46]. For these languages, type-safety is guaranteed at runtime and when a type-error is detected, the program reports an error and terminates. Although still type-safe, in the sense that execution will not proceed with faulty types, type-errors are discovered late at runtime. Additionally, the excessive type-checks required at runtime poses a performance overhead.

- **Performance**

  Dynamic typing also hinders optimization efforts of static compilers, since without any type information, a static compiler will have to generate generic code to handle all possible types combinations. Such code will have to carry out repetitive type-checks, indirect function calls and hash-lookups to resolve types.

3.2.2 Dynamic Objects

In static languages, a class declaration define the layout for all instantiations of the class. It contains a blueprint of all its objects given in the set of methods and attributes that each object will have. The key rule is that every field resides at a fixed offset from

the object start across all objects of that class. With the object layout known in advance, the code needed for field access is a mere load from a fixed offset from the object start.

In DSLs objects are dynamic. New attributes can be added or deleted at run-time. With a variable unknown set of attributes per object, a compiler cannot assume a fixed offset for a certain object attribute and therefore cannot load attributes efficiently. Objects then are more like a map (dictionary) where attribute names are mapped to values. In Python, a class is usually declared with no attributes; they are added to objects after instantiation. There is no guarantee on the order by which attributes are added and therefore objects of the same class do not necessarily have the same layout. In JavaScript, there are no classes altogether. Every object is a mere map between attributes' names and values. This significantly complicates code generation for field access. Instead of a simple load from a fixed offset, the code is now required to look up an attribute by name in the object map. If the attribute is not found, additional lookups are necessary in the object's type and super-types. Thus, instead of a single instruction for field access, loading an attribute becomes a sequence of successive hash lookups. This is clearly an expensive operation and is one of the major overheads of DSLs that is target for optimizations.

3.2.3 Meta-Programming

Meta-programming, or reflective programming, is the ability to modify the program structure and behavior at runtime. Appending new methods/attributes to an already-created class or creating new ones on-the-fly, intercepting attribute access and modifying it, and constructing source-code at runtime and executing it (e.g. the *eval* function in Python and Ruby) are all examples of meta-programming. Such functionality is not unique to dynamic scripting languages; they can also be found in Java and other languages. DSLs just make them more pervasive, available and easier to use. In DSLs everything is an object. That includes classes and methods. Methods of a class can be accessed as normal attributes. They can be modified, added and removed. Therefore, modifying a class method is a mere assignment to the method's entry in the class dictionary. Meta-programming, however, is rarely used in practice past initialization of classes.

A work by Holkner et al. [55] aims to understand the extent and scope of use of dynamic language features like runtime object and code modification. In particular, the authors examine whether Python programs only rarely use dynamic features, and whether this use of dynamic features is restricted to an initial startup phase in the application. For the programs and the set of dynamic features that they analyzed, the authors conclude that while programs do make use of dynamic features over the entire

execution, this use is relatively higher during startup, thus lending themselves well to runtime analyses and feedback-directed optimization.

3.3 Implementations

In this section, we briefly overview possible implementation for DSLs and the advantages and shortcomings of each.

3.3.1 Interpretation

Interpretation is the simplest and most straight-forward implementation of any language. An interpreter is a managed runtime environment (MRE) that executes instructions of a program written in some programming language. In their purest form, interpreters do not generate native code; all code instructions are carried out by the interpreter code. They may, however, offer a native implementation of the language's common functions and operations as part of the interpreter code itself (its runtime). Interpreters follow a lazy-model of execution where code is loaded only when needed. Interpretation can proceed in different ways.

1. **Interpretation from source:** In this form, an interpreter operates directly on the program source code. The interpreter includes a compiler front-end that will generate, and possibly optimize, an Abstract Syntax Tree (AST) from the source

code. AST generation happens for every source code file upon loading. The interpreter will then traverse the AST and execute code matching each node semantic. Executing each AST involves work done by the interpreter itself or through calls to its runtime and libraries. Ruby-1.8 [94] standard interpreter follows this form.

2. **Translation to IR:** Similar to interpretation from source, these form of interpreters include a simple compiler that will translate the source code to some Intermediate Representation (IR). The IR is usually a high-level linear representation of the source code. IR instructions are usually referred to as bytecodes. Bytecodes are designed to run on a virtual machine that can be either register- or stack-based. It is the interpreter's task to emulate and maintain the state of the virtual machine during bytecode execution. Python [88] standard interpreters are stack-based while the Parrot virtual machine interpreter [79] is register-based. Stack-based bytecodes have the advantage of being compact in size and easier to interpret while register-based bytecodes are faster to execute and are closer in form to native code which makes them easier to lower (compile to native code) as needed.

3. **Pre-compiled code:** Interpretation from pre-compiled code is similar to translation to IR except that the translation steps happen independently and before interpretation. This model is used for the Java programming language. A main

disadvantage this approach is the need to compile the entire source code first before execution. A step that slows down development cycle.

Interpreters are significantly slower than compiler-based implementation. There are several reasons for that. First, at the core of any interpreter is a dispatch loop which is responsible for reading and executing bytecodes. Bytecodes are executed either as part of the dispatch loop body or via calls to the runtime. In its simplest form, a dispatch loop is a loop over a large switch-case statement which matches a bytecode to its handler. This form of dispatching can pose a performance bottleneck because of the switch branch which is difficult to predict by branch target predictors. Second, bytecodes are dispatched and executed independent from each other. This prevents across-bytecode optimizations. Third, bytecode execution is carried out by shared generic interpreter code which prevents per-bytecode specialization. Although some of these drawbacks have been addressed in previous research [93, 83, 39, 123], interpreters still remain by far slower than compiler-based implementations.

Despite their inherent performance inefficiency compared to compilers, interpreters remain a popular choice for languages' implementations specially for DSLs. This stems from the following advantages:

1. **Flexibility:** Because of their simplicity, interpreters are easy to design, implement, maintain and modify. This makes an interpreter a flexible solution for lan-

guage implementation. Given the fast rate by which DSLs change in semantics, interpreters are favored for their implementation.

2. **Startup:** Compilation-based MRE invest startup time compiling and optimizing code on the assumption that the code will run long enough for the startup cost to be amortized. Interpreters do not incur such cost and thus provide better responsiveness and startup time. For that reason, interpreters are used along with compilers in mixed-mode MREs. A mixed-mode MRE will initially run the code over an interpreter until the code is executed frequently enough. When the code gets "hot", it is then compiled and optimized.

3. **Development cycle:** For interpreters that do not rely on pre-compiled code, the code is run directly from source. This saves time during development as the developer does not need to wait for the code to compile after every source code change.

4. **Low memory footprint:** Interpreters incur less memory footprint than compilers, since they do not need all the internal data structure for parsing, optimizing and generating code. Also, bytecode is much more compact than native code.

3.3.2 Just-in-time Compilation

Just-in-time (JIT) compilation, or dynamic translation, is a middle-ground between interpretation and static ahead-of-time compilation. It relies upon dynamically and lazily (re-)compiling the most-executed parts (hot code) of a program as it executes. JIT compilers are usually employed, with an interpreter, in a mixed-mode execution environment [69, 111, 9, 23]. Execution starts slow in interpreted mode until hot code is identified for compilation. Since compilation happens at run-time, JIT compilers can benefit by collecting profiles about the running program during the interpretation phase which guides code optimizations such as global code motion, inlining, unboxing and de-virtualization. Code optimization then becomes an adaptive process where optimization decisions are based on the observed past behavior of the program and with the assumption that this behavior will continue in the future. Deviation from the assumed behavior must be detected, via guards, and may demand de-optimization of the code and re-compilation.

JIT compiling for dynamic languages is, in principle, no different than that for static ones. The sources of overhead in dynamic languages implementations, however, are different from static ones. This demands additional optimizations, and profiles, to achieve better performance. For example, dynamic variable types require unboxing and type-specialization. Without them, the generated code will be type-generic, with all the

unboxing and type-checking involved. Also, information about object layouts can be used to cache attributes instead of expensive hash lookups.

Several JIT compilers for dynamic languages exist with different stages of completion. TraceMonkey [50] is a trace-based compiler for JavaScript that focuses on detecting hot traces in loop bodies and optimizing them. Once a loop header becomes hot, TraceMonkey starts recording the trace taken in the loop and the types seen by every bytecode. Typed LIR instructions are emitted for every bytecode along with guards to verify that the control-flow and types are identical to the ones recorded.

A similar approach is followed in PyPy [92], a specializing JIT compiler for Python. PyPy traces Python programs at the meta-level. A Python interpreter written in a restricted statically typed subset of Python (RPython) runs over the PyPy virtual machine. PyPy traces the interpreter dispatch loop and compiles traces of it. Since PyPy can compile any language, provided it is interpreted with an interpreter written in RPython, it can be looked at as a JIT generator that relies on Partial Evaluation techniques [49, 61]

Psyco [91] is a method-level just-in-time specializing compiler for Python. It relies on just-in-time specialization where the optimizer communicates closely with the interpreter during execution to query about variable types and values.

Shortcomings

Although JIT compilers have the advantage of adapting to the program behavior and compiling only necessary code, they still have their own shortcomings.

1. **Complexity and maintainability:** The most important issue with JIT compilers is the amount of complexity involved. Advanced techniques such as background compilation, de-optimization and On-Stack Replacement (OSR) [43] makes developing and debugging a JIT compiler difficult. The issue becomes more emphasized for dynamic languages where the language specifications change from one interpreter release to the next. A change that a JIT compiler has to cope with and mimic, which makes maintainability hard.

2. **Limited optimization budget:** Since optimizations happen at runtime, they account for execution cycles where program execution is paused, compilation takes place then execution resumes again. This cycle can take place multiple times depending on when hot regions of the program are discovered. This presents a trade-off between code quality and compilation time. Background compilation provides a solution to this by overlapping compilation with actual execution, but it demands multi-core architectures and comes with additional engineering complexity. Therefore, highly optimized generated code does not necessary translate to better performance. The optimized code has to run long enough to amortize

the cycles spent in optimizations. This is generally the case for server-side programs. In the domain of client applications, however, programs do not run long enough to compensate for the optimization cycles. Also, client applications are usually GUI-based, which means they should be highly-responsive to the user actions. Applying advanced optimizations early during execution is more likely to hurt performance and/or user-experience.

3. **Limited profiling budget:** For the same reason, there is a limit on the granularity of profiling information that can be collected for a JIT compiler to use. The more detailed the profile is, the higher the overhead and the longer the program needs to run after optimization to amortize the cost. A value profile, for example, is expensive to collect online.

4. **Startup-cost:** This is the combined overhead of the initial interpretation phase, profile collection and compilation. Every time an application is started, it suffers from this slow phase until optimized code is executed.

3.3.3 Ahead-of-time compilation

Ahead-of-time (AOT) or static compiler is a compiler that translates from one language to another ahead of any execution of the code. AOT compilers are generally simpler to implement than JIT compilers. Functionalities such as de-optimization, On-

Stack Replacement and others are not needed. Also, AOT compilers have an unlimited compilation budget, which makes engineering optimizers much simpler. This makes AOT compilers easier to develop and maintain. This relative simplicity, however, comes at the cost of certain drawbacks.

1. **Lack of real-use profiling data:** Lack of profile data may be acceptable for static languages such as C or for languages where dynamism is explicit (e.g. virtual functions in C++) which makes these languages more prone to static analysis. Profile data becomes more crucial to performance as more dynamism is inherent to the language. For example, inlining and de-virtualization can be more effective when they rely on profiling data for languages like Java, Python, Perl and others, where nearly all calls are virtual. More dynamism such as dynamic typing, objects and meta-programming language features makes static analysis techniques even more limited and in need of profiling guidance [6, 46, 48, 5, 47]. Obviously, AOT compilers can rely on profiles generated during development as a performance tuning step, but the profiles generated come from artificial test-inputs and are not representative of how the program will be used in the wild. Other techniques, such as annotations of the source code [29], are possible but, are tedious and error-prone. They defeat the purpose of the rapid and flexible development model that DSLs provide.

2. **Excessive code generation:** JIT compilers follow a lazy incremental compilation model where parts of the code are compiled when they become hot. This per-user adaptivity is not present in AOT compilers. AOT compilers compile either the entire program or parts that the developers think are worth compiling for performance reasons. Thus, more code is generated than what will be actually needed.

3.4 Optimizations

In this section, we briefly overview optimizations that are particularly useful for DSLs. Due to the dynamic nature of DSLs, resolving attributes, calling functions or even performing simple arithmetic operations require generic code that includes hash-lookups and indirect calls. These optimizations are based on the same principle: find a fixed frequent pattern of execution and make it faster. For a pattern to be true, certain constraints about the variable types and values must hold. The constraints are either proven to be true at compile-time or they are observed during runtime profiling. For the former case, any optimization based on those constraints is always true and the optimization is permanent. For the latter, the optimized code has to be "guarded" by checks to ensure that the constraints have not changed. An invalidation scheme may be needed if the constraints fail repeatedly.

3.4.1 Specialization

Specialization in DSLs can be based on variables types, values or both. An example of a classical simple form of compile-time value-based specialization is constant folding. If all the input values to an expression are constants, then the expression can be evaluated (specialized) statically. The resulting value can be used directly in the code thus saving the runtime cost of evaluating it.

Similarly in DSLs, knowledge of variable types and values help generate specific code for them that is faster than the generic code. For example, consider an add operation on two objects. The generic code for adding them will typically resolve the add operation on one of the objects, then invoke it. If the object types are known to be integers(either via static type-inference or dynamic runtime feedback), then a type-specific version of the code will bypass the operation lookup and invocation steps and perform integer addition directly.

Type and value specialization are adopted in production and research MREs for DSLs. Psyco [91], a Python MRE, uses partial evaluation [61] to generate type- and value-specialized clones of Python functions at runtime. TraceMonkey [51] is a JavaScript trace-based MRE that performs type-specialization over dynamic traces of execution.

3.4.2 Caching

Because of the dynamic nature of objects in DSLs(Section 3.2.2), unboxing of an object (i.e. accessing an attribute of an object) is an expensive operation requiring multiple hash lookups in the object, its type and super-types. This overhead is necessary for correctness since an object can change shape during execution. Caching is a sort of specialization that allows fast access to attributes. The idea is to maintain a cache of attributes that is referenced by the receiver object type. A cache invalidation mechanism is usually needed to detect cases where the object type or shape changes.

A global caching scheme is used in SmallTalk [33] to speedup method lookups. The cache is indexed by a pair of the receiver type and the method name. This approach is taken a step further by inline caching [24, 56, 33]. It is an optimization that reduces method lookups by linking methods based on the receiver type. The optimization is based on the observation that receiver object type is almost invariant at a particular call-site. Instead of doing expensive method lookup each time a call-site is executed, the lookup is performed only at the first execution to find the address of the target method. The call-site code is then patched to call the resolved target method address directly. Since the type of the receiver object may change in subsequent executions of the call-site, a guard is inserted in the callee prologue to ensure that the receiver object is always the one expected. If the guard fails, method lookup proceeds in the slow path and the call-site code is re-patched.

Hidden classes is a caching optimization for JavaScript used in Google V8 engine [117]. Since objects are prototype-based in JavaScript, the optimization associates a hidden class with each object. The class is simply a table mapping attribute names to offsets in the object. Every object points to a class such that all objects of the same shape always point to the same hidden class. At every attribute access code position, the code is patched such that the attribute offset is cached and is used to access the attribute directly (without lookups) in the object. The patched code is guarded by checking the receiver object hidden class against the expected value. If there is a mismatch, the slow look up path is taken and the code is re-patched.

3.4.3 Inlining

Inlining is a classical compiler optimization that replaces a call-site with the actual body of the call-target. It can improve performance by eliminating the cost of the method call/return instructions, arguments passing and saving/restoring register. It also facilitates across-methods optimizations and may improve code locality. While inlining is aggressively applied in static languages (e.g. C/C++, Java), it is more challenging for DSLs. Since the call-target is not known until runtime and may vary during execution, inlining is only possible with a method linking mechanism (e.g. Caching). The inlined code has to be guarded to ensure that it is indeed of the call-target body.

Chapter 4

Understanding the Efficacy of Interpreter Dispatch Optimization on Modern Architectures for Dynamic Scripting Languages

In this chapter, we present an empirical evaluation and performance analysis of the standard interpreted implementation of Python (CPython) using multiple architectures (IBM Power, Intel Core 2, and Intel Xeon systems) and a variety of metrics. We also investigate the efficacy of interpreter dispatch optimizations developed in the context of statically-typed languages, such as Java, when applied to Python. We find that techniques that reduce the overhead of interpreter dispatch are less effective for Python than for Java because dispatch accounts for a smaller portion of the total runtime overhead. That is, the Python runtime does more work per bytecode, due to the dynamic nature of the Python language, than a Java runtime, which in turn impacts the efficacy of optimizing dispatch. We employ a number of different benchmarks and programs in

both Python and Java and show how architectural differences impact the performance potential of dispatch optimization.

4.1 Introduction

Researchers and open source efforts continue to pursue ways of dynamically compiling dynamic scripting languages [91, 92, 109, 117, 53, 50]. However, popular versions of the distributed runtimes for these languages still employ interpreters for execution [28, 94, 81, 82]. Interpreters play a key role in the wide-spread use of these languages. This is because of certain advantages they have over compiler-based runtimes: They allow a shorter edit-compile-test loop and hence faster development. They are also easier to build which facilitates rapid prototyping of new language features and permits flexible language evolution. Moreover, the runtimes are architecture-independent (e.g., written in C and compiled with existing tool-chains) and can be compiled to any system without significant porting. Finally, dynamic languages are particularly challenging to compile because variable types are not known statically and must be inferred and speculatively translated.

Interpretation, however, is an inefficient execution mechanism. The interpreter must decode and dispatch each bytecode, incurring an overhead not present if executing compiled codes. Moreover, runtime resolution of variable types, dynamic objects and re-

flection capabilities confer to these languages a strong dynamic aspect. This causes bytecodes to be implemented high-level and type-generic, in contrast to bytecodes of statically-typed languages such as Java. Therefore, the bytecodes of dynamic languages tend to require more work for runtime resolution. In addition, interpretation of each bytecode is done independently of other bytecodes which hinders optimizations and results in poor code quality compared to codes produced by even the simplest code generator.

There has been significant research to identify ways of optimizing the interpretation process while maintaining the portability of the runtime [93, 39, 123]. For example, the dispatch-loop can be eliminated with indirect threaded interpretation (ITI), a technique in which each bytecode handler performs the decode of the next bytecode instruction and a jump through a lookup table to the next handler. Direct threaded interpretation (DTI) extends this process to avoid the decode step – each bytecode opcode is replaced with the address of the handler. Both of these optimizations reduce the overhead of conditional branches and indirect branches, which are typically hard to predict and thus costly on modern architectures. Such techniques have been proven quite effective for simple interpreters such as OCaML as well as for Java [39].

In our work, we are interested in understanding the performance characteristics of interpreted, dynamic languages on modern architectures. In this study, we focus on the Python programming language and evaluate the performance of the Python inter-

preter on multiple architectures (IBM Power, Intel Core 2, and Intel Xeon systems). In addition, we investigate the efficacy of popular, yet portability-preserving, interpreter optimizations such as ITI and DTI. We find that such optimizations have little benefit for Python programs despite their significant gain for Java [38].

We investigate the fundamental reasons behind this observation by comparing the execution of the Python and Java interpreters using a variety of metrics. In a nutshell, we find that because Python is dynamically typed, the runtime performs significantly more work per bytecode than Java. The Java bytecodes which contain type information are lowered to machine instructions much quicker. We quantify these differences via time spent by the different runtimes as well as with the bytecode dispatch rate. We also present differences in branching behavior and how the branch predictors of modern hardware (in simulation and using real systems) impact the efficacy of these interpreter optimizations.

4.2 Background

Interpreters are the main focus of our evaluation. In this section we discuss existing interpretation methods and optimizations related to streamlining interpreters, mainly dispatch optimizations. We also present some of the characteristics of the Python interpreter used in the evaluation.

4.2.1 Interpreters

There are three common ways to implement computer language interpreters:

1. Interpret from source code directly by converting it into an Abstract Syntax Tree (AST) which is then traversed for execution.

2. Compile the source code on-the-fly to an internal linear representation of operations (bytecodes) which are then interpreted one at a time.

3. Compile the source code to bytecodes prior to execution and interpret the already pre-compiled bytecodes.

The design of the intermediate representation (IR) used by an interpreter (e.g. AST, bytecodes) is affected by a number of factors including flexibility, portability, simplicity, optimization opportunity and features of the source language. Overviews of interpreters and threading dispatch techniques are available in [32] and [64].

Interpreters for statically typed languages benefit from the knowledge of the types of each operation at the time the methods are initially parsed. A language with a simple fixed type system can represent the type of every operation in the IR. Moreover, many of the statically typed languages have native types that are close to the architecturally supported types. For such operations, the interpretation of the IR is straightforward and cheap. Type-specific operations can be implemented as statically bound calls to the corresponding native methods (e.g. IntegerAdd, FloatAdd, ... etc.). Statically typed

languages naturally gravitate to lower-level representations to perform more work earlier and, potentially, for more safety and a more compact representation.

Interpreters for dynamically typed languages necessarily do not know the binding for each operation until instantiation and invocation at runtime. The IR is limited to the abstract operation whose specific implementation is determined (e.g. using function pointers) at runtime. Interpreters for dynamically typed languages must perform more work at runtime to resolve operand types and route the operation to the correct implementation. This coarser high-level IR causes the interpreter to perform more complicated operations that lead to more time in the natively-compiled code of the runtime compared to the dispatch loop.

Statically typed language interpreters can be constructed with a similar high-level IR, trading off work at different phases of the interpretation cycle or ahead of time computation versus runtime computation. One recent analysis of Java interpreter using Abstract Syntax Trees (AST) instead of bytecodes shows the benefits of preserving more semantic information in the high-level representation [54]. The coarser granularity of statements has the advantage of reducing the dispatch overhead (more work done per every bytecode dispatch).

Java bytecode operates on a stack machine and is strongly connected to the eight primitive data types supported by the language plus object references. Java uses its bytecode format as an external representation communicated to and executed by the

virtual machine. The Java Virtual Machine either interprets the bytecode or further compiles it into machine code using a JIT.

Python bytecode operates on a virtual stack machine as well, but, unlike Java, it must defer associating abstract operations with specific implementations until runtime because variable types are not known earlier. This property of the language motivates a higher-level and more abstract bytecode representation. Because type decisions are deferred until runtime, Python provides many more opportunities for type-based method specialization. A number of projects are focused on improving the execution of Python programs through compilation [91, 92, 62, 59, 113].

4.2.2 Dispatch optimizations

The overhead of bytecode dispatching can be improved in two dimensions. The first dimension involves reducing the overhead of every dispatch. The second dimension focuses on reducing the overall number of dispatches.

Threaded interpretation addresses the first dimension. The basic dispatching technique is switch-based as depicted in the code on the left in Figure 4.1. This implementation consists of a dispatch loop that iterates over each bytecode, branches to one of the enumerated labels to implement the operation (via a call or inlined), and loops back to dispatch the next bytecode. A major disadvantage of this approach is the use of one shared indirect branch for dispatching. Performing accurate target prediction

for that particular branch is quite difficult due to the random order by which bytecodes are dispatched and executed. This leads to poor branch target prediction on modern architectures [39, 123].

Switch-Case:

```
while((inst!=null)
{
 opcode = getOpcode(inst);
 switch (opcode){
  case opA:
    opA_handler(inst);
  break;
  case opB:
    opB_handler(inst);
  break;
  …
 }
 inst = getNextInst(inst);
}
finish();
```

Indirect Threading (ITI):

```
 inst = getFirstInst();
 if (inst==null) finish();
 opcode = getOpcode(inst);
 handler = handlers[opcode];
 goto *handler;
 …
 OPA_LABEL:
    … /* implement opcode A */
    inst = getNextInst(inst);
    if (inst==null) finish();
    opcode = getOpcode(inst);
    handler = handlers[opcode];
    goto *handler
 OPB_LABEL:
 …
```

Direct Threading (DTI):

```
 inst = getFirstInst();
 if (inst==null) finish();
 handler = getOpcode(inst);
 goto *handler;
 …
 OPA_LABEL:
    … /* implement opcode A */
    inst = getNextInst(inst);
    if (inst==null) finish();
    handler= getOpcode(inst);
    goto *handler;
 OPB_LABEL:
 …
```

Figure 4.1: Types of interpretation.

One technique to improve performance is indirect threading [106] (ITI), as depicted in the middle example in Figure 4.1. This technique eliminates the dispatch loop; each opcode handler implements decode of the next instruction in the execution stream, uses the opcode to index into a look-up table to extract the address of the next opcode handler, and jumps to the address extracted. This approach increases the code size of the interpreter since the decode and table look-up work is replicated by each handler.

ITI can be optimized to eliminate the table look-up using direct threading (DTI) (right-most example in the figure). DTI replaces the opcode of every instruction (upon first execution or ahead-of-time) with the address or label of the opcode handler [16]. This replacement increases the size of each instruction and still requires replicated code

for decoding at the end of each handler. Both ITI and DTI have improved dispatch overhead over the switch-based technique for two reasons. First, for every indirect branch, the number of possible targets decreases, which enhances target prediction. Second, any biased relation between opcodes is exploited (e.g. in Python a `JUMP_IF_FALSE` bytecode is usually followed by `POP_TOP`).

Another related technique is double indirect threading [34], which uses a second level of indirection — storing pointers to indirect blocks that contain pointers to simpler, common operations and operands. In the DTI example in the figure, the body initiated by `OPA_LABEL` is replaced with multiple indirect calls to other labels, each implementing part of the operation. This technique frequently was used in implementations of the FORTH language [44].

Context or Subroutine Threading [18] further adjusts the design to make all bytecode bodies callable and then translates the virtual code into a sequence of static calls to the bodies. Each instruction body is terminated by a return statement (the goto in each handler body in the DTI example in the figure is replaced by a return instruction); the return acts as the next-opcode dispatch. The reason this behaves better is that jumping to the opcode handler is static (no branch prediction is needed). Also, return address prediction is more accurate (since it relies on a stack of return addresses) than indirect branch target prediction [38].

Replicating bytecode handlers, either statically or dynamically, is another approach for reducing the randomness of the dispatch branch target and improving target-prediction accuracy. This technique relies on linking different instances of the same bytecode to different bytecode handlers. Thus reducing the number of possible targets for every indirect branch.

Techniques along the second dimension (reducing the overall number of dispatches) include selective-inlining [83] and super-instructions [38] formation. In selective-inlining, traces (sequences) of commonly executed instructions are combined in one super-instruction dynamically, and the bytecode stream is modified to use these super-instructions instead. This amortizes the dispatch overhead for frequent traces. Super-instruction formation is similar to selective-inlining except that it is done statically. Bytecode sequences that the interpreter developer expects to be common are grouped into super-instructions. The interpreter performs bytecode-rewriting to use the super-instructions whenever possible.

4.2.3 CPython VM

There are several implementations of the Python language [91, 92, 113, 59, 62, 110]. In this work, we focus on the most widely used: CPython [28]. The CPython implementation is also simple, portable and easy to understand and extend. CPython employs switch-based interpretation and simple reference counting garbage collection.

Because in Python all data types are objects, CPython maps all Python data types to C structures with a common header that contains the reference count, among other things.

CPython implements 113 different bytecodes (42 with arguments and 71 without). These bytecodes are high-level (generic) to support dynamic typing. For example, `LOAD_ATTR` and `STORE_ATTR` operate on generic objects to load/store a field given its name. The interpreter performs these look-ups by name using a dictionary data structure. Another example of a popular Python bytecode is `BINARY_ADD` which also operates on generic objects; it first checks the type of its operands. For Integers, the bytecode performs integer addition on the operands; in the case of overflow the interpreter expands the object from Integer to Long. For Strings, the interpreter performs concatenation. For all other operand types, the interpreter calls a generic Add method. By comparison, Java has typed bytecodes, such as `iadd` that directly map to an integer add machine instruction.

4.3 Methodology

To understand the behavior of Python programs and the impact of interpreter optimizations on CPython, we evaluate the performance of these programs using a variety of benchmark suites, profiling tools, virtual machines (VMs), and hardware architec-

tures. We first overview the benchmarks and technologies we employ for this study in this section.

4.3.1 Benchmarks

To characterize the behavior of Python programs we use the suite of applications provided by the Unladen Swallow project [113] which are listed and described in Table 4.1. We use default parameters. These applications exercise common activities found in Python programs including parsing and translation, (de-)serializing datasets, and HTML manipulation. Also included in this list is `pybench`, which implements a set of microbenchmarks that exercise low level Python activities including function calls, comparison operators, looping constructs, string manipulation, basic arithmetic, and others [85].

Benchmark	Description
2to3	A Python 2 to Python 3 translator translating itself
django	Django Python web framework building a 150x150-cell HTML table
pickle	Uses the cPickle module to serialize a variety of datasets
pybench	Runs the standard Python PyBench benchmark suite
spitfire	Uses the Spitfire Python web template system to build a 1000x1000-cell HTML table
unpickle	Uses the cPickle module to un-serialize a variety of datasets

Table 4.1: The Unladen-Swallow benchmarks

We also characterize the behavior of Python using programs from the Computer Language Benchmark Game (CLBG) [103]. We refer to this suite as the `shootout`

Benchmark	Description	arg
binarytrees	Allocates and traverses many binary-trees	17
fannkuch	Indexed-access to tiny integer sequence	11
fasta	Generate and write random DNA sequences	1000000
mandelbrot	Generate Mandelbrot set bitmap file	200
nbody	Double-precision N-body simulation	500000
nsieve	Counts prime numbers up to M, where M is f(arg) using Sieve of Eratosthenes algorithm	11
nsievebits	A variation of nsieve	11
partialsums	Computes partial sum of a series	5000000
recursive	Uses recursion to compute three recursive functions: ackermann, Fibonacci and tak	8
regexdna	Match DNA 8-mers and substitute nucleotides for IUB codes	A DNA sequence

Table 4.2: Shootout benchmarks and inputs

benchmarks. We describe these programs in Table 4.2 together with the inputs we used. The CLBG provides implementations of the same programs written in different languages. Although the benchmarks are less representative of real-life applications, considering the same programs in Python and Java lends insight to the differences imposed by these languages on the behavior of programs.

We also employ the DaCapo benchmarks when we characterize these differences [21]. DaCapo is a benchmark suite for Java programs that implements workloads that represent real-world workloads and Java program behavior [30]. We use version 2006-10-MR2 in this study.

4.3.2 Virtual Machine Implementations

For this study, we use CPython version 2.6 and use the default implementation as our Python baseline. To measure the effectiveness of DTI on Python, we extended CPython to use DTI for dispatching. Upon first invocation of every method, we widen and replace the opcode of each bytecode in the method with the target opcode handler for the opcode. We store handler offsets instead of absolute addresses for compactness. On dispatch, the interpreter parses the opcode field and computes the target handler address. We rely on the labels-as-variables GCC extension to enable this.

For the comparison between Python and Java we use HotSpot Zero [17] – a portable bytecode interpreter that integrates in to the OpenJDK HotSpot system [75]. Zero supports two types of dispatching: switch-case and indirect-threaded interpretation (ITI). ITI provides adequate insight into the differences between Python and Java and the potential of interpreter optimization because ITI always performs slightly worse that DTI. DTI improves upon ITI by eliminating the memory read on the lookup-table. Moreover, as our findings show, DTI for Python performs significantly worse that ITI for Java.

4.3.3 Profiling Tools

To collect our performance data, we employ a number of different profiling tools. To capture the execution time at the function level on the Intel architecture, we use OProfile [76] with the maximum sampling rate (6000 events). We also collect hard-

ware performance monitor data on the Intel Core 2 and Xeon using Perfmon [80]. We employ Perfmon for this purpose (over OProfile) since it is more flexible and provides us with more control over our broad range of experiments. On the IBM architectures, running the AIX Operating System, we use pmcount (a utility that uses the performance monitoring API to interface with the processors hardware performance counters) to collect hardware performance monitor data .

Next, we modify the CPython VM to gather information about the bytecodes and dispatching. For CPython and Java Zero we collect the average dispatch rate by counting the number of dispatches and dividing it by the number of execution cycles. For CPython we also collect total cycles per bytecode.

4.4 Python VM Characteristics

Using this methodology, we first investigate the performance characteristics of Python. We consider both the behavior of the virtual machine (VM) as well as the bytecode profile of Python programs.

4.4.1 VM characteristics

We make three high-level observations on Python based on the evaluation of our benchmarks.

1. **Most Python programs spend the majority of the time in the VM**.

   That is, applications written in Python, such as those we investigate herein, do more than act as "glue" code between components or modules written in other languages. Figure 4.2 shows the breakdown of time for CPython and Zero. It shows the time spent in the VM that includes the interpreter dispatch loop (VM.LOOP), the VM runtime (VM.RUNTIME), other libraries (LIB) such as any interpreter extensions and C libraries and the operating system (OS). OTHER includes time for other daemons and processes executing concurrently, including the OProfile sampling system. We compute these values using system-wide, time-based sampling. We aggregate samples and categorize them based on the method/module each sample belongs to. CPython (as does Zero) spends the majority of its time in the VM. This indicates that the Python interpreter should be the primary target for optimization.

2. **Python spends less time in the interpreter dispatch loop and more time in the VM runtime than Java.**

   The VM runtime includes runtime support for implementations of individual opcodes (we refer to this component as the language runtime), various C-extension modules, such as regular expression and object serialization extensions, and other runtime support, such as garbage collection and threading. Most implementations of Python bytecode involve calls to the natively-compiled language runtime, such

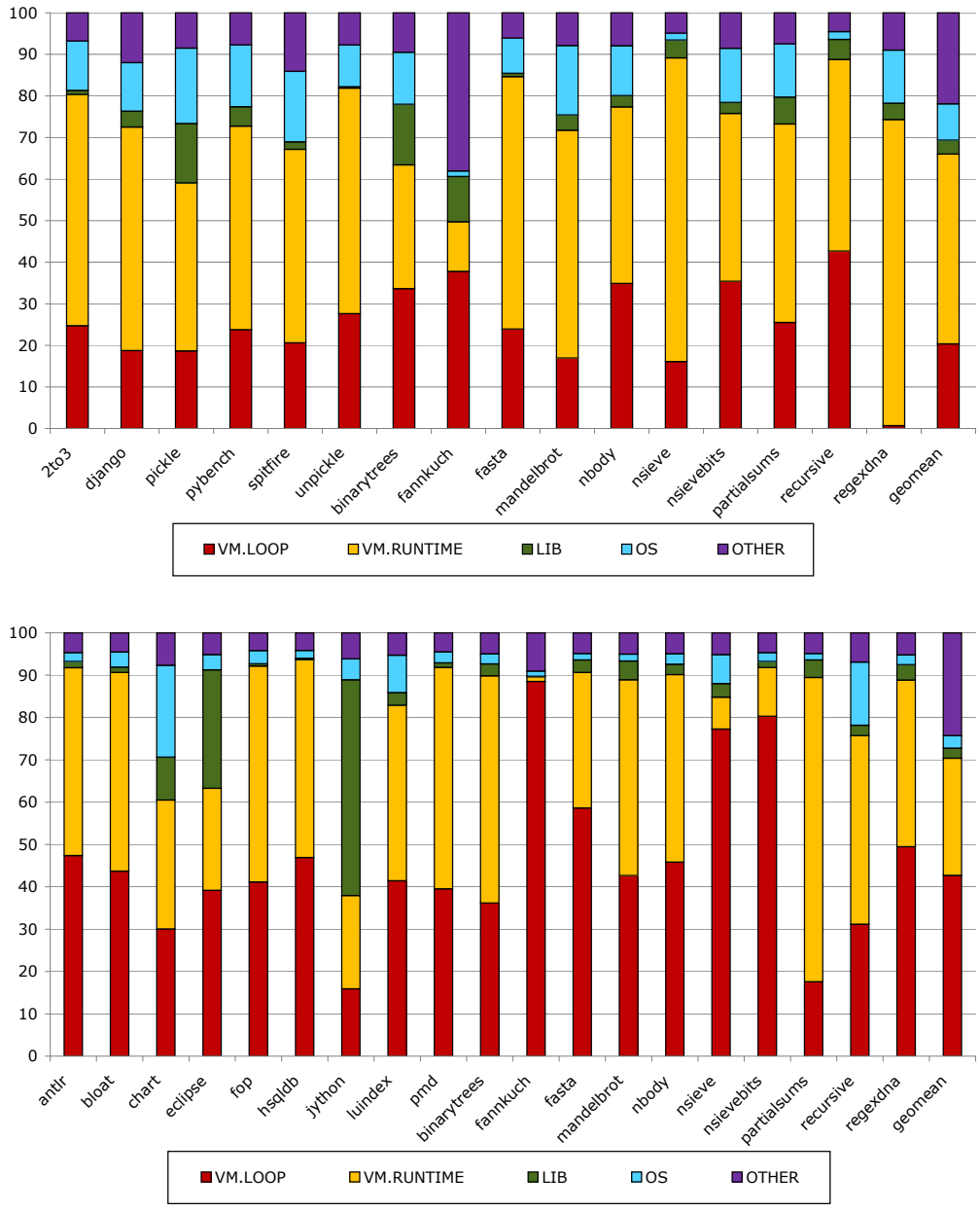


Figure 4.2: Time breakdown for CPython(top) and Java Zero(bottom).

as the call to `PyNumber_Positive` in the `UNARY_POSITIVE` opcode handler shown below from CPython. It dynamically invokes a type-specific handler for the object `v`:

```
case UNARY_POSITIVE:
    v = TOP();
    x = PyNumber_Positive(v);
    ...
```

As shown in Figure 4.2, Python spends significantly less time in the dispatch loop (on average 20%) than Java (average 40%). In one extreme case, `regexdna`, spends only 1% of the time in the dispatch loop, mainly because CPython implements regular expression manipulation via the native `re` module. The benchmark performs DNA matching using regular expressions and has very few dispatches to the `re` module, which performs the actual computation natively. For other programs, the language runtime is the most significant component in the VM runtime. Although `pickle` relies on the `cPickle` module, it does not behave similarly to `regexdna`. `Regexdna` passes a large chunk of data to the native module for processing, while `pickle` works on smaller chunks and frequently calls back into the Python runtime to serialize Python objects.

Since Python spends half as much time in the dispatch loop, optimizations that target the dispatch loop, such as threaded interpretation and simple jitting that leave calls to language runtime intact in the generated code, are likely to be less effective for Python. The significant time spent in the language runtime for Python programs, indicates that there are opportunities for semantic-based optimizations such as type specialization and unboxing.

3. **Python dispatches bytecodes less frequently than Java.**

   Figure 4.3 compares the dispatch rate (per 100 cycles) for Python and Java for the `shootout` programs, indicating a lower dispatch rate for Python than for Java. The trend also holds on average when we add the `unladen` and `DaCapo` benchmarks to the Python and Java suite, respectively.

   At the language level, two features contribute to the lower dispatch rate of Python. First, certain Python bytecodes are more expensive than their Java counterpart due to their type-generic nature. These bytecodes therefore consume more cycles per dispatch. For example, the bytecode `BINARY_ADD` represents the addition of two objects (e.g., adding two integers, two floats or even concatenating two strings), and is much more expensive than the type-specific add operation in Java bytecode such as (e.g. `iadd`) because of the type resolution that is required for the Python bytecode.

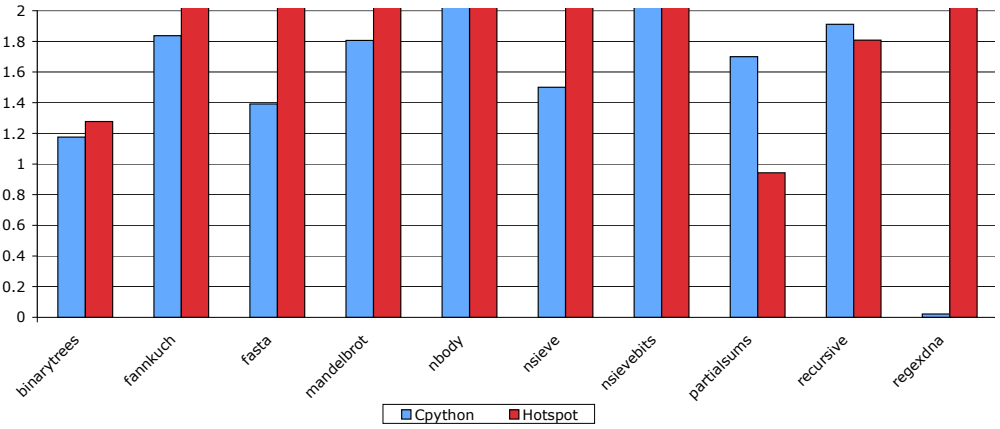


Figure 4.3: Dispatch rate comparison between CPython and Java Zero for `shootout`.

Second, some of the Python bytecodes have built-in semantics. For example, the `PRINT` operations on `list`, `map`, and `tuples` are all built-in. In Java, these operations are supported through other low-level bytecodes, and therefore require more dispatches to achieve the same computation. Python opcodes in this category typically outperform the equivalent Java implementation.

One outlier in Figure 4.3 is `partialsums` where the dispatch rate of Python is higher than that of Java's. A close examination reveals that the Python version uses the bytecode `BINARY_POWER`, while the Java version calls a native method `StrictMath.pow()` to perform the same computation. This `pow` operation executes repeatedly in a tight loop. Both Python and Java require roughly one dispatch per `pow` computation, but the Python version is significantly more efficient. As a result, the Python version is twice as fast as the Java version, leading to a higher dispatch rate for the same amount of work.

Factors other than language semantics can also influence dispatch rates. For example, `regexdna` has an extremely low dispatch rate because of its use of native external modules for regular expression processing.

4.5 Efficacy of Dispatch Loop Optimization

We next analyze the efficacy of threaded interpretation for Python and study the primary factors that influence it. These factors include characteristics of the Python language and interpreter, the benchmarks studied, and the architecture of the underlying machine. As in previous sections, we use Java for comparison, where necessary.

4.5.1 Impact of the Language Runtime and Benchmark Characteristics

In Section 4.4, we presented analysis of the CPython interpreter, and the runtime breakdown of time spent in different components of the system. We also contrasted the data with that collected on a comparable Java VM. As discussed, the amount of time spent in the dispatch loop, the nature of Python bytecodes, and the opcode mix (at the Python bytecode level) are the primary factors affecting the efficacy of interpreter dispatch optimizations.

	Python		Java	
	Shootout	Unladen-Swallow	Shootout	DaCapo
Dispatch Rate	1.21	1.14	3.44	1.53
DTI/ITI Speedup (%)	6.61	1.99	18.10	9.72

Table 4.3: Dispatch rate (dispatches / 100 cycles) and speedup comparison for Python and Java on Intel Core2 machine

Table 4.3 corroborates these findings with a summary of dispatch rate and performance gain from threaded interpretation. We compute dispatch rate as the number of dispatches per hundred cycles. The data exhibits a correlation between dispatch rate and performance gain due to DTI and ITI. Moreover, the performance gains due to threaded interpretation are greater for Java than for Python for the VMs and benchmarks we study. For the Python suites, the dispatch rate and the performance improvement due to threading for shootout is higher than that for unladen-swallow. This is because for unladen-swallow more work is being performed by the Python VM outside of the interpreter loop which causes optimization such as threading, which optimizes the interpreter loop, to have a lesser impact on the overall performance.

Figure 4.4 shows the effect of DTI/ITI on hardware performance metrics gathered on the Intel Core2 machine for each benchmark. The figure shows number of cycles, instructions executed, branches count, branches miss rate, indirect branches miss rate and L1 ICache miss rate. We also include data for the Java benchmarks for comparison. The observations we make are:

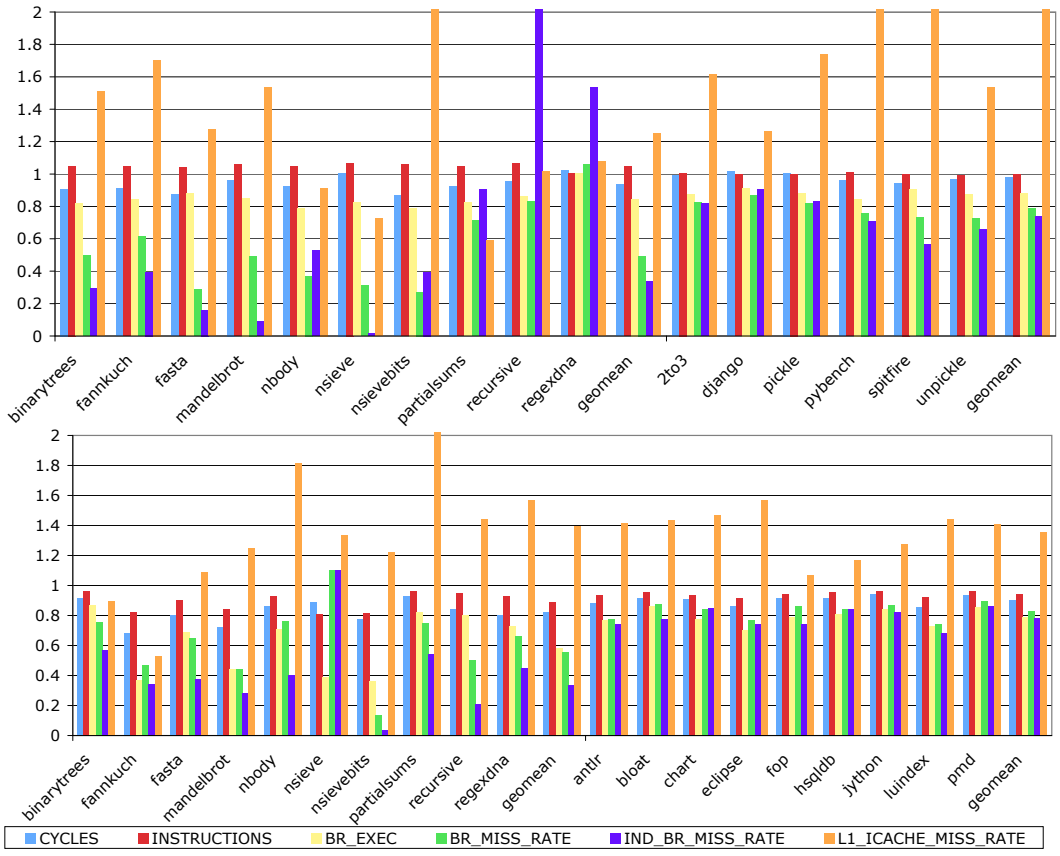


Figure 4.4: Effect of DTI/ITI on CPython (top) and Java Zero (bottom) benchmarks. This figure shows per benchmark metrics gathered on the Intel Core2 machine normalized to the baseline.

- We see a nearly consistent reduction in indirect branches misprediction rate for both Python and Java, which is expected from DTI and ITI. This reduction pays off more for Java than Python in terms of overall performance since Java has a higher dispatch rate and spends more time in the dispatch loop.

- There is a general trade-off between the L1 ICache miss rate and improvement in indirect branch prediction rate. This is because in threaded interpretation the dispatch code is replicated across every bytecode handler, which increases the memory footprint of the bytecode handlers. For some cases for shootout, the ICache miss rate goes down. We attribute this to the tight-looping nature of the shootout code which allows changes in the interpreter code layout to cause accidental improvement in ICache miss rate.

- There is a decrease in the number of branches executed for both Python and Java. This is because threaded interpretation eliminates the need of the interpreter main loop and its backward branch. This decrease in branches count is reflected in the total instruction count for Java but not for Python. The reason is that for DTI extra work is needed to transform the bytecode stream on-the-fly (expand the opcode fields) to hold pointers to the handlers.

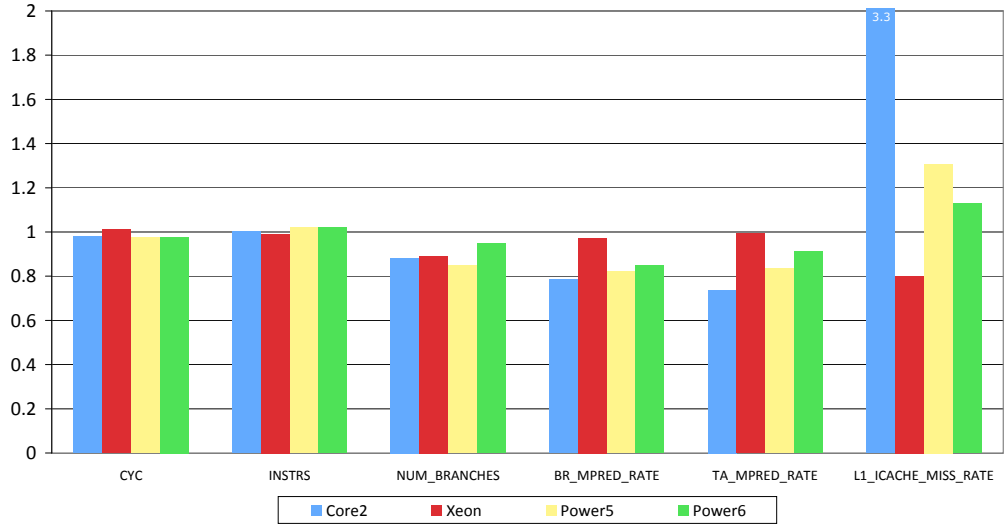
4.5.2 Analysis Across Architectures

In addition to program, language, and runtime characteristics, the underlying architecture also influences the efficacy of threaded interpretation. We next present hardware performance counter data on four different modern architectures to study the impact of architectural design decisions on threaded interpretation. We use two Intel machines: Core2 and Xeon, and two IBM machines: Power5 and Power6. Information about the architectures is in Table 4.4. Each of the processors used, performs target address prediction for indirect branches in addition to direction prediction. However, the parameters and algorithms used vary and are proprietary.

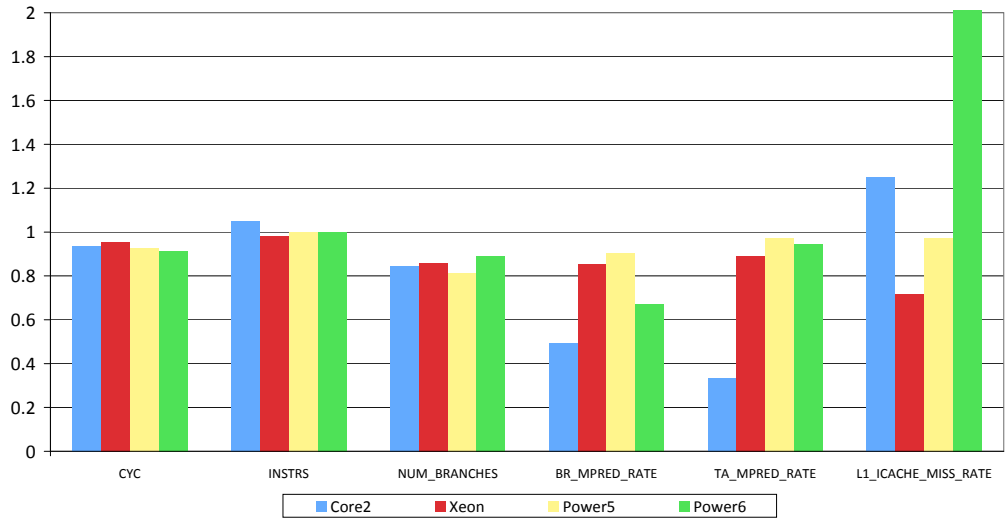
Machine	Description
Intel Core2	Core 2 Duo E6700, 64-bit, 2.66GHz, 2 x 32 KB L1 ICache
Intel Xeon	Pentium 4, 32-bit, 2.4 GHz, instruction trace cache: 12K uOps
IBM Power5	1.66 GHz, out-of-order core, 64KB L1 ICache
IBM Power6	4.5 GHz, in-order core, 64KB L1 ICache

Table 4.4: Description of the different architectures

For each architecture, we measure the number of branch instructions executed, the branch misprediction rate (total and target address), and ICache miss rate, in addition to cycles and instructions when DTI is enabled. We normalize all metrics to the default switch-case interpreter. Figure 4.5 shows the normalized values for the Python benchmarks. We present two separate charts for each of the Python benchmark suites. Each



(a) Unladen-swallow



(b) Shootout

Figure 4.5: Effect of DTI on Python benchmarks on four different architectures. The Y axis shows the geometric mean across all benchmarks in the suite, normalized to baseline.

cluster of bars represents the normalized value of the geometric mean across the entire benchmark suite for a particular architecture.

From the normalized mean cycles in Figure 4.5, we observe that the speedup due to DTI is similar across architectures (˜2% for unladen-swallow and ˜7% for shootout), despite the major differences in the hardware implementations. All four architectures also show a decrease in the total number of branches executed, and a slight increase in the number of instructions executed for the reasons mentioned previously in Section 4.5.1.

Although the overall effect of DTI, in terms of speedup, is similar across architectures, the branch misprediction and ICache miss characteristics vary. Except on Xeon, DTI increases the L1 ICache miss rate on all other systems. This is due to an increase in code size due to dispatch code replication in each handler. However, for the `shootout` suite, ICache miss stalls account for only a small fraction of stall cycles (between 0.05 and 4%), thus a small increase of the miss rate does not translate in an execution penalty.

Impact of Branch Prediction

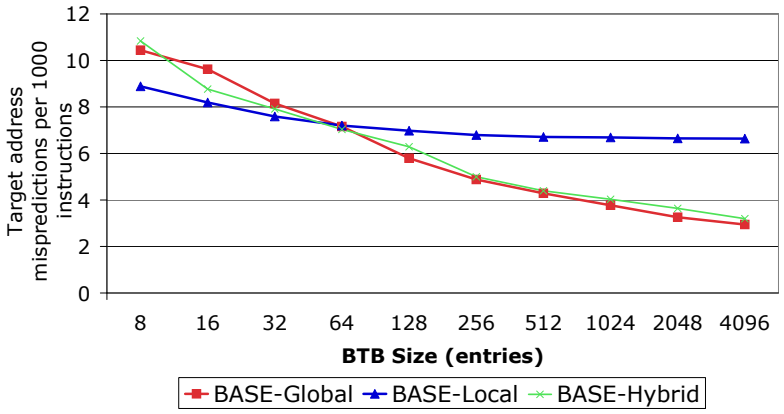
We next examine the effect of DTI on branch misprediction. In this study, we consider the impact of branch prediction mechanisms on the efficacy of DTI in more detail. From Figure 4.4, we see a nearly consistent decrease in the indirect branches

misprediction rate, as expected from DTI or ITI. However, there are some cases in Python where this trend is violated. We speculated that this is due to conflicts in the branch target buffer (BTB) caused by excessive replications of the dispatch branch. This effect is architecture-dependent due to the differences in BTB size, layout, and implementation.

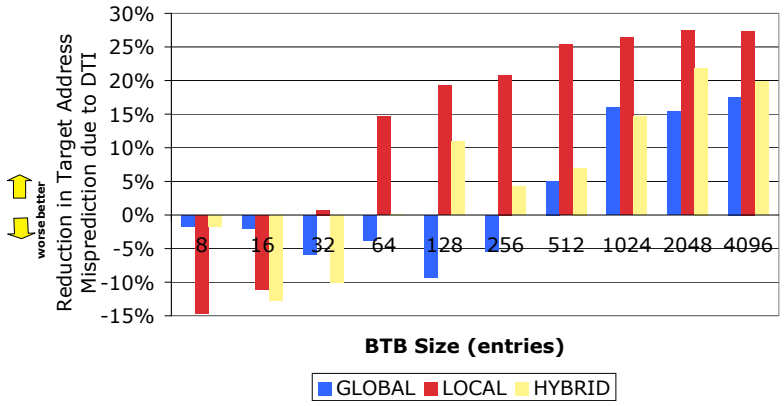
To validate this conjecture, we simulate the branch behavior to collect details on misprediction due to the dispatch branch, and the impact of BTB configurations and DTI on overall BTB misprediction rate. We chose as one benchmark, the Python `2to3` program from the unladen-swallow suite. The simulator takes as input traces of PowerPC binaries and functionally simulates three BTB configurations: *LOCAL* which is indexed solely by branch PCs, that is, each static branch has at most one entry in the BTB; *GLOBAL* which is indexed by branch PCs and a global direction history of k previous branches, and *HYBRID* which is equally split into *LOCAL* and *GLOBAL*.

The dispatch-branch is the hottest mispredicted branch in the binary. Overall, it accounts for more than 50% of dynamic indirect branches and more than 50% of BTB misprediction. Figure 4.6 (a) plots the BTB misprediction rate, for the switch-case dispatch case with varying BTB sizes. Not surprisingly, increasing BTB size always has a positive effect on reducing overall BTB misprediction rate. For *LOCAL*, however, the benefit rapidly diminishes and eventually flattens out beyond 128 entries, where the

k is computed as LOG2 of the BTB size.



(a) Default switch-case dispatch



(b) Improvement due to DTI

Figure 4.6: Effect of BTB size and history on target address misprediction.

data for the dispatch branch shows that increasing the size for *LOCAL* no longer impacts the misprediction rate for the dispatch branch.

Figure 4.6 (b) plots the effect of DTI on overall BTB misprediction rate. It shows that BTB misprediction is sensitive to both the size and the type of the BTB. The replication of branches by DTI has a negative effect on BTB misprediction rate for smaller sizes for all three BTBs. Overall, *LOCAL* benefits more from DTI than the other two BTBs do. For *GLOBAL*, DTI starts to have a positive impact on BTB only beyond a size of 256. The BTB on Core2 has 4096 entries, where the simulation result roughly matches the DTI BTB misprediction improvement observed on the real machine.

In summary, the trend of using *GLOBAL* rather than *LOCAL* BTBs in modern processors reduces the positive impact of DTI on BTB misprediction. On the other hand, the trend of larger BTBs helps to secure such impact.

4.6 Related Work

In Section 4.2.2, we identified and discussed related work on interpreter dispatch. In this section, we identify research contributions that characterize interpreter performance and that propose techniques for its improvement.

Romer et al. [93] study the performance of several interpreters (MIPSI, Java, Perl, and TCL) and conclude that the best way to optimize their execution is to spend more

time in native libraries. For the benchmarks we study however, we observe that most time is still spent in the interpreter.

JIT compilers have become increasingly popular and offer an important, albeit non-portable, solution for the optimization of interpreted languages. Aycock [13] provides a survey of JIT compilation and Davis and Waldron [31] survey the JIT techniques used for Java interpreters. Piumarta et al. [83] propose selective inlining of Java bytecodes. In their approach, they reduce interpretation dispatch overhead by dynimcally combining frequently executed traces of bytecodes into a single super bytecode. Zaleski et al. [123] expand that idea to a trace-based dynamic compiler for Java. Their implementation achieves a $2\times$ performance gain over direct threaded interpreter implementation. Gal et al. [50] use trace compilation techniques to speedup the execution of JavaScript programs and demonstrate performance improvements of up to 10x compared to their baseline interpreter.

Branch prediction in microprocessors has been an active area of research. Such techniques are likely to benefit interpreter performance given the significant branching required by its implementation. A limit study on the prediction of indirect branches is presented in [37]. Other interesting techniques with significant potential include hybrid branch prediction [40] and hardware devirtualization [63].

4.7 Conclusion

In this chapter we evaluate the performance of the Python interpreter as a representative of emerging dynamic scripting languages using several architectures. We observe that the dynamic nature of the language strongly affects the design of the bytecodes for Python.

We quantify the impact of this difference on performance using a number of different metrics and programs. We find that although Python and Java interpreters spend nearly the same proportion of time in the runtime, Python spends approximately half as much time in the interpreter loop. The Python VM spends this time instead in the language runtime lowering the bytecodes (i.e. making them less generic). The dispatch rate of Python also supports these observations: Python is able to dispatch many fewer bytecodes per cycle than Java interpreter. These results indicate that optimization effort should be directed toward the language runtime as opposed to the interpreter dispatch loop.

Much recent work on interpreter optimization has however, been focused on improving the efficiency of the dispatch loop. The primary technique to enable this include variations of "threading" – linking together bytecode handlers. These techniques have been shown to be very successful for Java and other statically-typed languages. In the second half of our study, we investigate the efficacy of threading and its behavior

on modern architectures (branch predictors, caches, etc.). We show that as our Python characterization predicts, threading is significantly less effective for Python interpretation than for Java.

This work lays the groundwork for understanding empirically the differences in execution of dynamic scripting languages as opposed to statically-typed languages. It illustrates the need for new optimizations and how they should be different from those that we have employed in the past for Java and statically typed languages. Many efforts have made important gains in this direction, e.g. Psyco [91], trace optimization [50], and type inference and specialization.

Chapter 5

Potential of Interpreter-based Optimizations for Python

The increasing popularity of scripting languages as general purpose programming environments calls for more efficient execution. Most of these languages, such as Python, Ruby, Perl and PHP are interpreted. Interpretation is a natural implementation given the dynamic nature of these languages, and interpreter portability has facilitated wide-spread use. In this chapter, we analyze the performance of CPython, a commonly used Python interpreter, to identify major sources of overhead. Based upon our findings, we investigate the efficiency of a number of interpreter-based optimizations and explore the design options and trade-offs involved.

5.1 Introduction

Dynamic scripting languages (DSLs), such as Python, Ruby, Perl and PHP, lend themselves well to interpreter-based implementations. Besides being suitable to imple-

ment the dynamic features of these languages, interpreters also bring several advantages over compilers. They provide a much simpler implementation that is easy to modify, allowing flexibility in the language definition itself. They are portable as they do not have to compile code to a specific operating system and instruction set. Moreover, without the need to compile the code, they facilitate increased developer productivity. Despite their merits, interpreters still fall behind compilers in speed of execution. Although this is the case for interpreters for both static and dynamic languages, the distribution of execution overhead is different. For static languages (e.g. Java), compile-time information facilitates straightforward translation of the source code to low-level bytecodes. Low-level bytecodes are faster to interpret and execute because they contain more information, e.g. operand types, about the operations they implement. Low-level bytecodes shift the overhead of interpretation to the dispatch process since each operand handler is itself efficiently implemented. As a result, much prior work focuses on improving interpreter efficiency by reducing dispatch overhead [93, 39, 123]. The case, however, is different for DSLs. The lack of type information and the inherent dynamic features they provide force higher-level (more abstract, type-generic) bytecode representations (many details about the operation are not known statically and therefore must be interrogated as part of the bytecode translation process) [28, 94, 81, 82]. DSL runtimes thus must perform more work within each handler making the overhead of dispatch less dominant than in their static language counterparts.

In Chapter 4, we demonstrated the difference between static and dynamic interpreter implementations and showed that optimizations that target the dispatch loop benefit static languages (Java) to a greater degree than dynamic languages. Since DSL bytecodes are untyped, and require numerous type-checks, dictionary lookups and indirect jumps, the bytecode dispatch overhead is shadowed by other sources of overhead. In this chapter, we identify three primary sources of overhead in a popular Python interpreter. The first source of overhead is type resolution and field access, which involves resolving variable types and performing necessary name lookups to resolve object fields. Secondly, there are excessive loads/stores to/from the operand stack. Finally, there is significant method calling overhead which involves argument packing and unpacking. Based on our findings, we propose and evaluate a set of interpreter-based optimizations that target these sources of overhead in an attempt to improve the performance of Python programs while maintaining the portability of the implementation (interpreter-based). Using caching of attributes to avoid name lookup, eliminating loads/stores to/from the operand stack, and inlining of frequent runtime calls, we are able to extract performance gains of up to 28% and 15% on average.

There are several implementations of the Python language [59, 62, 110]. In this work, we focus on the most widely used standard implementation: CPython [28]. CPython is a simple implementation of the language, written in C, that is portable and easy to understand and extend. CPython employs switch-based interpretation and

a combination of simple reference counting and cycle-detecting garbage collection. CPython compiles Python source to high-level type-generic bytecodes that run on a stack-based virtual machine. Being type-generic, the bytecodes defer associating abstract operations with specific implementations until runtime. This is carried out via a sequence of type-checks and indirect-branches which makes bytecode handling slower than for statically-typed languages. CPython performs name-based late binding on variables and methods (attributes). Every Python object has a dictionary (a hash table), that maps attributes names to values. When loading an attribute, a chain of hash table lookups are performed on the receiver object, its type and super-types. Attributes as well as global variables are stored in dictionaries.

One of the dynamic features of CPython are *descriptors* which are setter/getter objects that are used to associate arbitrary code with attributes accesses. For example, a descriptor can contain code to validate the correctness of a value before setting an attribute to it. Also, descriptors are used to dynamically bind methods to receiver objects. When resolving a method in an object dictionary, its descriptor is found instead which, when invoked, creates the method object on-the-fly and binds it to the receiver.

5.2 Methodology

To understand the behavior of Python programs, their sources of overhead and the impact of interpreter optimizations on CPython, we evaluate the performance of benchmarks out of the Unladen Swallow project [113] which are listed and described in Table 5.1.

We modified the CPython-2.6 [28] source to collect a variety of profiles and to experiment with different optimizations. We ran our experiments on and Intel Core 2 dual-core 64-bit machine clocked at 2.66 GHz with 2x32 KBytes of L1 instruction cache and 4 MBytes of shared L2 cache running Linux 2.6.24 patched with Perfmon2 [80], the hardware performance monitoring interface for Linux.

Benchmark	Description
2to3	A Python 2 to Python 3 translator translating itself
django	Django Python web framework building a 150x150-cell HTML table
html5lib	Parse the HTML 5 specification using html5lib
pickle	Use the pure-Python pickle module to pickle a variety of datasets
pybench	Run the standard Python PyBench benchmark suite
richards	The classic Richards benchmark
rietveld	Macrobenchmark for Django using the Rietveld [90] code review app
spambayes	Run a canned mailbox through a SpamBayes [108] ham/spam classifier
unpickle	Uses the cPickle module to un-serialize a variety of datasets

Table 5.1: The Unladen-Swallow Benchmarks

5.3 Performance Analysis

Using this methodology, we first characterize the performance of CPython. To understand, at a high level, where time is being spent in Python programs, we classify the bytecodes into several classes based on their actions. For each class, we measure the number of bytecodes executed and the percentage of time spent there. The classes we use are as follows:

- **Type Resolution and Field Access** includes `LOAD/STORE_ATTR` and `LOAD/STORE_GLOBAL`. When given an attribute name and a receiver object, `LOAD/STORE_ATTR` perform the necessary work to resolve the attribute name. This operation is expensive and involves a number of indirections and dictionary lookups, specially if the attribute lies up in the inheritance chain.
  `LOAD/STORE_GLOBAL` perform the same task for global variables. They look for the attribute name in the global dictionary, if not found they look for it in the built-ins dictionary. Since at most two lookups are necessary, they are faster than `LOAD/STORE_ATTR`. In the standard CPython implementation, these lookups are performed every time one of these bytecodes are executed with no caching of previous results.

- **Locals Loads/Stores** are bytecodes that transfer values between the locals/constants and the operand stack. There are three bytecodes in this class. Namely, `LOAD_FAST`,

STORE_FAST and LOAD_CONST. These bytecodes are cheap, yet, as we will show later, they are encountered quite often during execution.

- **Method Call and Return** includes bytecodes that perform method calls and returns. The most common opcode in this class is CALL_FUNCTION, which pops a function object and its arguments from the operand stack and invokes it. This bytecode is used for calling Python functions as well as C (built-in or extension) functions depending on the function object it operates on. When calling a Python function, the call overhead involves packing the arguments into a tuple, setting up a new call-stack frame and initializing it, and unpacking the argument tuple in the callee. Calls to C functions are cheaper than to Python functions.

- **Compare and Conditional Jump** includes COMPARE_OP and opcodes with the prefix of JUMP_IF. These bytecodes produce or consume generic objects. Comparing two objects and checking whether an object is true or false are type-specific operations making these bytecodes amenable for type specialization and unboxing.

- **Generic Numeric** includes generic bytecodes for Python numeric and string objects, such as the ones with the prefix BINARY, UNARY, or INPLACE. These bytecodes are expensive as they are type-generic, thus involve type-checks and

indirect-jumps. They are also good candidates for optimizations such as type specialization and unboxing.

- **Stack Manipulation** are bytecodes that manipulate the top elements on the operand stack. For example, `DUP_TOP` duplicates the top element on the stack.

- **Control Flow** includes bytecodes to manipulate loops such as `GET_ITER`, which gets the iterator of an object, and `FOR_ITER`, which invokes the iterator. These bytecodes are also good candidates for type specialization as loop iterators are typically ranges of integers.

- **Built-in Operators** includes bytecodes for built-in container types such as list, map, and tuple, e.g., `LIST_APPEND`. These bytecodes save the necessary calls to the runtime to achieve the same functionality.

- **Others** includes all others.

Figure 5.1 and Figure 5.2 show the count and time distribution for each class of bytecodes, respectively. The percentages are from the total cycles/counts for all bytecodes. We make the following observations from the bytecode profile:

1. **Type Resolution and Access** *consumes a significant fraction of time.* For most of the benchmarks we analyzed, this classes of bytecodes takes more than 20% of the time. In the CPython implementation, subsequent executions of the same

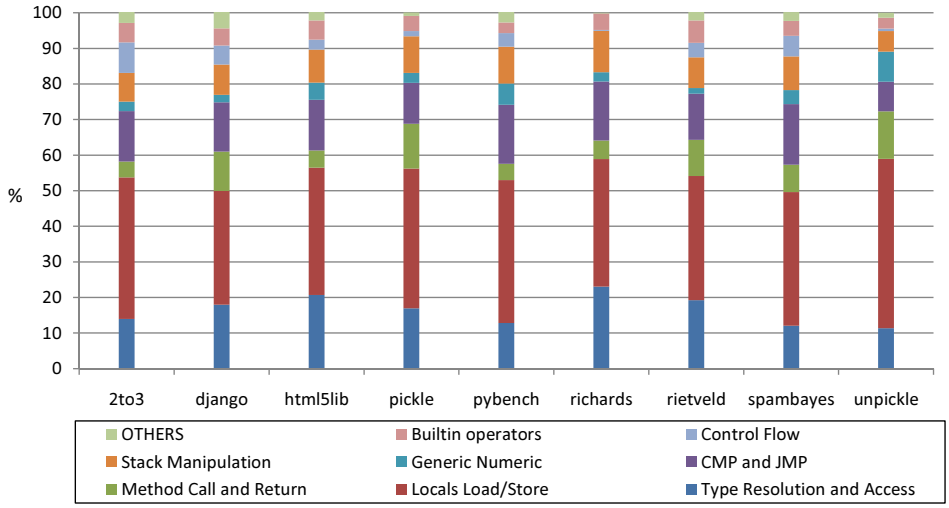


Figure 5.1: Bytecode Histograms based on bytecode class (counts)

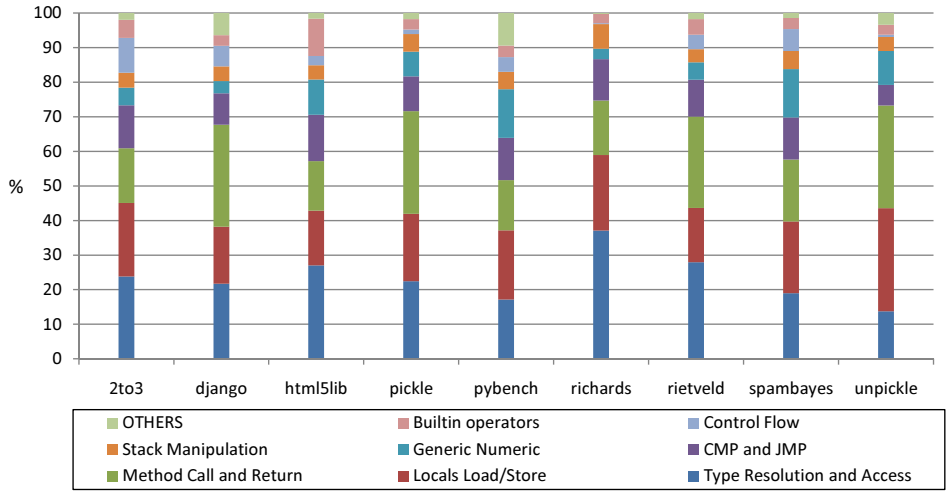


Figure 5.2: Bytecode Histograms based on bytecode class (cycles)

`LOAD_ATTR`, say in a loop, causes the type resolution to be redone even in cases when invariance of the outcome is guaranteed. CPython does no effort to optimize this except for caching of bound methods by the type they are bounded to and the method name. However, since methods and fields resolution share the same execution path in CPython, many hash lookups are needed before CPython eventually detects that the field requested is a method and not a field. This causes the method cache to be referenced late in the bytecode handler after most of the hash lookups are already done. In fact, we have not observed any speedup due to the method cache and in some cases CPython runs slightly faster without it.

2. *The most frequently executed bytecodes are* **Locals Loads/Stores**. Although they are cheap operations, being executed repeatedly results in considerable amount of time spent on moving values from/to the operand stack.

3. *Significant time spent in* **Method Call and Return**. Note that the bar for cycle distribution of this class in Figure 5.2 includes only the 'overhead' of calling a method. It does not include the time spent in the target method, nor time for resolving the function via its name. The disproportionality of the large distribution in the 'Cycles' compared to its distribution in 'Counts' in Figure 5.1 indicates high overhead in a single method call. This indicates potential for optimizations that either eliminate or reduce method call overheads, such as method inlining.

4. **Type Resolution and Access, Locals Loads/Stores and Method Call and Return** *account for more than 50% of the time.* Each of these classes of bytecodes consume a reasonable amount of execution time, but when combined they dominate the execution frequency and time for all benchmarks. This behavior calls for a simple optimization to target each type of the three bytecode classes.

5.4 Optimizations

Our analysis of Python programs over CPython motivates us to investigate the overheads caused by the three primary sources of overhead: type resolution and access, local variable loads/stores to/from the operand stack, and method call/return. Our goal is to identify, design, and implement simple interpreter optimizations that target each source in an effort to improve Python performance and to better understand the challenges we face in doing so.

We improve type resolution and access using a caching scheme to reduce access time and dictionary lookups. For local variable loads and stores, we investigate a mechanism that eliminates loads and stores from the bytecode stream. For method call/return, we investigate inlining opportunities. We examine the efficacy of each optimization and how well-suited it is for the overhead it targets. Moreover, we explore the design parameters and trade-offs involved.

5.4.1 Attributes Caching

Loading global variables and object attributes in Python is a common, yet expensive, operation. One reason behind this is that, due to the dynamic nature of the language, global variables and object attributes are stored in hash tables (dictionaries) and are referenced by their names. A single read can involve several dictionary lookups by traversing up an object's type inheritance chain. Another source of overhead is indirect function calls that the runtime performs to handle an access in a type-generic manner. Two bytecodes are most frequently used for globals and attributes loading: `LOAD_GLOBAL` and `LOAD_ATTR` . Each bytecode instance performs a lookup for a constant variable name that it is tied to. The variable name for every instance is constant throughout execution.

`LOAD_GLOBAL` , given a constant global variable name, resolves it as follows:

1. Look up variable name in the globals dictionary. If found, return it; else
2. look up in the built-ins dictionary. If found, return it; else
3. raise name-not-found exception.

`LOAD_ATTR` is given a receiver object and a constant attribute name. The lookup process is fairly complex. Since Python supports multiple-inheritance, each type object has a method resolution order (MRO) structure to optimize the lookup operation. The MRO is a list of references to a type object's super types placed in the order of traversal

up the inheritance chain during an attribute lookup. This saves a traversal of the inheritance chain (and pointer chasing) which is replaced instead by a single scan over the the object's type MRO. This is particularly important for Python as it allows multiple inheritance and potentially complex inheritance structures. The lookup process also depends on descriptors that we reviewed in Section 5.1. An attribute read proceeds as follows:

1. Look up the attribute name in the dictionaries of each type object in the MRO of the receiver object's type. If the entry is found and is a data descriptor, invoke its getter to read the attribute; else,

2. look it up in the instance dictionary of the receiver object. If found, return attribute; else,

3. if the entry found in the MRO is a method descriptor, invoke its getter to read attribute; else,

4. return the entry found in the MRO as the attribute; else if no entry was found,

5. raise attribute-not-found exception.

To avoid the above chain of lookups, we propose two caching schemes: caching of global variables and objects attributes.

LOAD_GLOBAL Cache

Design

The operand of a `LOAD_GLOBAL` is constant and is the index of a global variable name in the pool of names used by the code. On code loading, we replace the operand of every `LOAD_GLOBAL` to point to a structure that holds the original operand in addition to a single-entry caching structure (Figure 5.3). Thus, each `LOAD_GLOBAL` has its own cache. The cache holds the following values: a pointer to a dictionary (`dictObject`), a pointer to a dictionary entry (`dictEntry`), a version number and an execution frequency counter.

Initially, each `LOAD_GLOBAL` is executed normally, following the slow path of global variable lookup, and the execution frequency counter is incremented. Once a `LOAD_GLOBAL` gets hot (execution frequency goes above a threshold, which we set to 100 based on performance tuning), its cache is initialized. The cache holds a pointer to the dictionary where the variable from the last access was found and another pointer to the dictionary entry that holds it. Subsequent executions of a cached `LOAD_GLOBAL` will use the dictionary entry pointer to fetch the value directly. Since a pointer to the dictionary entry containing the variable, and not the variable itself, is cached, we guarantee that the value fetched is always up-to-date.

LOAD_GLOBAL Cache Invalidation

Although caching a dictionary entry pointer makes the cache valid even if the global variable value has changed, there are still reasons for which cache invalidation is needed:

1. **Dictionary shape change**: The shape of a dictionary changes if an element is deleted or the hash table representing the dictionary is resized. Thus, the cached dictionary entry pointer becomes invalid.

2. **Shadowing**: A variable name in one dictionary shadows another if it has the same name as an existing variable in another dictionary where the first dictionary precedes the second in the lookup chain. For example, if the dictionary entry of the variable ”foo” is already cached from the built-ins dictionary, inserting a new variable ”foo” in the globals dictionary makes the cache invalid since the globals dictionary is looked up first.

We detect these cases by attributing a version number to every dictionary. Version numbers are incremented whenever a dictionary shape changes. When caching a variable from a dictionary, the version number is copied to the cache. If the cached version number and the dictionary version number (fetched using the cached dictionary pointer) do not match, a cache miss is declared and we bail out to the slow execution path and update the cache. Whenever a new variable is added to the globals dictionary

(STORE_GLOBAL), the version number of the built-ins dictionary is incremented; thus, we conservatively invalidate all cached variables from the built-ins dictionary since there is a possibility they have been shadowed by the new insertion. Although, seemingly over-conservative, this simple solution is sufficient since adding new variables happens mostly at the beginning of execution and is quite rare afterwards.

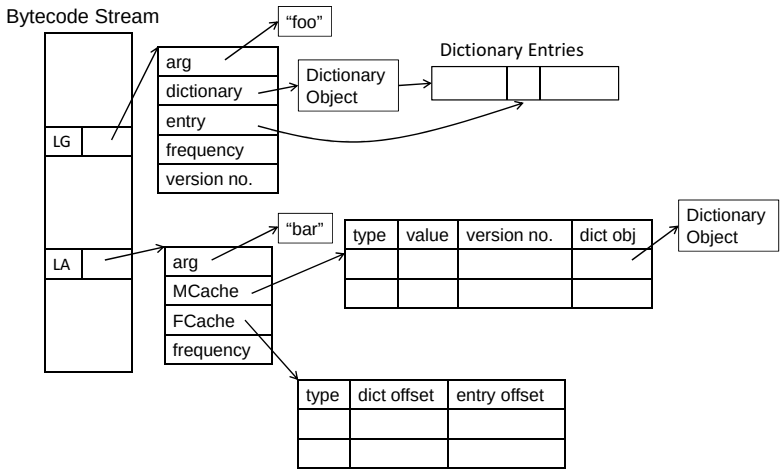


Figure 5.3: Cache layout for LOAD_GLOBAL and LOAD_ATTR

LOAD_ATTR Cache

Every LOAD_ATTR has potentially one of two types of cache structures, one for methods (MCache) and one for instance fields (FCache). Both caches are referenced by the type of the receiver object. The allocation of these caches, however, is deferred until it is known that the LOAD_ATTR is amenable to caching. Every LOAD_ATTR can

be in any of the following five states. We express each using a different, specialized opcode:

- `LOAD_ATTR` : This is the initial state for all `LOAD_ATTR`s . In this state, we check if the `LOAD_ATTR` is "cacheable" or not. A cacheable `LOAD_ATTR` is one that follows the default logic of attribute access . Also, a cacheable `LOAD_ATTR` has to read the attribute directly from a dictionary, not through a data descriptor. This is because the descriptor logic is arbitrary and can even generate values on-the-fly instead of reading from a dictionary. Finally, we do not cache fields loaded from type objects (i.e. static fields) simply because maintaining a valid cache for this case is fairly complex especially that static fields are rarely used. If the `LOAD_ATTR` is cacheable, its opcode is rewritten to be `LOAD_ATTR_CACHEABLE`, else `LOAD_ATTR_NORMAL`.

- `LOAD_ATTR_NORMAL`: A non-cached `LOAD_ATTR` following the slow path of attribute access.

- `LOAD_ATTR_CACHEABLE`: A cacheable `LOAD_ATTR` . At this state, if the `LOAD_ATTR` is executed more than two times, it is transformed into either `LOAD_ATTR_CACHED_M` or `LOAD_ATTR_CACHED_F` based on whether it loaded a method or an instance field.

This is the case when tp_getattro field in a type object points to PyObject_GenericGetAttr(). Using CPython C-API, users can define their own functions to get attributes, in which case tp_getattro will point to a user-defined function making `LOAD_ATTR` uncacheable since the access logic becomes arbitrary.

- `LOAD_ATTR_CACHED_M`: A cached `LOAD_ATTR` that loaded a method from a type object. Hence, it loads it from its MCache instead.

- `LOAD_ATTR_CACHED_F`: A cached `LOAD_ATTR` that loaded an instance field. Hence, it loads it from its FCache instead.

We define an instance object as any object that is not a type object (i.e. cannot be instantiated). If an instance field is loaded, be it a method or not, the FCache is used. Although, in theory, it is possible for a `LOAD_ATTR` to mix loading of methods from type objects and fields from instance objects, in all the cases we have studied, we have not encountered such a case.

When a cached `LOAD_ATTR` is executed, it reads the type of the receiver object on the top of the operand stack. The type is then used to reference either the MCache or the FCache. For the MCache, the value of the method object, or rather the method descriptor used to create it is cached . The MCache also has a version number that is used for invalidation.

For the FCache, since we reference instance fields whose values vary across objects, the offset of the field in the receiver object (dictionary offset and entry offset) is cached. Caching of instance fields is based on the assertion that objects of the same type will most likely have dictionaries of identical shape. In such cases, for all objects of the

In CPython, the type object dictionary contains a descriptor of the method, instead of the method object itself. Method objects are allocated on the fly, when referenced, where they are bound to their receiver objects.

same type, it is sufficient to cache the offset of the instance field instead of having a cache entry per object. It remains, however, to guarantee that the fetched field is the correct one. We achieve this by comparing the `LOAD_ATTR` operand (the attribute name) with the key fetched from the dictionary. Since strings are interned in CPython, this is a quick equality check of pointers.

If the type referenced is not found, we add a new cache entry and resize the cache, if necessary. All caches start with a single entry, upon the first resize, we make it five entries. For all future resizes, we double the number of entries.

LOAD_ATTR Cache Invalidation

Cache invalidation may be needed on dictionary insertion and deletions. Insertions can cause shadowing of attributes and can cause a resize of the dictionary if it gets full. Similarly, deletions can cause unshadowing of attributes and dictionary downsizing. Insertions are common when new instance objects are initialized (attributes assigned for the first time) while deletions are less common and considered more of meta-programming. Both insertions and deletions are far more common to instance objects than to type objects. Type objects are constructed using the `BUILD_CLASS` bytecode which initializes the type object and adds its methods. Typically, type objects remain fixed after initialization and are rarely manipulated during execution. In fact, we have not encountered any cases in our study where type objects are modified. Our cache

invalidation scheme deals with the two cases of modifying type objects and instance objects dictionaries differently.

1. **Type object dictionary insertion/deletion**

   Since this case is quite rare, we handle it in a simple conservative manner. Whenever an insertion or deletion occurs on a dictionary whose owner is a type object, we perform a global flush of all MCaches and FCaches. MCaches are flushed because the new modification could have caused (un)shadowing of a cached attribute or simply overwritten its value. FCaches are flushed because a new insertion to a type object may contain a data descriptor for an already cached instance attribute. A data descriptor can return arbitrary values for the attribute. The global invalidation causes all `LOAD_ATTR`s to go back to their initial state and the caches are re-populated again. Despite the apparent large cost of flushing all caches, we have never seen this happen in our benchmarks.

2. **Instance object dictionary insertion/deletion**

   (a) **MCache**

   A modification to an instance object dictionary can (un)shadow a type object attribute up the type lookup chain. We detect type objects dictionaries that are affected and increment their version numbers. MCache lookup involves comparing the cached and the dictionary version number, if they are

different, the cache is flushed. Note that finding the dictionaries affected in the type hierarchy (i.e. whose version number is to be incremented) is piggy-backed on the lookup process as part of setting the attribute value. A a scan of the MRO is already performed to check if a setter descriptor exists for that attribute (similar to Section 5.4.1).

(b) **FCache**

Since object dictionaries are leaf dictionaries in the type hierarchy, they can never be shadowed. Thus, a modification can only invalidate the cache if a dictionary resize is required. CPython performs resizing by allocating a new larger dictionary and copy entries from the old dictionary to it. This may cause the cached entry offset in the FCache to be invalid. The dictionary offset, however, is always valid, since it is part of the object header and is fixed per type. Hence, the FCache merely needs to detect that the cache layout still leads to the correct entry. This is achieved by comparing the requested attribute name (the arg field in Figure 5.3) and the key stored in the entry. The comparison is a cheap pointer comparison since CPython interns all strings.

Analysis

We carried out a set of experiments to understand the effectiveness of the caching scheme. Figure 5.4 shows the ratio of dynamic `LOAD_ATTR`s count that cannot be cached. Most of the benchmarks have a negligible percentage of executed non-cacheable `LOAD_ATTR`s (below 5%). On average, only 0.7% of the `LOAD_ATTR`s are non-cacheable. Pybench has a significantly high percentage of uncacheable `LOAD_ATTR` s (26%), this is because one of the sub-tests of pybench (Lookups.py) tests static fields access by repeatedly loading them. These results show the applicability of the caching scheme on the majority of `LOAD_ATTR`s in the code.

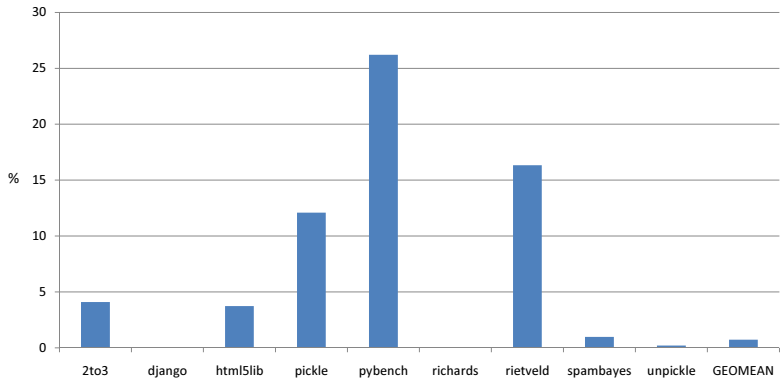


Figure 5.4: Percentage of non-cacheable `LOAD_ATTR`s

For the `LOAD_ATTR` caching, we experimented with three possible varieties:

1. **LA multi swap:** The cache is resizable, starting from a single entry. The last referenced cache entry is swapped with the top of the cache as a simple heuristic to speedup cache access.

2. **LA multi no swap:** Same as above, but without swapping. We test this configuration because if the majority of the caches are small then swapping might only incur overhead without much benefit.

3. **LA single:** Each cache has only a single entry which is overwritten on a cache miss.

We test the above configurations with and without `LOAD_GLOBAL` caching. Figure 5.5 reports on the speedup achieved for all combinations. We can see that `LOAD_GLOBAL` caching by itself (LG) is not sufficient to gain a reasonable speedup and can even lead to a slowdown. Notice, however, that for nearly all benchmarks, adding `LOAD_GLOBAL` caching to `LOAD_ATTR` caching improves overall performance (up to 16%), despite that in some cases this happens by mere constructive interference of the two optimizations (pickle and spambayes). Overall, the best configuration to use is `LOAD_GLOBAL` caching with multi-entry `LOAD_ATTR` caching with swapping which achieves 8% speedup on average and up to 17% (richards).

Figure 5.6 shows the hit ratio for all caches (`LOAD_ATTR` method and instance field cache, and `LOAD_GLOBAL` cache) using the multi-entry with no swapping configuration. The figure shows a nearly perfect cache performance. Most of the cache misses are actually due to a cold cache. The instance field cache exhibits the highest miss

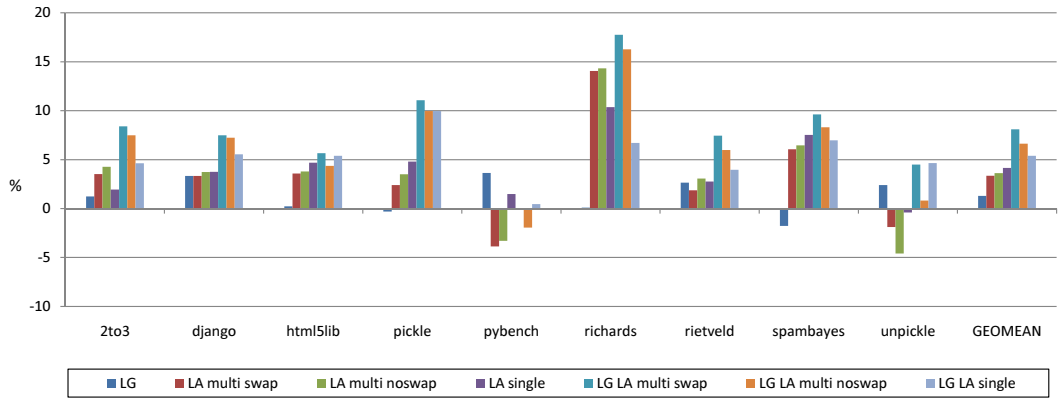


Figure 5.5: Speedup of different combinations of caching

rate. This is the effect of few `LOAD_ATTR`s operating on receiver objects with different layouts.

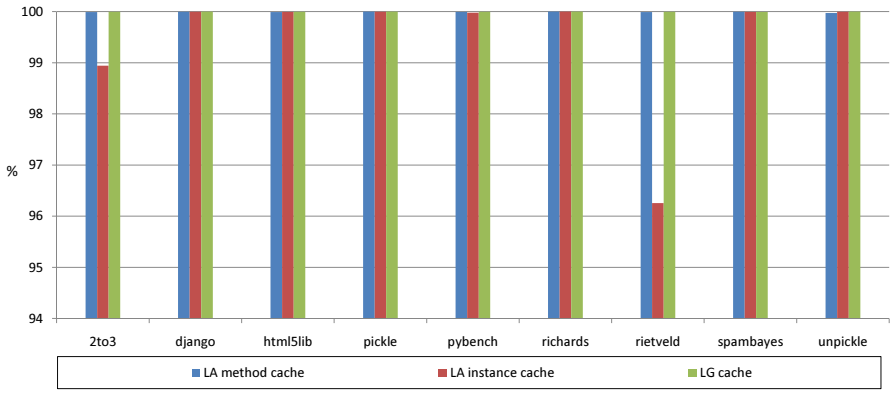


Figure 5.6: Cache hit ratio for all three types of cache: `LOAD_ATTR` method and instance field caches and `LOAD_GLOBAL` cache

It may appear at first that we achieve nearly perfect caching due to the unlimited resizing that we allow. This is not the case, however. Figure 5.7 presents a histogram of the caches sizes for the `LOAD_ATTR`s caches. For each benchmark, we count the number of caches that have 1, 5, 10, 20, 40 and 80 entries. We do that for the method

and instance fields caches. The figure shows that the vast majority of the caches have a single entry. This is most interesting for instance field caches. Since a cache entry corresponds to an object type, this indicates that most `LOAD_ATTR`s are monomorphic; that is they operated on objects of the same type. The maximum cache size reached is 80 entries; only two programs reach this maximum size (5 caches for 2to3 and 18 for pybench).

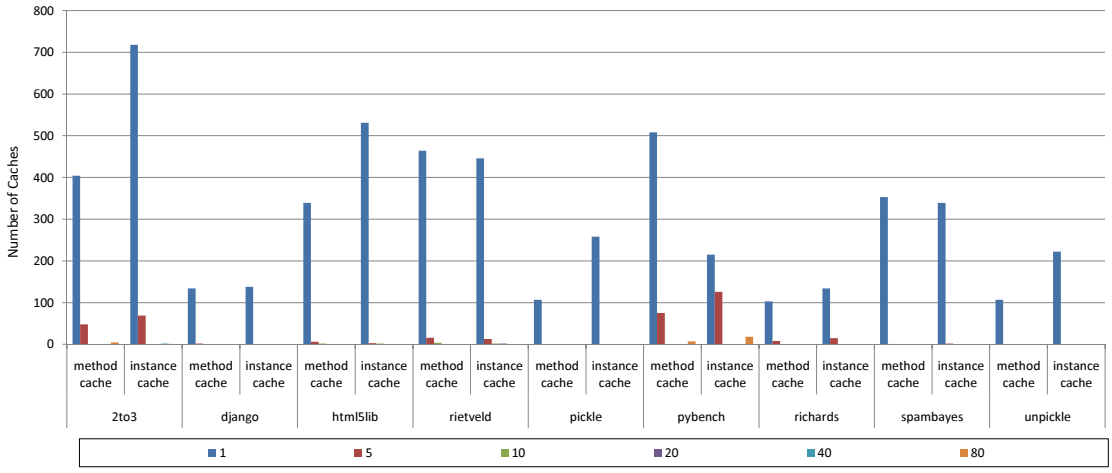


Figure 5.7: A histogram of the `LOAD_ATTR` cache sizes. It shows the state of the cache sizes at program termination

Finally, we look at the impact on memory usage. Figure 5.8 shows the extra memory needed in KiloBytes for multi-entry `LOAD_ATTR` cache and `LOAD_GLOBAL` cache. The memory includes all necessary data structures and cache entries used to maintain the cache. Most benchmarks consume less than 150KB of extra memory. Naturally, the `LOAD_ATTR` cache consumes more memory due to the dynamic nature of the bytecode

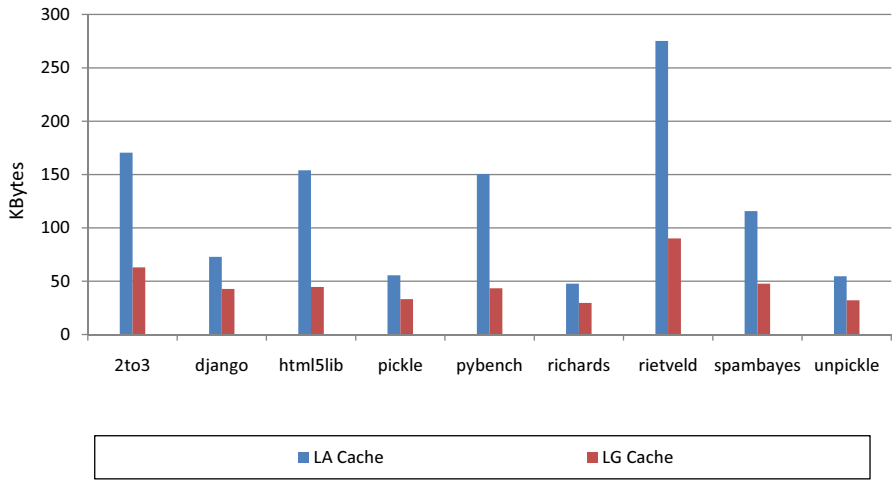


Figure 5.8: Amount of memory, in KiloBytes, needed for all caches

where more cache entries are needed. The size of one `LOAD_ATTR` FCache entry is 16 bytes, MCache entry is 28 bytes and `LOAD_GLOBAL` cache entry is 28 bytes.

Notice that in Figure 5.7, 2to3 has a higher number of `LOAD_ATTR` cache entries than rietveld, yet rietveld consumes more memory in Figure 5.8. The reason is that we pre-allocate all of the cache data structure needed for each `LOAD_ATTR` except for the actual cache entries which are allocated lazily. This means that some `LOAD_ATTR`s might have memory allocated for them although they have no cache entries (e.g. they were never executed). Hence, the allocated structures can add to the total memory size and not to the total number of cache entries.

5.4.2 Load/Store Elimination

We next consider how to eliminate loads and stores – the bytecode instructions that move values between local storage and the operand stack. In this section, we investigate a static bytecode transformation technique that converts certain bytecodes from stack-based to register-based versions. The latter access the locals from the virtual local registers directly.

We perform the operation selectively on bytecodes for which the transformation eliminates the need to copy values to/from the stack. Figure 5.9 shows a simple example of the process. In the given basic block of bytecodes, three values are loaded on the stack via a `LOAD_GLOBAL` , a `LOAD_CONST` and a `LOAD_FAST` which are then consumed by `STORE_SUBSCR`. By converting the `STORE_SUBSCR` to a register-based version , we can eliminate the `LOAD_CONST` and the `LOAD_FAST`.

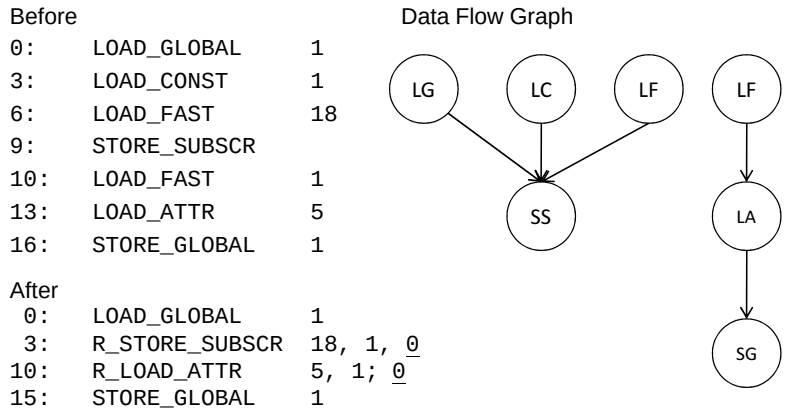


Figure 5.9: Example of Load/Store elimination with the corresponding Data Flow Graph. Underlined operands are stack operands.

We copy the source local register address from the loads as arguments to the new register-based bytecode (`R_STORE_SUBSCR`) which can now read its operands directly from the locals, saving four memory references (two for each load eliminated). Notice, that it is not possible to eliminate the `LOAD_GLOBAL` since it fetches its variable from a dictionary. In such cases, we leave the `LOAD_GLOBAL` and place zero (underlined) in the `R_STORE_SUBSCR` argument list. This indicates that the read should be from the stack. `R_STORE_SUBSCR` thus operates by reading its argument list left-to-right; for non-zero arguments, the values are read from registers, else they are popped from the operand stack.

Similarly, in Figure 5.9, we transform `LOAD_ATTR` to employ register-based local variable access. In this case, we eliminate only `LOAD_FAST`. Since `STORE_GLOBAL` inserts a value into a dictionary, the output from the `LOAD_ATTR` is pushed on the stack. Notice that the only loads/stores eliminated are those that operate on registers. Namely, `LOAD_FAST` , `LOAD_CONST` and `STORE_FAST`, which we refer to, in this section, as loads/stores.

We use a simple static analysis to perform this optimization upon bytecode loading. In particular, we build a control flow graph (CFG) from the bytecode stream and use abstract interpretation to build a data flow graph (DFG) for every basic block. Using the DFG, we select which nodes to transform to register-based using one of four selection criteria:

- **STRICT** A DFG node is transformed to register-based iff all of its immediate predecessors and successors are loads/stores. This criterion maximizes the number of loads/stores eliminated per transformation.

- **INPUT** A DFG node is transformed to register-based iff all of its immediate predecessors are loads.

- **MAJORITY** A DFG node is transformed to register-based iff the majority of its immediate predecessors and successors are loads/stores.

- **ANY** A DFG node is transformed to register-based iff at least one of its immediate predecessors and successors is a load/store.

There are key trade-offs that we make with this optimization. The stack-based version of a bytecode is more compact than the register-based version. In the former, the operands are implicit and are read from the operand stack, while in the latter, each operand's location must be explicitly included in the bytecode. Therefore, we are trading off code size for the number of eliminated loads/stores. We, thus, must transform only when this extra code size is amortized by the elimination of loads/stores.

Each of the above criteria has it advantages and disadvantages in that sense. Doing a STRICT transformation guarantees gain out of every transformed bytecode, but, since the criteria is strict, few bytecodes can be transformed. The INPUT criterion is more relaxed, it requires only the inputs to come from loads, it still transforms when

advantageous, yet, being more relaxed, it transforms more bytecodes. MAJORITY is even less relaxed but still applies a simple heuristic that ensures gain. ANY is the most relaxed of all, eliminating the majority of loads/stores in the code while increasing the code size significantly.

Another trade-off that this optimization makes is the complexity of the register-based bytecode handlers. If a register-based bytecode can mix reading from register and from the stack, then checks are needed in its handler to determine where to read from. For STRICT and INPUT, register-based bytecodes always read from registers, thus the handlers are simple. MAJORITY and ANY require checks.

Analysis

Figure 5.10 compares the dynamic count of eliminated loads/stores for all four criteria. Rietveld is missing here as we were unable to get it to run with this optimization. One can see that STRICT performs poorly and eliminates, on average, less than 20% of the loads/store executed. INPUT is much better with an average of 60%. The numbers go up for MAJORITY and ANY, which eliminate almost all loads/stores. Based on the trade-offs mentioned, we adopt INPUT as our selection criteria. It eliminates more than half of the loads and allows simple implementations of the register-based bytecodes handlers.

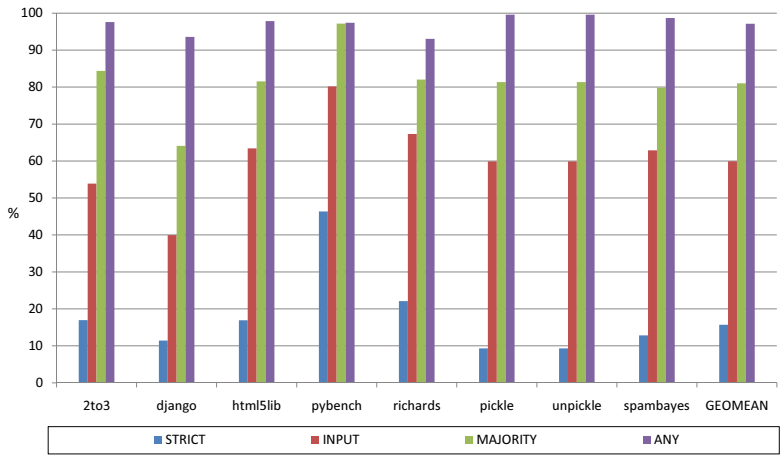


Figure 5.10: Profile of the percentage of eliminated Loads/Stores for all variations of the Elimination technique

The next question we investigate is how many register-based bytecodes to support. Figure 5.11 addresses this question using a cumulative function of the estimated speedup plotted against the number of register-based bytecodes supported. To produce this data, we measure the average execution time of a single `LOAD_FAST` and `LOAD_CONST`. We then feed our static analyzer with how many and which register-based bytecodes to support. The static analyzer finds the corresponding number of bytecode loads eliminated from which we get an estimate of the save in execution time. In the figure, the x-axis is the number of register-based bytecodes supported. We start with the most frequent bytecodes and move downwards. The y-axis is the corresponding estimated save in execution time. The figure shows that by 15 bytecodes nearly no more speedup is gained. From our experience and evaluation data, we currently support the register-based version of the 15 bytecodes listed in Table 5.2

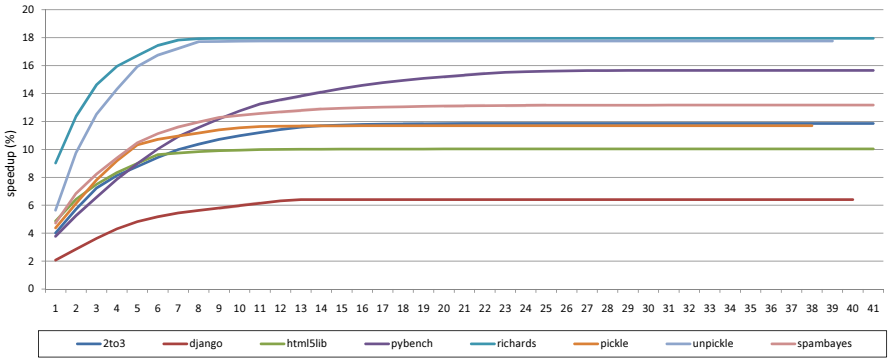


Figure 5.11: A cumulative function of the speedup estimate plotted against the number of bytecodes transformed to Register-based

LOAD_ATTR	COMPARE_OP	BINARY_SUBSCR	RETURN_VALUE
SLICE	BUILD_TUPLE	STORE_ATTR	YIELD_VALUE
STORE_FAST	BINARY_ADD	BINARY_SUBTRACT	STORE_SUBSCR
BUILD_SLICE	INPLACE_ADD	BINARY_MULTIPLY	

Table 5.2: Bytecodes for which a register-based version is supported

To support simple register-based bytecode handlers, all locals and constants must be referenced in a uniform manner. This is not the case in CPython, as constants are stored in code objects while locals are part of the virtual call-stack frames. To overcome this, we maintain a copy of the constants of a code object in all call-stack frames that correspond to it.

Figure 5.12 shows the overhead incurred by the static analysis on the bytecode and the constants copying. We measure this by carrying out all the code transformations without actually using the transformed code. On average, the overhead is less than 2% and 5 out of the 8 benchmarks shown have less than 1% overhead.

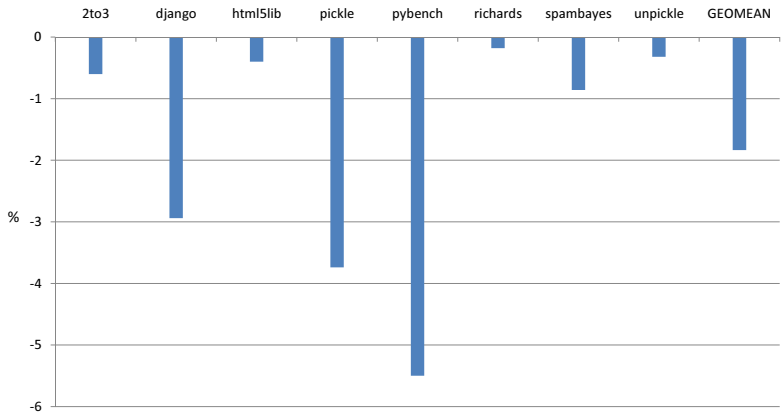


Figure 5.12: Static analysis and constant copying overhead for Load/Store elimination

Finally, Figure 5.13 reports the efficacy of the optimization and some hardware performance metrics. In terms of speedup, we achieve 5% speedup on average and as much as 9%. There is a consistent, and sometimes large, increase in L1 instruction cache miss rate. This is due to the addition of new bytecode handlers in the dispatch loop. Some increase is also seen in the L2 cache miss rate, we attribute this to the code size increase as well as to the data structures that we employ to implement the static optimizations. For all benchmarks, the amount of work (instruction count) performed is reduced.

5.4.3 Inlining

The last optimization that we investigate is inlining of method calls aiming to reduce call overhead. To know which type of method calls to target, we measure the dynamism of C and Python method calls. We measure dynamism by profiling each

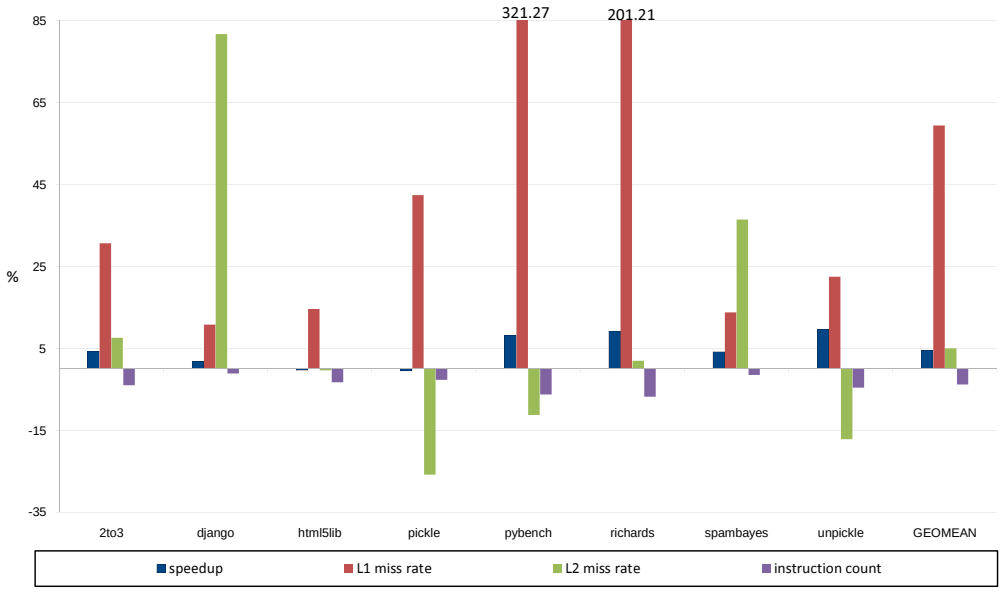


Figure 5.13: Speedup of Load/Store elimination optimization. The figure also shows effect on L1 ICache and L2 Cache as well as the decrease in the instructions executed

call bytecode and determining its morphism by counting how many call-targets it has. We then determine the dynamic count of calls made through monomorphic and polymorphic bytecodes. We also distinguish C calls (calls to the runtime or C extensions via Python calling convention) and Python calls. Figure 5.14 shows the dynamic count percentage of each of the four types of calls. Almost all of the C calls are monomorphic calls and for nearly all benchmarks, there is no polymorphic C calls made. This is a good indication that C calls are a potential inlining target.

Another evidence is shown in Figure 5.15, which shows the percentage of call-sites responsible for 90% of the calls made for C and Python methods. The figure shows that the 90/10 rule holds strongly for calls to C methods, where less than 10%

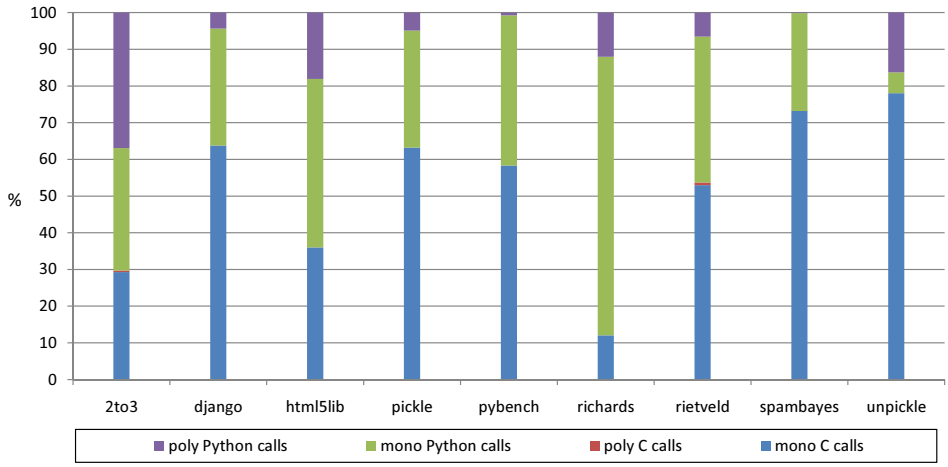


Figure 5.14: A breakup of the dynamic count of method calls by their morphism and type (C or Python)

of the C methods call-sites are sources of 90% of the calls invoked. In other words, few C methods call-sites cause the majority of the call overhead. This is not the case for Python methods where the calls are almost uniformly distributed. These results motivate us to look more closely into the call targets of the most frequent C calls. We find that `isinstance()` (a method to detect if an object is an instance of a class) is a frequently used built-in function, especially for django. We tried a simple optimization where we employ a special bytecode to implement this function – to simulate inlining it into Python bytecode. Figure 5.16 shows up to almost 8% speedup. We attempted to inline additional methods in this way, but we observed that adding more opcode handlers degrades performance. This is because the CPython dispatch loop is sensitive to change. Pickle and pybench show slowdown attributed to this effect.

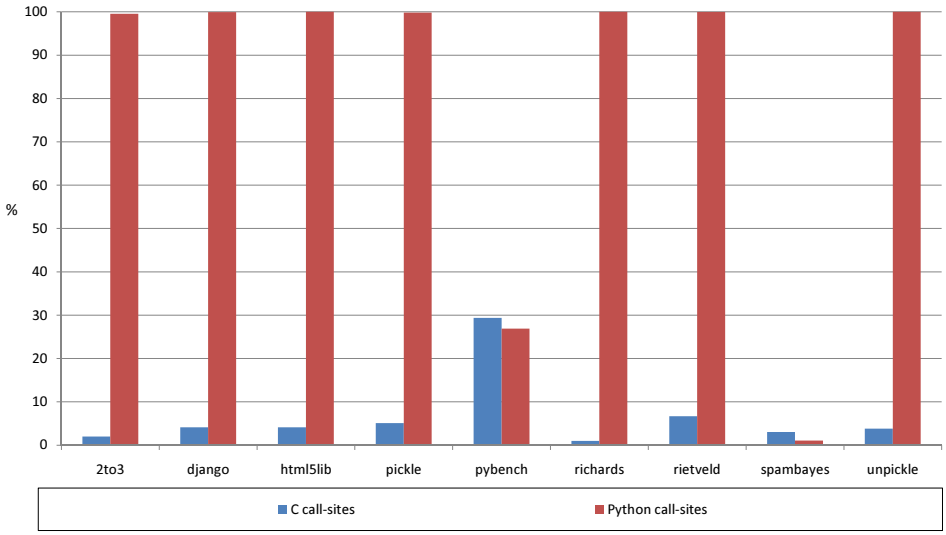


Figure 5.15: Percentage of C and Python call-sites responsible for 90% of the calls made

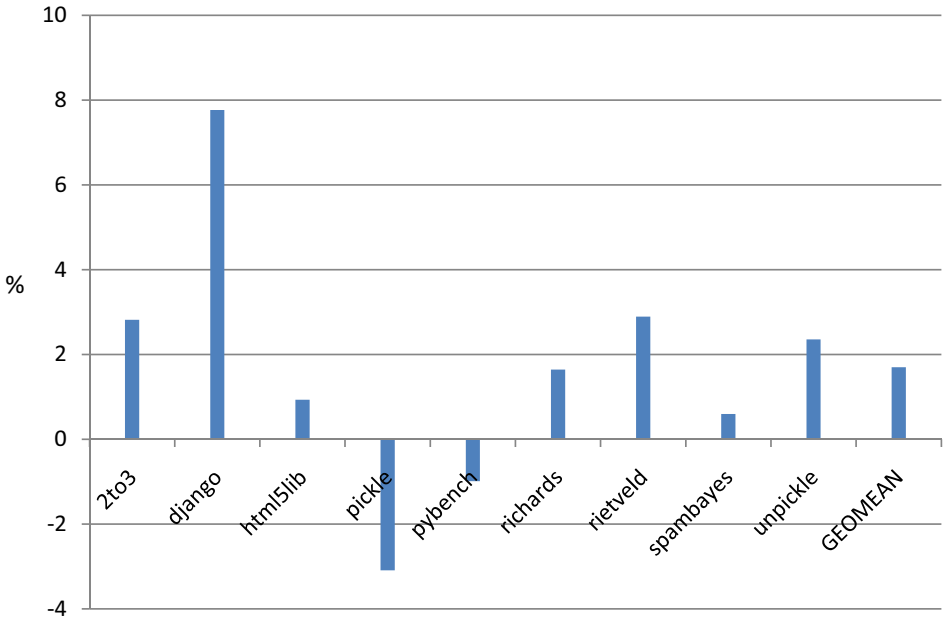


Figure 5.16: Speedup with a special ISINSTANCE bytecode

Finally, in Figure 5.17, we report the speedup of all three optimizations in combination, using all cache configurations. We achieve a maximum speedup of 28% and 15% on average. Multi-entry cache with swapping remains the best performing caching configuration for most cases. Multi-entry with no swapping and single-entry are quite similar performance-wise.

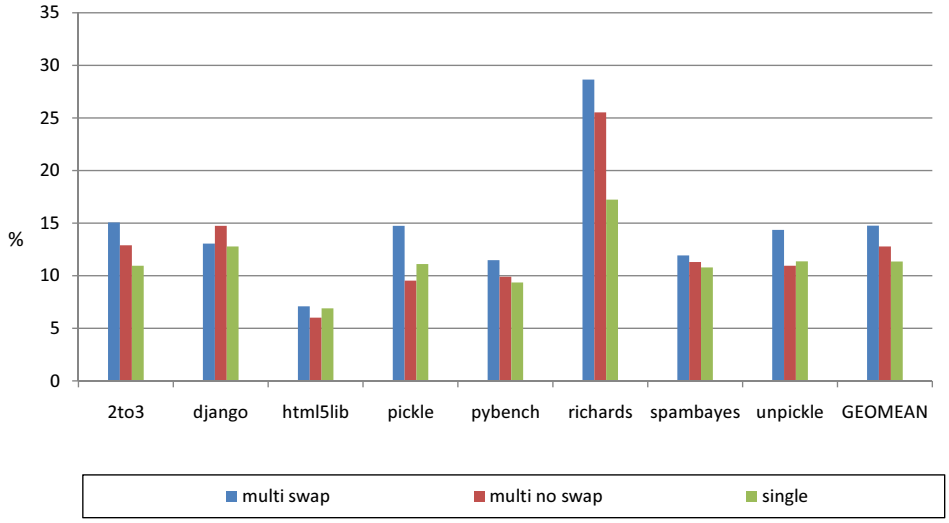


Figure 5.17: Summary of speedup when all optimization are enabled.

5.5 Related Work

In this section, we identify research contributions that characterize interpreter performance and that propose techniques for its improvement.

A work by Holkner et al. [55] aims to understand the extent and scope of use of dynamic language features like runtime object and code modification. In particular,

they examine whether Python programs only rarely use dynamic features, and whether this use of dynamic features is restricted to an initial startup phase in the application. For the programs and the set of dynamic features that they analyzed, they concluded that while programs do make use of dynamic features over the entire execution, this use is relatively higher during startup, thus lending themselves well to runtime analyses and feedback-directed optimization.

Hidden Classes is a caching optimization used in Google V8 JavaScript engine [117]. The idea is to have a table for objects with the same layout mapping attribute names to their offset in the instance object. This technique is well-suited for JavaScript since it is prototype-based and there is no notion of classes. In Python, that is not the case and every object is an instance of some class. In our results, we have shown that objects instantiated from the same class show, to great extent, identical layouts. This was demonstrated with the extremely low miss rate for the instance field cache in Section 5.4.1. Therefore, the actual object class acts as Hidden Class in our setting. Additionally, Hidden Classes are implemented in a compilation-based runtime while our approach is purely interpreter-based.

Similarly, in unpublished work [68], Lua language implementers employ caching within code generation. Code accessing hashes with constant keys are specialized for that key/hash. This is similar to polymorphic inline caching [25] where the code generated is specialized based on the outcome of method resolution.

5.6 Conclusions

In this chapter, we evaluate the performance of the Python language and CPython interpreter to identify sources of overhead. We classify the CPython bytecodes into categories and, by timing the bytecodes, are able to uncover that more than half of the execution time falls into three main categories: loading attributes, copying of locals to/from the operand stack and method calls. We devise three simple interpreter-based optimizations to target these sources of overhead. Attribute caching aims to minimize the needed dictionary lookups by caching the attribute or its location within the object. Load/Store elimination performs on-the-fly static analysis of the bytecodes to reduces movement of objects between the VM registers and operand stack. Finally, we inline C calls into the bytecode via a special bytecode. We analyze the efficacy and behavior of each of the optimizations and present the design trade-offs associated with every optimization. We demonstrate a performance improvement of up to 28% and 15% on average.

Chapter 6

The Remote Compilation Framework: A Sweetspot Between Interpretation and Dynamic Compilation

In this chapter, we present the Remote Compilation Framework (RCF), a feedback-directed compilation system as-a-service for Python programs. The goal of our work is to target an intermediary point in the runtime design space between interpreter-based runtimes and dynamic compilation systems for dynamic scripting languages to improve the performance of their programs without increasing the software complexity and memory footprint of their runtimes. RCF collects low-overhead, sample-based, across-input profiles for a Python program that is executed by users "in-the-wild". The information is communicated to a remote optimization server which constructs a global profile of type-annotated calling contexts. The server uses this profile to specialize the program via guarded speculative static compilation. RCF returns the specialized version of the program to users as a software update. We also present a phase detection

mechanism that enables RCF to detect shifts in the global profile and to adapt to them by recompiling the program.

We extensively evaluate our approach using community benchmarks and real applications. We show that RCF is able to produce code that approaches the performance of a popular dynamic compilation based runtime system for Python with average memory footprint that is $2.7 \times -7\times$ smaller. We also evaluate the accuracy and responsiveness of RCF phase detection and recompilation mechanism.

6.1 Introduction

The ubiquity of runtime systems for dynamic scripting languages (DSLs) is in part due to the use of efficient interpreters [39] for program execution that are written in C/C++ [28, 94, 81]. This runtime development strategy allows language designers to leverage mature and heavily supported tool-chains (e.g. GNU) for static compilation of their runtime systems on a wide variety of architectures, operating systems, and devices, without having to support a similar complex infrastructure themselves. This strategy also facilitates small runtime memory footprints and rapid deployment of new language features since the language semantics (in the interpreter) are implemented independently of and separately from the compilation system.

On the other hand, dynamic compiler systems for DSLs, especially those that perform feedback-directed specialization [91, 92, 51], have the potential to achieve significant performance gains over interpretation. These systems, however, are very complex, non-portable (code generation within the runtime must target a particular OS and ISA), and have much larger memory footprint relative to compiler-free runtimes. This complexity requires significant manpower for implementation, language evolution, and maintenance. This is particularly true because the semantics of these languages are defined by the latest interpreter release. With new versions of the interpreters coming out frequently, engineers have to focus on catching up and mimicking the language semantics instead of improving performance and footprint. This can all lead to abbreviated system/project lifespans [113, 91] or lack of support for the latest language release [87, 62, 59]. Moreover, these systems have a limited compilation budget which is exacerbated by the challenges imposed by code generation for the various dynamic features that these languages offer (object unboxing, type checking, polymorphic function dispatch, reflection, etc.).

In this chapter, we investigate a compilation-based solution that targets an intermediary point in the runtime design space between efficient interpretation and dynamic compilation in an attempt to improve performance of dynamic language programs while maintaining the simplicity, small footprint, and portability of their runtime systems.

An exception is JavaScript which has written specification (ECMA-262).

Our approach is to employ hybrid execution (a combination of compilation and interpretation) but to decouple the two: We employ an instrumented interpreter at the user's machine and feedback-directed static compilation at a remote *optimization server*. A user's runtime collects (with low-overhead) feedback from her use of a particular application "in-the-wild". The optimization service collects and merges this information from different users and uses it to specialize the application. It then returns the optimized version of the application to all users as part of a software update.

To enable such a model, we combine and extend past work on ahead-of-time (AOT) and remote compilation for static languages [96, 66, 78, 105], calling context profiling [127, 97, 12], and cooperative multi-user (multi-input) feedback collection [74, 73, 67]. We focus on Python as a representative dynamic language because of its widespread and similarities (language and implementation) to other popular dynamic languages such as Ruby, Perl and PHP.

We make two new empirical observations about the dynamic behavior of typical client-side Python programs that we then exploit to increase the performance potential of offline specialization of Python code. First, we find that dynamic object typing behavior is similar across different program inputs. As a result, we can draw on multiple runs of a program (and multiple users) for feedback about common type patterns to extract stable type information with only a small number of inputs. Second, we find that if we consider calling context, we can identify more opportunities for specialization.

We act on these two observations to collect and aggregate calling-context-aware type information from multiple users of a program (to spread the overhead of its collection across multiple users so that it is very low for any particular user). We specialize the hot methods that we identify from the aggregate profile by translating Python to C with static type information. In our current prototype, we use a combination of by-hand optimization and type-annotation of the Python code and automatic type-specialization and translation to C using Cython [29]. We use traditional static compilation (gcc) to compile the code. We guard the specialized code using checks to guarantee correct execution. The resulting optimized code can be executed using any unmodified Python interpreter.

We also integrate a phase detection and recompilation mechanism within RCF that enables it to continuously adapt to changes in program behavior. This is particularly important for larger applications that have multiple components to them which can cause different inputs to exercise totally different parts of the code resulting in variations in behavior over time. In such case, there is no guarantee that the aggregate profile will not deviate after the optimization point. Instead of performing a one-time profile collection and optimization, RCF can continuously adapt to changes in program behavior. Our technique is based on further monitoring of the aggregate profile beyond the optimization point. If enough changes are observed that will violate RCF optimization assumptions, RCF will trigger recompilation of the code based on the new profile.

We empirically evaluate RCF and its components for Python using a set of community benchmarks as well as real applications. We find that our system effectively extracts type specialization opportunities via sample-based multi-input profiles to significantly improve performance ($1.1 \times -1.7\times$ for real applications and $1.3 \times -3.4\times$ for community benchmarks). Moreover, we are able to do so without significantly increasing memory footprint of the runtime. We also find that RCF sampling imposes very low overhead ($<2\%$) and, for the workloads tested, converges to a stable profile very quickly. In addition, by considering calling context, we expose significantly more specialization opportunities for some applications. We compare RCF to a popular Python runtime that employs dynamic compilation, feedback-directed optimization, and type/value specialization: PyPy [87]. We find that RCF facilitates similar performance gains with a $2.7 \times -7\times$ smaller memory footprint and a significantly simpler and portable user runtime system implementation. Finally, we evaluate RCF phase detection accuracy and responsiveness to shifts in program behavior.

The rest of this chapter is organized as follows. In Section 6.2, we present our remote compilation framework. Section 6.3 presents our profiling approach (calling context trees and across-input sampling). In Section 6.4, we present the RCF remote optimization service and describe our approach to program specialization. Section 6.5 describes RCF phase detection and recompilation mechanism. We then present our

methodology in Section 6.6 and analysis and empirical results in Section 6.7. We finally discuss related work and conclude (Sections 6.8 and 6.9).

6.2 Remote Compilation Framework

The goal of our work is to investigate whether it is possible to develop a system that gleans the benefits of both efficient interpretation and feedback-directed compilation without imposing their drawbacks. Our solution is the *Remote Compilation Framework (RCF)*, a hybrid optimization framework that decouples interpretation from feedback-directed optimization. RCF relies on performance profiles collected from users "in the wild" to guide remote code optimization via feedback-directed static compilation. The efficacy of RCF stems from our observation that type profiles tend to be very similar across inputs. Hence, it is safe to aggregate profiles from different inputs without masking the program's per-input characteristics. As a result, RCF needs only to collect a small amount of information from each user and employ a small number of profiles to identify stable behavior across inputs and users. To preclude the need for a complex runtime implementation on the user's platform, we move the advanced optimization system to a remote optimization server. An optimization server can be used to manage one or more applications at a time.

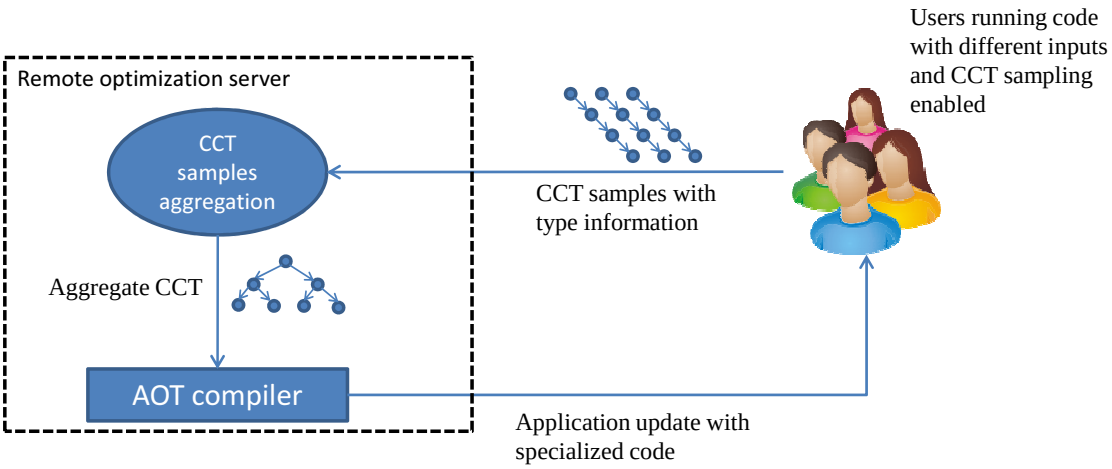


Figure 6.1: Remote Compilation Framework (RCF) overview.

Figure 6.1 overviews RCF. While users are running a program with different inputs and use cases, the runtime system (an instrumented interpreter) collects samples randomly about the program's behavior. Since samples come from numerous users, per-user sampling rate is very low, thus incurring negligible overhead. The user's runtimes send samples to an optimization server where they are merged to form an aggregate profile that identifies frequent and common program behavior across inputs.

A feedback-guided static compiler at the optimization server uses the aggregate profile to identify hot methods to optimize, specializes methods based on their type profiles, optimizes monomorphic call sites, and performs inlining and other optimizations. Since we cannot prove all such optimizations to be correct statically, we surround such program points with guards that fall-back to the generic version of the specialized code. The specialized program is then sent back to the users in the form of an appli-

cation update. This application can be executed using an unmodified (or instrumented) runtime. We next detail the main components of RCF: profile collection, the remote optimization service and phases detection.

6.3 RCF Profiling

RCF employs calling context trees as its profile representation. RCF collects these from users "in the wild" using sample-based profiling. In this Section, we detail each of these approaches.

6.3.1 Calling Context Tree Profiles

A Calling Context Tree (CCT) [4, 12, 127, 97] is a representation of program behavior as a tree for which nodes represent methods and directed edges represent calls between methods. An edge from node *A* to node *B* is a call from method *A* to method *B*. An edge exists between two nodes if and only if a call took place during execution. Hence, any path from the tree root to any node represents a calling context. Multiple calls to the same method from the same calling context are merged into one node.

In RCF, we extend the traditional CCT to distinguish call sites. This means that if *A* calls *B* twice but from different call-sites then there will be two different nodes for *B* under the same context. Distinguishing nodes by call sites is essential to enable per-call-

site and per-context specialization; without it, a method's behavior under a particular call-site or call-context may appear highly variable and specialization opportunities can be missed.

Our CCT representation captures method behavior at three levels: per-context, per-call-site, and across all call-sites. We extract per-context behavior trivially by inspecting the annotations of the CCT node that represent that method for a specific context. We merge CCT nodes for the same method and call-site to obtain per call-site behavior. Finally, we merge per-call-site behavior to yield the global behavior for that method. Using this information we are able to identify type-static, per-context, call-sites as candidates for type-specialization and partial evaluation. We show that using context information greatly increases the number of candidates that have a single call target (Section 6.7).

We annotate every CCT node with information about types and execution frequency of the method it represents. The key annotations that we employ are:

- **Execution Count/Time:** Method invocation count plus the number of times a back-edge was taken. Most dynamic compilation systems use this profile to identify methods that consume a majority of the execution time. We shall refer to this as execution count or time, interchangeably.

- **Argument Types:** A histogram of tuple counts for each set of arguments passed into each method. The higher the number of histogram entries, the more dynamic

the method is. We refer to methods with one unique argument tuple as *single-typed methods* (vs *multi-typed methods* for those with more than one). We refer to the call-sites that call these methods as *single-typed call-site* and *multi-typed call-site*, respectively.

- **Bytecode Types:** For the most common CPython bytecodes (e.g. loads/stores, function calls), we record the type of objects on which they operate or return. For example, for stack loads, we record the types of the objects pushed on the stack. By profiling for these bytecodes only, we reduce profiling overhead while being able to easily infer the types for all other bytecodes. We store bytecode types in a table indexed by bytecode index (bci). For every bci, we record a type histogram of the types seen by (and their frequency at) that bytecode. We call bytecodes with a single entry in the type histogram *single-typed bytecodes*.

6.3.2 Multi-Input Remote Profiling

To collect CCT samples from different users, the users employ a standard CPython interpreter [28] that we extended to collect sample-based profiles from Python programs. These interpreters can communicate the samples during execution or store them to disk for off-line communication to an RCF remote optimization service.

This is because the CPython source to bytecode compiler has a simple, non-optimizing bytecode generator.

The interpreters sample only Python code (CPython bytecodes, not native code). Each CCT sample consists of a sequence of CCT nodes describing the calling context at the point during execution at which the sample is collected. A leaf node in the CCT sample is the sampled method; only leaf nodes contain annotation data.

The interpreters initiate sampling on a sampling event: either a method call or a backward branch. For each sampling event, the runtime decrements a counter. When the counter expires, the runtime samples the code and collects bytecodes type information, until the next sampling event occurs. When this happens, the runtime resets the counter, terminates the sample, and buffers at the user side. When the buffer overflows, the interpreter writes the samples to disk or transmits them to an optimization server.

The optimization server merges samples into an aggregate CCT using a method similar in spirit to that described in [12] but in our case we do so across multiple inputs and multiple users. To merge a sample, we search for the sample context in the aggregate CCT, if the context already exists, we augment the type profile of the sample to that of the corresponding aggregate CCT node and increment its frequency. Unmatched samples are added to the CCT as new nodes and edges.

Figure 6.2 shows an example of merging a sample with the context present in the aggregate CCT (a), and without (b).

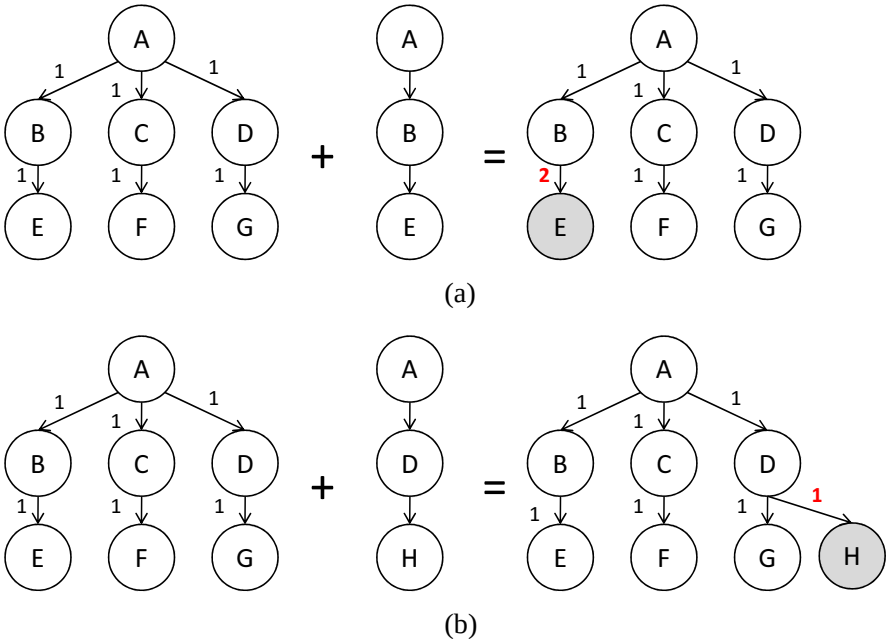


Figure 6.2: Example of sample aggregation. In (a), the sample context and the leaf node already exists, hence the leaf node *E* frequency is incremented. The type profiles of that node are merged. In (b), the sample leaf node is missing, hence a new node *H* is added to the tree with frequency 1.

6.4 RCF Remote Optimization Service

Bytecode	Generic code	Specialized code

LOAD_FAST 1
LOAD_FAST 2
BINARY_ADD
STORE_FAST 3
	c = PyNumber_Add(a, b)	if (type(a) == List and
		type(b) == List)
		{
		c = PyList_Append(a, b)
		} else {
		c = PyNumber_Add(a, b)
		}

Figure 6.3: Example of specializing addition of two objects if the types are known to be Lists.

Across-input profile aggregation has the potential to identify more dynamic behavior than what is actually present in per-input profiles. An aggregate may, for example, cause single-typed methods to become multi-typed or monomorphic call-sites to be polymorphic. For example, if the method *add* is called with argument tuples *(int,int)*, *(float, float)* and *(str,str)* for inputs I_1, I_2 and I_3, respectively, the aggregate profile in this case will report *add* as a multi-typed method with three argument types. Our optimization service in such cases will not specialize *add* (based on types), but instead it emits generic code and potentially miss an opportunity that would otherwise benefit some subset of users.

Fortunately, we find with this work, that this is not the case in practice. We find that for the benchmarks and real client-side Python programs that we have studied, per-input, static behavior tends to remain highly similar across inputs (Section 6.7). Hence, we are able to rely on an aggregate profile to improve performance even for

new inputs that did not contribute to the aggregate profile. In this section, we describe the optimizations that the RCF optimization server employs. This includes more traditional type specialization as well as new techniques for feedback-directed optimization that are particularly useful for dynamic languages, including clones dispatching and context-aware specialization.

6.4.1 Type Specialization

Type specialization aims to reduce the number of type-checks, unbox operations, indirect calls, and other forms of overhead associated with type-generic code. For example, if an arithmetic operation is performed on two Python objects, then the implementation must first resolve the types of the two objects, find the correct method associated with that operation, and indirectly call it. However, if the types of the two objects are known to be integers, for example, then the compiler can unbox these objects directly and can directly implement (lower) the arithmetic operations.

Figure 6.3 shows an example of specializing addition of two Python objects. The bytecode loads two operands (a and b), adds them and pushes the result on the stack, and stores the result to a local variable c. Since the semantics of the addition operation is defined by the types of its operands, the generic code makes a call to *PyNumber_Add* which indirectly dispatches the correct method for addition based on the type of the receiver a. If the aggregate profile reports that the types of the two *LOAD_FAST*s are

Lists, then the specialized code can directly invoke the correct method to append *Lists*. For correctness, the specialized code must be guarded by type-checks to ensure that the types of objects *a* and *b* are indeed *Lists*. If the guard fails, we fall back to the generic code.

It is also possible to generate more specialized code for other types and add it to an if-else-if ladder. In RCF, however, we specialize code for single-typed bytecodes only (bytecodes with single type in the aggregate profile). For multi-typed bytecodes, the compiler generates generic code. This simplifies our specialization system and achieves almost no guard overhead on today's superscalar machines. RCF can be extended to handle more type-generic cases.

6.4.2 Clones Calling Mechanism

We refer to the method bodies that the optimization service specializes for particular types as *clones*. In RCF, we generate clones only for hot, monomorphic call-sites. Although it is possible to have multiple clones for polymorphic and/or multi-typed call-sites, we chose to have one clone per-call-site for simplicity. If a call-site is polymorphic, we do not specialize for it; if it is multi-typed, we generate one clone only specialized for the single-typed bytecodes within.

When RCF generates a clone for a call-site, its compiler associates an ID number with the target method object and its clone. At the call-site, the runtime first resolves

the method by name. If the method body has the expected ID number, its clone is invoked instead. Figure 6.4 shows the code for clone dispatching. Each clone has a global guard in its prologue that type-checks the method argument, if any of the checks fail, the generic method is called instead. The global guard is redundant in terms of correctness, since every specialized piece of code is already guarded. However, if the argument type checks fail, we have found that it is likely that many guards will fail inside the method body. We thus decide that in such cases to switch to the generic version early.

```
resolve method "foo" on the receiver object
if foo.id = expected_id then
    call clone
else
    PyObject_Call(foo)
endif
```

Figure 6.4: Clone calling mechanism. The guard checks if the resolved method is the expected one. If false, the generic method is called instead.

6.4.3 Direct Calls and Inlining of Clones

Since method resolution takes place before the clone is dispatched, the compiler can emit code to call the clone directly, using the C calling convention, instead of using the slower calling convention that Python implements. This eliminates the extra work of packing arguments into tuples in the caller and unpacking them in the callee, following

the Python calling convention. We also dispatch built-in methods directly, without method resolution, since Python disallows built-in methods to be modified. With the same checks as shown in Figure 6.4, these method clones can also be inlined. For example, a call to the *append* method on a *List* object receiver can be called directly or inlined, provided that a type check on the receiver is present.

```
resolve method "foo" on the receiver object
if current_cntxt_hash = expected_hash and
   foo.id = expected_id then
    call clone
else
    PyObject_Call(foo)
endif
```

Figure 6.5: Per-context clone calling mechanism. The guard checks if execution is at the correct context hash and if the resolved method is the expected one.

6.4.4 Context-aware Specialization

As part of this work, we also discovered that we can partition dynamic call-sites by calling context to reveal significantly more static behavior in some programs. Polymorphic call-sites in many of the programs we studied are monomorphic within one or more of its calling contexts. This enables us to use optimization service to more aggressively specialize programs. Context-aware clone resolution is similar to normal clone resolution (Figure 6.5) except that we require an additional comparison on the context identifier as part of the guard that determines if the clone should be dispatched.

We implement the method described in [22] to facilitate low-overhead deterministic hashing to encode calling contexts during execution. The hashes this technique generates are not unique, but the chance of collisions are extremely low (3 in 10 billion for a 64-bit hash value). On a method call, the runtime evaluates the next context hash as $f(V, cs) = 3 \times V + cs$, where V is the current context hash and cs is the call-site hash (hash of the call-site source code file name and line number). We store the context hash in the corresponding call-stack frame, so no hash updates are needed on returns.

Note that despite the slight probability of a hash collision, even if it happens, it can never lead to incorrect execution. This is because a call to a clone is guarded by a check on the resolved method ID. We thus guarantee that the clone dispatched is of the correct method regardless of the context hash. A hash collision will only lead to dispatch of an incorrect clone of that method which will cause the guard in the clone prologue to fail, and execution will eventually fall to the generic code. Therefore, the highly unlikely collision hash can only cause unnecessary execution of a clone prologue. In our experiments, we have not observed any collision in hot contexts.

6.5 Phase Detection and Re-Compilation

RCF is based on samples collection from users in the wild where it tracks the application behavior as it is being executed with different inputs. During the profiling

period, RCF merges samples from various users to form an aggregate profile. Once the profile is stable enough, RCF decides to optimize and sends an update to the users. In our current scheme, RCF becomes unaware of future changes in program behavior past the optimization point. This is generally acceptable for most client-side Python applications. These applications are relatively single-purpose with highly peaked profiles that are highly similar across inputs. For these applications, a one-time profile-based compilation is usually sufficient.

A problem starts to arise if an application is large enough to have multiple components to it. This is possible for complex client-side programs and server-side applications and frameworks (e.g. Python/Django, Python/TurboGears, Ruby/Rails, PHP/Trax, Google App Engine [52], AppScale [26], etc.). Multi-component applications can employ different inputs that exercise totally different parts of the code yielding high differences across per-input profiles and possible variation in the aggregate profile over time. In such cases, there is no guarantee that the aggregate profile will not deviate after the compilation point which will reduce the impact of the optimizations. For such cases, RCF needs an adaptive mechanism in which it detects profile changes and reacts to them by recompilation. Such changes can originate from:

1. **Change of popularity of inputs among users:** A less popular input may become popular in the future leading the aggregate profile to be more biased towards that input.

2. **New input(s) emerge:** New inputs may be exercised in the future that were not part of the collected aggregate profile.

3. **Old input(s) disappear:** Some inputs that contributed essentially to the aggregate profile may become rarely exercised in the future.

RCF needs to dynamically react to these possible changes by adjusting its optimization plan, recompiling the code and sending an updated version to the users. This calls for a detection mechanism that fulfills the following:

1. Fast detection of phase shifts.

2. Short delay between phase shift detection and start of recompilation.

3. Ability to distinguish long stable phases from short transient ones.

In this Section, we present this mechanism and explain the heuristics behind it.

6.5.1 Profile Comparison

Phase detection is based on comparison of the aggregate profile at different points in time. Profiles overlap provides such means of comparison by measuring how much one profile is covered by another. For a given profile, RCF optimizes the top entries covering 50% of the execution counts (invocation and backedge counts). Following this optimization policy, our overlap metric measures how much the top 50% of one

profile covers in another profile. In other word, if we were to compile the top 50% based on one profile, how much of the other profile execution counts get covered by the compiled code. Figure 6.6 illustrates how we calculate overlap between two profiles P and Q. The computation relies on the idea of *suboverlap* that we will explain shortly.

```
# P and Q entries are sorted in non-increasing order
# based on their execution frequencies
function overlap(P, Q)
begin
    result = 0
    L = 50% execution frequency mark in P
    total = sum of execution frequencies in Q
    for i = 1 to L
    do
     # get entries
        p = P(i)
        q = find entry in Q identical to p
        if q is not null then
            result = result + freq(q) * suboverlap(p,q)
        endif
    done
    return result / total
end
```

Figure 6.6: Overlap computation. The function finds how much of Q is covered by the top 50% of P's total execution frequencies.

In RCF, every profile entry consists of a call-site and a corresponding call-target. We associate with each entry a profile of sorted arguments types describing what types of arguments are passed to the call-target and by what frequency. The arguments types profile defines how a method is type-specialized for a particular call-site. Therefore,

when computing overlap, it is important to take into consideration how similar the arguments types profiles are for every matched profile entries. We accomplish this via a metric we call *suboverlap*. The suboverlap of two profile entries is the overlap between their argument types profile. It is computed by comparing the two type profiles strictly in order: Two type profile entries must exist exactly in the same position. We compute suboverlap in order, since that is the way an optimizing compiler will look at the type profiles when specializing (i.e. optimize for the most frequent case, then the second most frequent, etc.)

A suboverlap ranges from 0 (entirely different type profiles) to 1 (identical type profiles), inclusive. Figure 6.6 shows how suboverlap is used to weigh the frequency of every matched profile entry. If the two profile entries have identical type profiles, the coverage is full, and we add the frequency of the covered profile entry. If they have completely different type profiles, suboverlap is 0 and the matched profile entry is ignored. Figure 6.7 illustrates how suboverlap is computed.

Since we perform suboverlap for successive type profile entries, it is sensitive to the order of these entries. If two type profiles have the same entries but are ordered differently, suboverlap will be zero. Since in RCF the profiles are sampled and thus inexact, suboverlap order can cause a problem. Consider the sampled type profiles shown in Table 6.1. Assume, in the complete exhaustive profile, the frequency of types signatures A and B are exactly equal and that the same is true for C and D. One can

```
// TP and TQ are argument types profiles that
// are sorted in non-increasing order
function suboverlap(TP, TQ)
begin
    result = 0
    L = min(length(TP), length(TQ))
    total = sum of execution frequencies of TQ
    for i = 1 to L
    do
        // get entries
        tp = TP(i)
        tq = TQ(i)
        if tp equals tq then
            result = result + freq(tq)
        endif
    done
    return result / total
end
```

Figure 6.7: Suboverlap computation. The suboverlap function takes two sorted type profiles (TP and TQ) as input. Unlike the overlap computation, suboverlap matches entries strictly in order.

see that if that profile is sampled the frequencies are unlikely to be identical due the randomness of the sampling process. Therefore, although in Table 6.1 the profiles seem quite similar, the suboverlap will be zero.

Arguments Types Profile 1		Arguments Types Profile 2	
Types signature	Frequency	Types signature	Frequency
A	2001	B	2003
B	1999	A	1997
C	501	D	503
D	498	C	495

Table 6.1: Two sampled profiles. Although type signatures A and B are supposed to have the exact frequency, their frequencies are not identical due to sampling randomness. The same holds for C and D. This small variation causes the suboverlap to be zero.

Such situations call for the suboverlap computation to tolerate small differences in frequencies. We modified the suboverlap computation to take the sampling randomness into account. Instead of comparing the profiles with strict ordering, we divide the profile into an ordered list of sets or buckets. We scan the profile in order and in every set we place entries with *nearly equal* frequencies. We consider any increase in frequency below 10% between one entry and the next as nearly equal. We chose 10% based on empirical observation that sampling causes $\pm 10\%$ fluctuation in equal frequencies. After we construct the sets, we compare them in order, but within every set, we compare the profile entries with no ordering. Figure 6.8 shows the improved version of suboverlap computation. Table 6.2 shows the profiles from Table 6.1 after splitting them into sets. Elements of set S1 from both profiles are compared with no ordering and both

```
// TP and TQ are argument types profiles that
// are sorted in non-increasing order
function suboverlap(TP, TQ)
begin
    result = 0
    total = sum of execution frequencies of TQ
    divide TP into a list of sets SP
    divide TQ into a list of sets SQ
    L = min(length(SP), length(SQ))
    for i = 1 to L
    do
        sp = SP(i)
        sq = SQ(i)
        for j = 1 to length(sp) do
            for k = 1 to length(sq) do
                if sp(j) == sq(k) then
                    result = result + freq(sq(k))
                    break
                endif
            done
        done
    done
    return result / total
end
```

Figure 6.8: Modified suboverlap computation to tolerate sampling noise. Suboverlap function takes two type profiles (TP and TQ) as input. It constructs a list of sets of nearly equal entries for every profile and then compares them in order. Comparison within every set is with no order.

Arguments Types Profile 1	Arguments Types Profile 2
S1={A:2001, B:1999} S2={C:501, D:498}	S1={B:2003, A:1997} S2={D:503, C:495}

Table 6.2: The profiles after splitting them into sets. The elements of each pair of sets are compared with no order. Final suboverlap is 1.

elements will be matched. The same applies to set S2; that makes the final suboverlap equals to 1.

6.5.2 Phase Detection

We aim to detect significant deviation in the aggregate profile from the last profile RCF used for optimization. If a sufficient and stable deviation is detected, RCF triggers recompilation according to the new profile. We next explain how we employ overlap and profile comparison in our phase detection algorithm.

Profile Snapshot

RCF receives a stream of samples from users in the wild and merges them into an aggregate profile. To compare the aggregate profile at different points in time, we need to record a copy of it at these points. We call this copy a profile snapshot or *Snapshot* for short. We continuously record snapshots at equal short intervals (*Snapshot Interval (SI)*) of received samples (e.g. every 100 samples). Snapshots are accumulative which means earlier snapshots are subsets of later ones.

Convergence Cycle

By computing the overlap between the latest snapshot and the snapshot of the profile used for optimization, we can detect whether the aggregate profile has changed enough. If the resulting overlap falls below a threshold, we declare a phase shift. This approach, however, can be very slow in detecting phase shifts and some can even go undetected. Consider, for example, a profile that has been stable for a long time. The more samples received, the more the profile becomes skewed towards a particular set of hot methods. Now, if a phase shift occurs, we will see samples from different methods being received. However, since the profile is already skewed, it will take a long time for these samples to affect the profile shape (skew it towards a different set of methods). This makes the phase detection delay dependent on how much time the application has spent in the previous phase. Also, if the new phase is relatively shorter than the old one, it might not contribute enough samples to alter the profile significantly, and it can thus go undetected.

To overcome these problems, we merge samples into the aggregate profile for a profiling interval then flush it and start over again. We refer to this interval as *Convergence Cycle (CC)*. A CC always starts with an empty aggregate profile. During a CC, samples are merged as they are received and snapshots are recorded periodically. RCF inspects the snapshots and detects when the profile has converged. It then decides to flush the profile and start a new CC.

Note that the CCs terminate only when the profile has converged instead of being of fixed length. There are two reasons for this. First, as we will explain later, phase detection depends on comparing last snapshots of every profiling interval. Since RCF collects samples from different sources, we expect a significant degree of randomness in the samples received and the generated snapshots. A fixed-length interval will terminate regardless of whether its last snapshot is representative or not. This can yield fluctuations in snapshots and false phase detections. On the other hand, a convergence-based interval will never terminate during fluctuation periods of the profile. This provides some certainty that the last snapshot is representative of the program behavior during that interval. Second, the length of the profiling interval dictates how the intervals align with the phases. This makes the choice of the fixed length quite difficult and application-dependent. CCs, however, are adaptive to how the profile changes over time.

RCF detects a CC convergence by repeatedly computing the overlap between the latest snapshot S_i and another snapshot in the past S_{i-d}, where d is the *Snapshot Distance (SD)* between them. If the overlap remains above a *Convergence Threshold (CT)* for a specified number of snapshots (*Convergence Cycle Duration (CCD)*), RCF terminates the CC and starts a new one. Figure 6.9 shows the steps of CC convergence detection. Note that if the overlap drops below CT, the *duration* count is reset. Only when *duration* reaches CCD we declare the CC converged. Figure 6.10 demonstrates

how the overlap varies for four CCs. Note how flushing the profile drops the overlap to zero at the start of every CC. The overlap then rises above CT and remains there for CCD snapshots until the profile is flushed again and a new CC starts.

```
// pastS: past snapshot
// curS: current snapshot
// curS and pastS are SD snapshots apart
function hasConverged(pastS, curS)
begin
    ol = overlap(pastS, curS)
    if ol > CT then
        duration = duration + 1
        if duration = CCD then
            return True
        endif
    else
        duration = 0
    endif
    return False
end
```

Figure 6.9: Convergence detection for a CC. curS is the latest snapshot. pastS is SD snapshots in the past.

Phases

RCF views a phase as a sequence of CCs with homogeneous behavior. Every CC is represented by its *Last Snapshot (LS)*. The LS represent the last recorded state of the converged profile for that CC before its flushed. To determine similarity between two CCs, RCF computes the overlap between their LSs. To detect need for recompilation, RCF performs the following steps which are showed more formally in Figure 6.11.

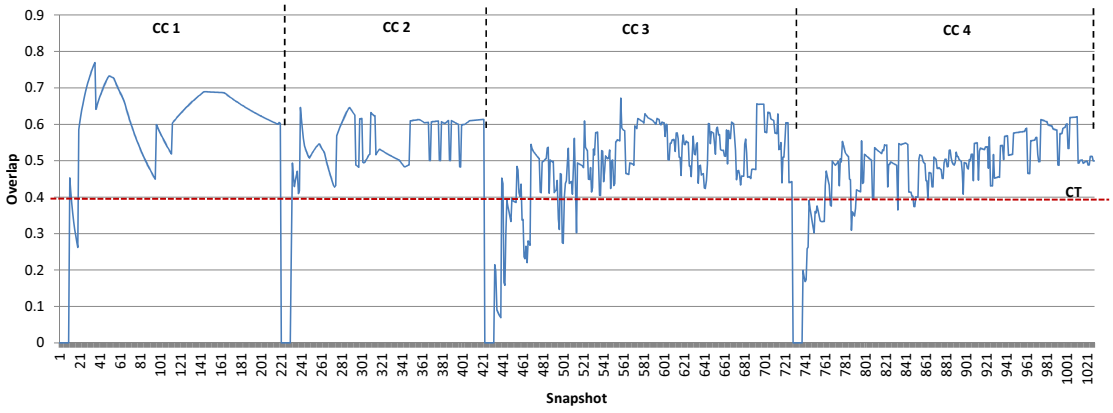


Figure 6.10: Example of four convergence cycles (CCs). The Y-axis is the overlap with snapshot distance (SD) = 10. The X-axis is the snapshot count. Snapshot interval (SI) is set to 100 samples. The convergence threshold (CT) is marked at 0.4. The overlap fluctuates at the start of every CC but stabilizes after 100 snapshots. CCD is set to 200 after which a new CC is started.

1. **Detect a phase shift:** RCF keeps track of the LS of the CC where the last compilation occurred. That is the aggregate profile upon which the last compilation was based. For every CC that ends, RCF compares its LS with the LS from the last compilation. If the overlap falls below a threshold (*Phase Threshold (PT)*), RCF detects a phase shift.

2. **Wait for a stable phase:** This steps measures the homogeneity of the CCs after the phase shift to detect if the profile has entered a new stable phase or is just going through a transient period. To achieve this, RCF compares the current LS with the previous one. If the overlap remains higher than the PT for a continuous number of CCs (*Phase Convergence Duration (PCD)*), the phase is stable again.

```
// state and duration are global variables
// optLs: LS of last optimization
// curLS, prevLS: current and previous LSs
// return True if recompilation is needed, False otherwise
function recompile(optLS, curLS, prevLS)
begin
    if state = phase_stable then
        ol = overlap(optLS, curLS)
        if ol < PT then
            // move to phase shift state
            state = phase_shift
            duration = 0
            return False
        endif
    else if state = phase_shift then
        ol = overlap(prevLS, curLS)
        if (ol > PT) then
            duration = duration + 1
            if duration == PCD then
                // phase has been stable long enough
                // move to stable phase state
                state = phase_stable
                duration = 0
                // is it a different phase ?
                ol = overlap(optLS, curLS)
                if ol < PT then
                    return True
                else
                    return False
                endif
            endif
        else
            // another phase shift interrupts
            duration = 0
            return False
        endif
    endif
end
```

Figure 6.11: Phase detection algorithm. The function takes the last optimization LS (`optLS`), the current LS (`curLS`) and the previous LS (`prevLS`).

3. **Check for transition to a new phase:** Now what remains is to determine if the reached stable phase is a new phase or just a continuation of the previous stable phase. This is achieved via one last comparison between the newest LS and the LS from the last compilation. If the overlap is below PT, this is a new phase and is different from the previous stable phase for which we optimized. If not, then the phase shift was just a temporary fluctuation of the profile and no recompilation is needed.

6.6 Experimental Methodology

To investigate the efficacy of the various design decisions we make in RCF and to evaluate its potential and overhead, we use 12 real Python applications (written by others and made available as open source via the web). We overview our application suite in Table 6.3. The last column of the table indicates whether the application uses `C`, a command-line interface (CLI) or `G`, a graphical-user interface (GUI). We use these programs to investigate the effect of across-input profile aggregation, the overhead due to sampling calling contexts and type profiles, the rate of profile convergence, and the potential of context-awareness in identifying monomorphic, single-typed calls.

We analyze the performance of RCF optimization using a subset of applications (`docutils`, `2to3`, `markdown`, `gourmet`, and `pdfshuffler`), chosen arbi-

trarily. We also employ a set of microbenchmarks (`binarytrees`, `fannkuch`, `fasta`, `meteor`, `nbody`, and `mandlebrot`) that have been used in other studies on DSL optimization. These microbenchmarks come from the Programming Language Shootout [103]; we overview their functionality in Table 6.4. We compare the performance of RCF-based optimization with that of a popular dynamic compilation and type specialization system: PyPy [92]. For this study we use PyPy versions 1.5 and 1.6 [87], and perform the comparison using the subset of these programs that PyPy is able to run. These programs are `docutils`, `markdown`, `fannkuch`, `fasta`, and `meteor`.

For each program, we consider four different inputs that we have generated as test inputs and through arbitrary use of the programs (e.g. the GUIs) by students. We chose the inputs to exercise different functionalities of the programs under study. We generate profiles for three of the four inputs, on a per-thread basis, using CPython (v2.6) [28], extended with our sample-based profiling support. For every application, we combine profiles from three inputs into one global aggregate profile. We use the fourth input to analyze the efficacy of profile-guided optimization across-inputs (for an input not used in the profile aggregation step).

Our optimization step relies on Cython [29], a Python-to-C translator and optimizer that translates Python code with type annotations into efficient C extension modules. We start by applying a set of by-hand Python-level optimizations and type-annotations

Name	Description	G/C	LOC
2to3-2.6	Python 2.x to 3.x translator	C	83469
Brainworkshop-4.8.1	Memory trainer	G	61531
Doctuils-0.7	Text to HTML/Latex converter	C	58327
DrPython-3.11.3	Python IDE	G	16568
Markdown-2.0.3	Text to HTML converter	C	2569
Gourmet-0.15.7	Recipe manager	G	43714
PdfShuffler-0.5.1	PDF documents management tool	G	1031
PyParsing-1.5.5	General grammar parsing tool	C	10581
Solarwolf-1.5	Arcade game	G	5270
TowerDefense-0.5	Arcade game	G	3884
w3af	Web attack and audit framework	G	58133
wapiti	Web apps vulnerability scanner	C	5810

Table 6.3: Description of 12 real Python applications evaluated. Applications with a graphical user interface are marked with 'G', those with a command-line interface (CLI) are marked with 'C'.

Micro Benchmark	Description
Binarytrees	Allocate and deallocate many binary trees
Fannkuch	Repeatedly access a tiny integer-sequence
Fasta	Generate and write random DNA sequences
Mandelbrot	Generate a Mandelbrot set and write a portable bitmap
Meteor	Search for solutions to shape packing puzzle
Nbody	Perform an N-body simulation of the Jovian planets

Table 6.4: Description of Python microbenchmarks applications evaluated.

to hot code (Section 6.7.4). We then compile the optimized Python code to C using Cython which automatically performs the required specialization and adds the necessary guards as needed. Although done manually, the optimization step can be automated as part of a Python AOT compiler. We use Cython v0.14.1 and gcc v4.4.3.

For our experiments, we execute the programs 10 times within a test harness and report the average (with bars for standard error of the mean), across all but the first

warmup run. We consider the warmup run separately, for which case we execute the programs without the test harness 10 times and report the average and standard deviation across all 10. Our execution platform is a Linux-2.6.32-27 machine running on an Intel Core i5 clocked at 2.67GHz, with 8GB of memory.

A subset of our applications are GUI-based. To measure speedup, we recorded usage scenarios for each application and replayed them multiple times using Sikuli [104]. These applications rely on PyGTK [86], a C library for Python, for user interaction and graphical display. To measure performance (speedup) for these programs, we measure the execution time of the optimized call-sites only. We exclude the GTK loop and other non-optimized Python code due to the non-determinism of the system and the challenges it poses to repeatable execution time collection.

6.7 Empirical Evaluation

We next empirically evaluate RCF. We first consider the efficacy of its profiling component and then present its potential for improving performance of Python programs.

6.7.1 Effect of Profile Aggregation

In this section, we quantify the similarity between the aggregate profile and per-input profiles that contribute to the aggregate, in both the amount of recurring type behavior and method hotness. We consider similarity at the method (call-site and body) and the bytecode level in this section. High similarity indicates little information is lost in the aggregation process, and as such, an optimization plan that is guided only by the aggregate profile can benefit programs that employ its constituent inputs. The amount of similarity across inputs indicates the potential for across-input and ahead-of-time optimization.

We first evaluate behavioral similarity for each input and the aggregate using the time spent in methods called from monomorphic call sites (single target call sites) in Figure 6.12. The Y-axis is execution time (approximated by method call and back-edge counts) spent in these calls normalized to the total time. The ratio is almost identical (3% variation) across inputs and the aggregate profiles for all applications except *pyparsing* and *w3af*.

In Figure 6.13, we perform the same evaluation but for time spent in methods called from monomorphic call sites that are single-typed (for which there is a single argument (type) tuple). As expected, there is less time spent in these methods than if we disregard argument tuples. For example, around half of *pyparsing* call-sites are monomorphic, yet only 10% are monomorphic to single-typed methods. On the contrary, the meth-

ods called from monomorphic call-sites in *drpython* are almost all single-typed. This second graph also exhibits higher variation across inputs. For *docutils*, *gourmet*, *pyparsing* and *w3af*, one or two of the inputs have different ratios than the rest. For all other applications, the aggregate profile remains similar to that of the per-input profiles.

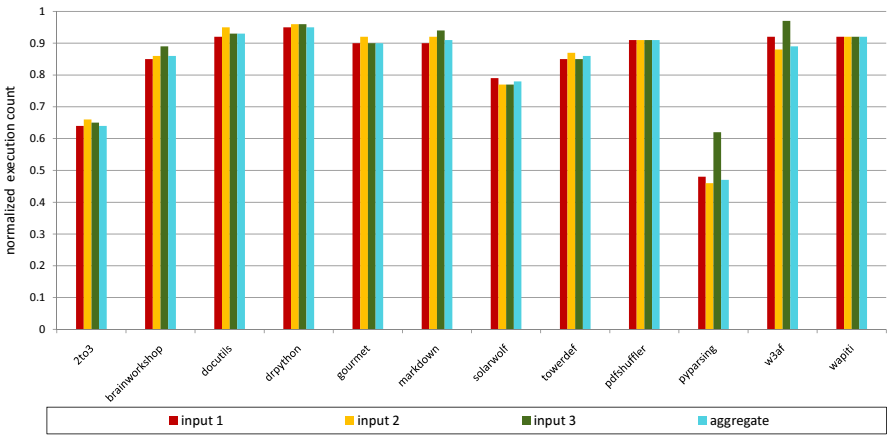


Figure 6.12: Effect of profile aggregation on the total execution count of monomorphic calls. Numbers are normalized to the workload total execution count. Higher is better.

We next consider behavioral similarity across inputs at the Python bytecode level. Figure 6.14 shows the ratio of bytecodes executed that are single-typed. With the exception of *gourmet* and *w3af*, the figure shows that the ratio of single-typed bytecodes tend to remain constant across inputs. This suggests that by specializing based on the aggregate profile, we can benefit different inputs.

So far we have assessed how representative an aggregate profile is of per-input profiles in terms of the amount of static behavior it identifies across different executions of a program. The data indicates that our aggregate profiles are good representatives

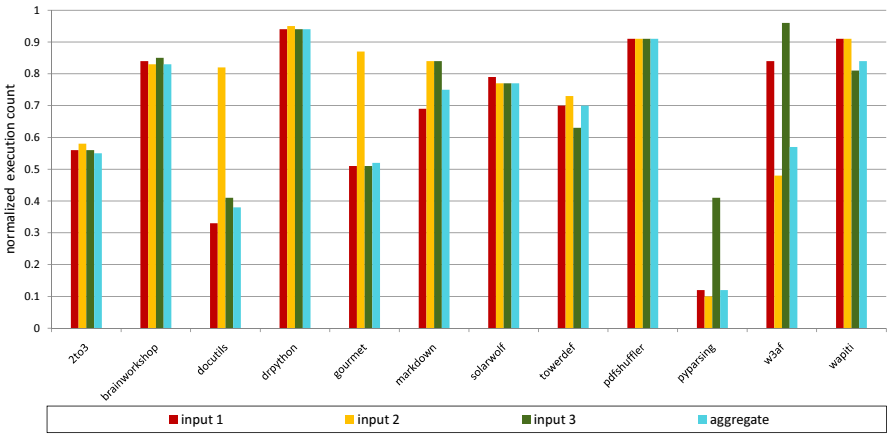


Figure 6.13: Effect of profile aggregation on the total execution count of monomorphic calls with fixed argument types. Numbers are normalized to the total execution count. Higher is better.

of individual inputs (i.e. they do not lose significant information). However, this result is only useful if an optimizing compiler specializes all methods in the code. Since, in our work, we attempt to balance performance gains with memory footprint, we only optimize hot methods. To do so, the hot methods must also be the same (or similar) across inputs.

Figure 6.15 quantifies hot method similarity of our programs. We measure the overlap of the methods that constitute the top 50% of approximated execution time between the aggregate and per-input profiles. In other words, if the compiler specializes the top 50% methods in the aggregate profile, we measure how much of the per-input profile gets covered by the specialization. For seven applications, the figure shows that the overlap is 50% or more for all inputs (up to 90% in *solarwolf*). For the remaining five programs, overlap ranges from 40% to 50%. The second input of *gourmet* exhibits

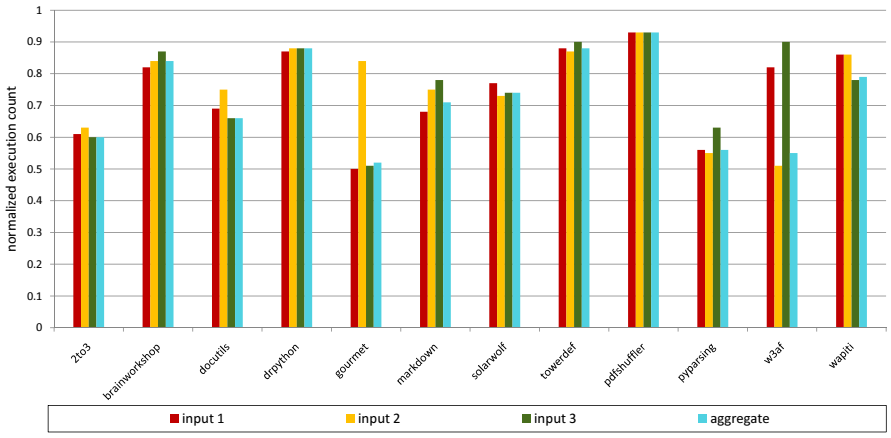


Figure 6.14: Effect of profile aggregation on the dynamic count of single-typed bytecodes normalized to the total bytecodes dynamic count. Higher is better.

a low overlap (20%). This is because this input exercises part of the application that the other two inputs do not.

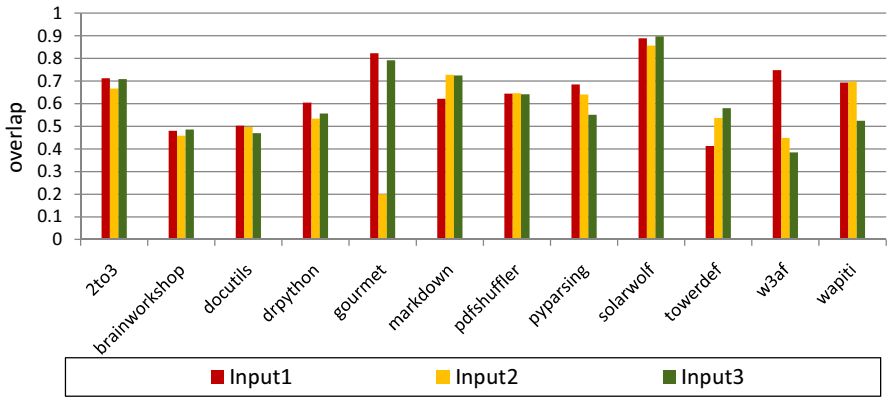


Figure 6.15: Overlap between the aggregate profile and each of the per-input profiles contributing to it. The graph shows the coverage over the per-input profiles when the top 50% execution count of the aggregate profile are optimized.

6.7.2 Sampling Overhead and Accuracy

We next evaluate the overhead and accuracy of the RCF CCT sampling system. We first consider overhead. Figure 6.16 shows the average runtime overhead for the command-line interface (CLI) applications with a randomized sampling rate between 5000 and 10000 events (calls or back-edges). On average, the overhead is less than 2%. Some applications show minor speedup. This is because the sampling overhead is so low that it falls within the margin of noise in measurements. *wapiti* has higher performance variation than the other programs. This is likely due to its network I/O as part of its implementation and not due to performance sampling.

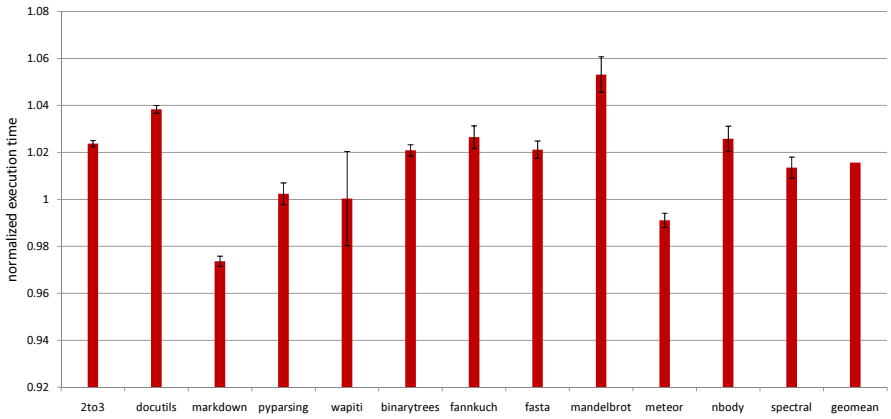


Figure 6.16: Runtime overhead of sampling calling context and bytecode types.

In Figure 6.17, we investigate how fast a sampled aggregate profile converges to the exhaustive aggregate one. That is, how many samples are needed before the sampled profile overlap with the exhaustive profile exceeds a threshold. We sample evenly from

The performance of the CPython main dispatch loop is very sensitive to modification. Hence, by adding extra code, one can obtain minor speedups like the ones shown here.

three different inputs to build the sampled aggregate profile. On every hundred samples, we record a snapshot of the sampled profile and compute the overlap. Our overlap metric measures how much of the exhaustive profile is covered by the top 50% call-sites of the snapshot. That is, if we choose to specialize the top call-sites accounting for 50% of the execution count in the sampled snapshot, how much of the exhaustive profile gets covered. Hence, a reasonable overlap threshold for convergence is 50%.

Surprisingly, by reading only 5000 samples, we achieve an overlap of at least 50% for all applications. This indicates that, for the programs we consider, if we perform online sample collection (i.e. the user's runtime communicates samples to the optimization server while the program is executing), very few samples are needed from a small number of users to build a sampled profile that is sufficiently representative to guide optimization.

6.7.3 Potential of Context-aware Specialization

We next investigate the feasibility and potential of context-aware specialization. We first measure the runtime overhead that is imposed by tracking the context hash (updating the context hash upon every method call). Figure 6.18 summarizes our findings. The average overhead is 0%, while the maximum is 4%. Similar to the sampling overhead, some applications get minor speedup because the overhead is so low that it falls within margins of noise in measurements. Next, we analyze the effect of context-

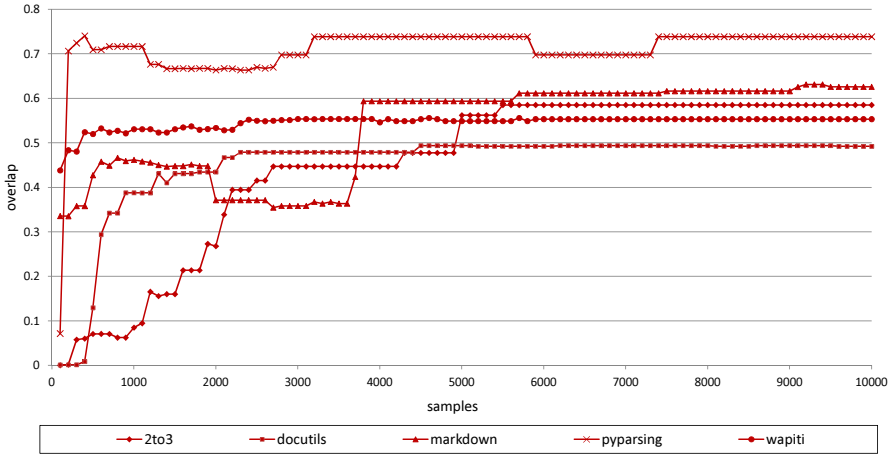


Figure 6.17: Convergence of the sampled profile. X-axis is the sample count and Y-axis is the overlap between the sampled and the exhaustive profile for the top 50% of the execution count. Higher is better.

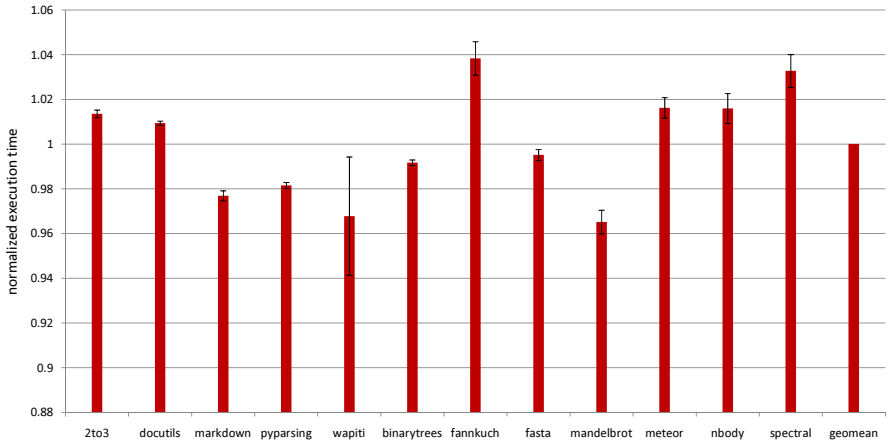


Figure 6.18: Runtime overhead of context hashing and tracking.

awareness on extracting static execution behavior from programs. As we did above in Figure 6.13 and Figure 6.14, we measure the amount of time spent in methods called from single-typed monomorphic call sites (Figure 6.19) and the amount of single-typed bytecodes (Figure 6.20) when we consider context information. The data shows that using context information increases the time spent in these specializable sections of code for all applications. The increase is large for dynamic applications such as *pyparsing*, *docutils* and *w3af*, indicating potential for additional performance improvement over not using context-awareness. Other programs exhibit smaller improvements (*drpython*, *pdfshuffler* and *brainworkshop*). Programs that are developed in a way that does not use dynamic features extensively benefit less from context awareness. Such programs are arguably easier to optimize and as such, our context-aware profiling technique can benefit more dynamic programs – those that are currently challenging to optimize effectively.

Finally, we investigate the added benefit in terms of execution count coverage that a program is likely to obtain through specialization of code for which calling contexts are distinguished. In this experiment, we order monomorphic single-typed call-sites by the time spent in target methods. We consider the methods in order of hotness (greatest to least) and estimate the code size (using bytecode size) that is required to specialize each of the clones. We plot the code size against the percentage of cumulative execution count that is spent in the specialized methods. We do this with and without calling

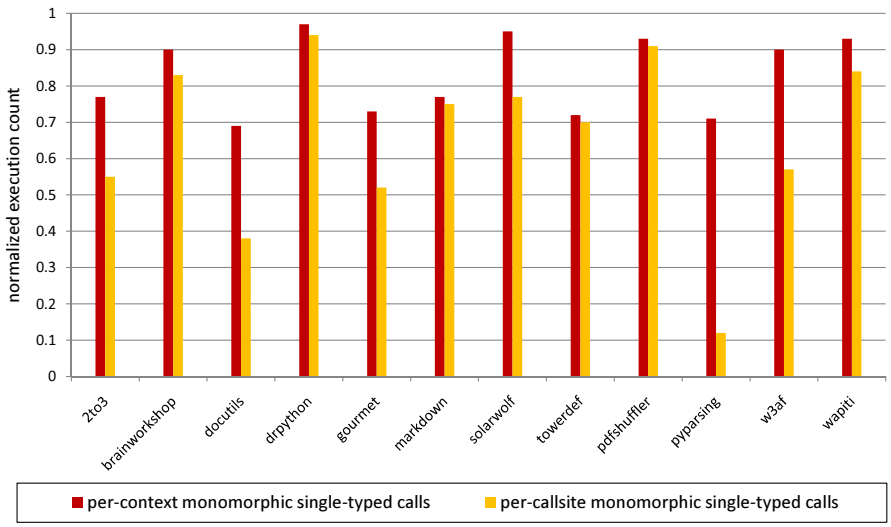


Figure 6.19: Effect of context-awareness on the execution count of monomorphic calls with fixed argument types. Numbers are normalized to the workload total execution count. Higher is better.

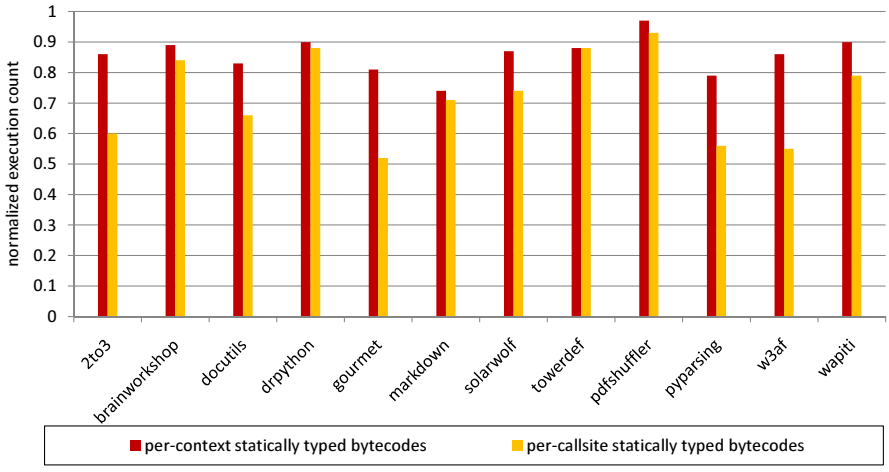


Figure 6.20: Effect of context-awareness on the dynamic count of single-typed bytecodes normalized to the total bytecodes dynamic count. Higher is better.

context information. Figure 6.21 presents these results for all 12 applications. The graphs show that half of the applications achieve significantly more execution count coverage for the same code size. This result indicates that such programs may be more aggressively specialized than was possible previously. As a proof of concept, we were able to speedup *pyparsing* by 3% by using calling context information to specialize its hottest and highly polymorphic call-site. This optimization would not have been possible otherwise due to the highly dynamic nature of the program.

6.7.4 RCF Specialization and Optimization

We carry out the optimizations by identifying top monomorphic call-sites that aggregate to around 50% of the execution count. The target methods are treated as hot code and are targets for our Python-level optimizations. Once we have the list of call-sites and target methods, we apply type-specialization in addition to a set of other optimization techniques:

1. **Inlining** - We perform inlining selectively on hot monomorphic call-sites. At the call-site, method resolution takes place and a guard will check if we have the correct method. If true, execution of the inlined code proceeds, otherwise, a generic call is made to the resolved method. Inlining eliminates Python calling overhead, part of which is arguments packing and unpacking, in addition to enabling across-methods optimizations.

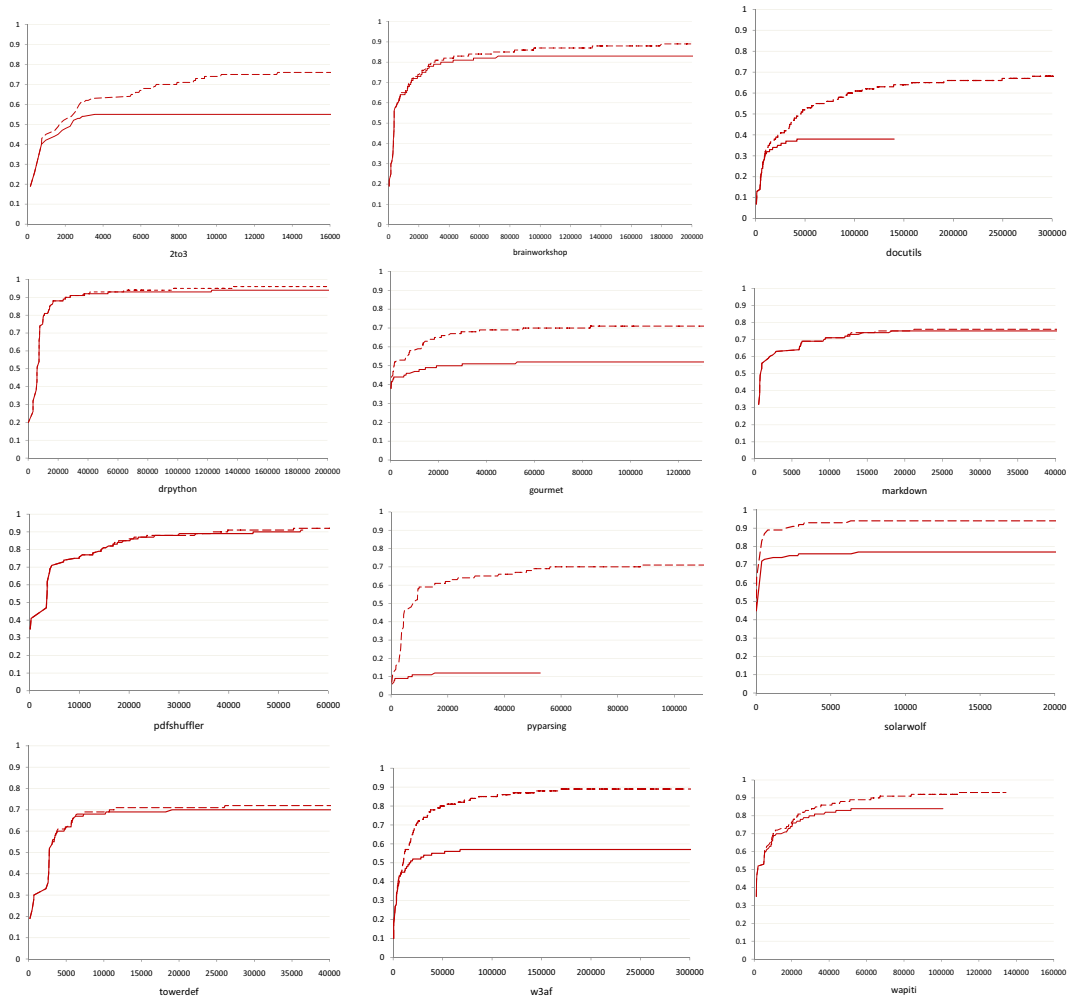


Figure 6.21: Comparison between per-call-site (solid) and context-aware (dashed) specialization for all of the Python applications. The X-axis is the size in bytes of the Python methods to specialize and the Y-axis is the percentage of execution count covered by specialized code.

2. **Loop Optimization** - We perform code hoisting, in which we move out loop-invariant code out of the loop body. For example, in CPython, global variables are dictionary-based and hence take more to access than local variables which are indexed by offset. Therefore, when safe, we hoist global variables access outside loop bodies and cache them into local variables. Furthermore, we transform Python loops to more efficient C-like loops. For example, instead of iterating over a *List* using high-level constructs such as `for item in items`, we lower it to be `for i in range(0, len(items))` and we access each item with the index number. This is done only for built-in types where iterators, among other attributes, are read-only.

3. **Type Specialization** - Guided by the aggregate profile, we type-annotate arguments and local variables of hot methods and compile them using Cython to a C extension module which we then compile using *gcc*. We modify call-sites to these methods to use the optimized method clone instead, adding guards as needed.

6.7.5 Speedup

We next present the speedup in performance that our RCF prototype achieves using these optimization. As mentioned in Section 6.6, we present results for a subset of our applications and microbenchmarks (chosen arbitrarily) for the remaining experiments.

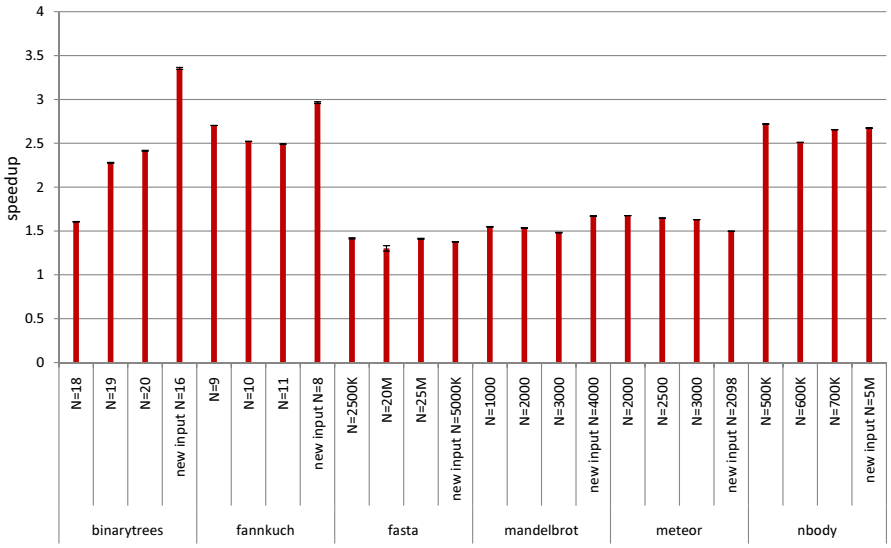


Figure 6.22: Speedup for six microbenchmarks using four different inputs. Three inputs are part of the aggregate profile and the fourth is a new input.

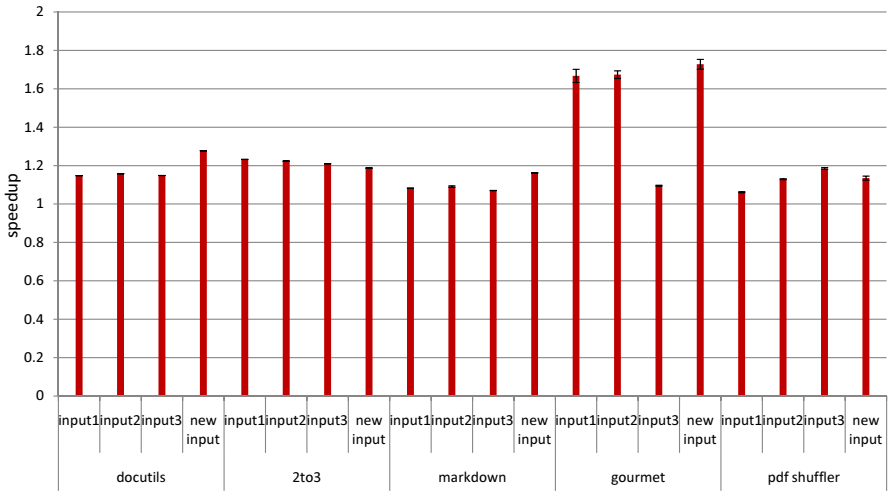


Figure 6.23: Speedup for real applications using four different inputs. Three inputs are part of the aggregate profile and the fourth is a new input.

Figure 6.22 and Figure 6.23 show the average speedup and error bars for six of the microbenchmarks and five applications, respectively. We show four different inputs for each program. The first three are used to generate the aggregate profile which is used to guide the optimizations. The fourth profile is a new input – not used as part of the aggregate profile. The speedup for the microbenchmarks ranges from 1.3× to 3.4×. The speedup for the applications ranges from 1.1× to 1.7×. The microbenchmarks benefit to a greater degree, as expected because they have a small, tight kernel which can be easily optimized with very few clones. The applications have flatter profiles. This speedup for applications is significant, however, in that it is on par or better than that reported by other Python optimization efforts that employ complex dynamic compilation such as Google's (now defunct) Unladen Swallow project [113, 114].

The other interesting result here is that the fourth (new) input shows nearly equal, and sometimes much higher, speedup for all programs despite not contributing to the aggregate profile and the optimization plan. This means that RCF can enable performance benefits for users that employ arbitrary and new inputs for some programs. It also shows that for these programs at least, a core subset of behavior can be captured by a small number of inputs.

6.7.6 PyPy Comparison

To compare these speedups against an on-going dynamic compilation project, we next compare our results to PyPy. PyPy compiles hot traces of the interpreter dispatch loop, and performs partial evaluation of methods with type and value specialization. We employ a different set of programs for this evaluation since PyPy was unable to execute most of our programs because it either does not support PyGTK or crashes during execution. We compare RCF to PyPy versions 1.5 (April 2011) and 1.6 (August 2011).

Steady-state Performance

We compare steady-state performance by averaging execution time over 10 runs excluding an initial warm-up run.

PyPy-1.5

Figure 6.24 shows steady-state comparison for the four inputs. Except for *fannkuch*, in most cases, RCF performance is similar to PyPy. PyPy outperforms RCF for *fannkuch* and RCF outperforms PyPy for *fasta* and *meteor*. PyPy has a chance to optimize all hot methods that repeatedly executed across the 10 runs. In addition, since most of the compilation and optimization happens in the first (warmup) run, this data omits most of the overhead of compilation that PyPy imposes.

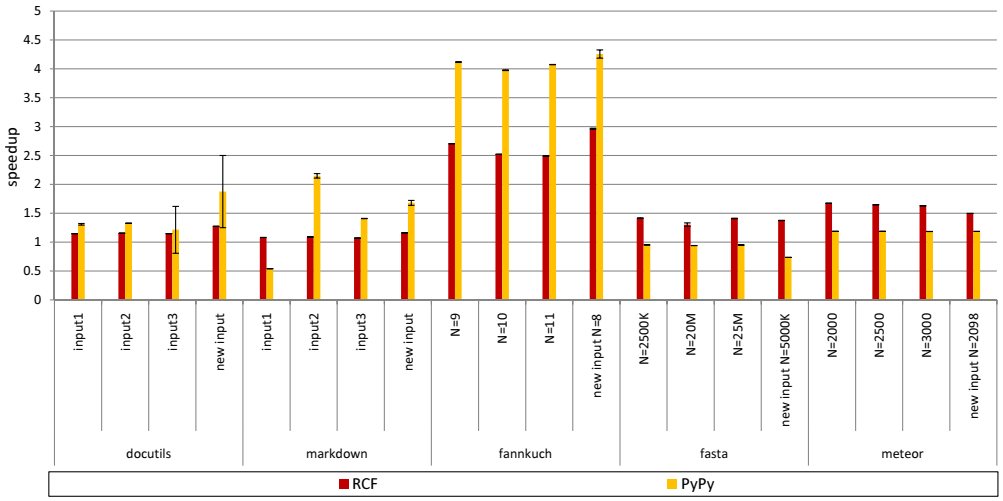


Figure 6.24: Speedup comparison of RCF to PyPy-1.5 without startup cost. We average the execution time of each workload over 10 runs after an initial warm-up run.

PyPy-1.6

Similarly, Figure 6.25 shows the steady-state performance comparison with RCF. RCF outperforms PyPy on all inputs for *meteor*, and the first input for *markdown*, for which PyPy shows a 50% slowdown. RCF also shows approaching speedup to PyPy for *fasta*. PyPy achieves better performance for all other programs. PyPy benefits stem from (1) our exclusion of PyPy compilation overhead due to steady-state comparison in these results, (2) the use of value specialization by PyPy (which we do not employ in our RCF prototype), (3) specializations that target the behavior of the current input (as opposed to RCF's conservative application of optimizations using aggregate, multi-input profiles), and (4) compilation of a greater number of methods than RCF.

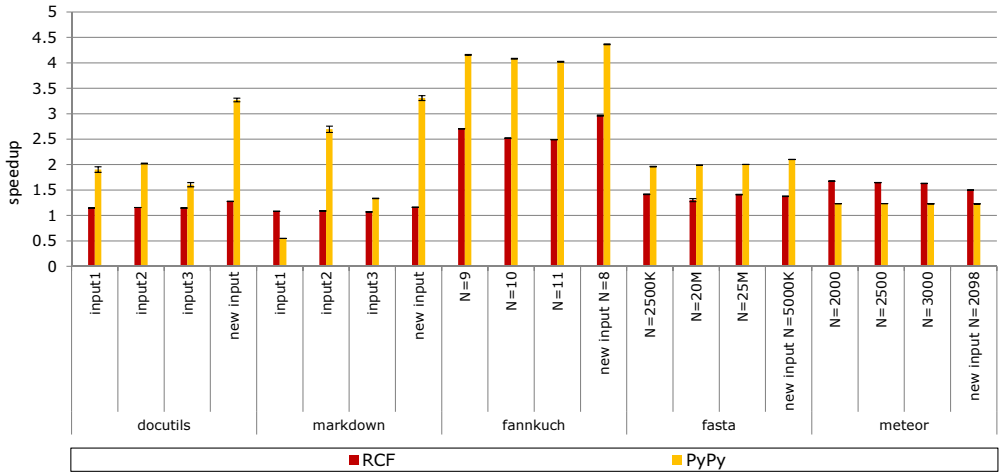


Figure 6.25: Speedup comparison of RCF to PyPy-1.6 without startup cost. We average the execution time of each workload over 10 runs after an initial warm-up run.

Cold Run Performance

We next consider the overhead of compilation and optimization that PyPy imposes. Such overhead impacts startup time, user interactivity, as well as overall performance. We consider only the warmup run, which we execute 10 times without the harness, and present the average and error.

PyPy-1.5

Figure 6.26 shows the result for PyPy-1.5. When we consider the overhead of dynamic compilation, RCF now outperforms PyPy for all programs except *fannkuch* for which the differences are now smaller. This result suggests that RCF is more suitable for client applications where startup cost and user interactivity is negatively impacted by

dynamic compilation systems that are unable to amortize their optimization overhead fast enough.

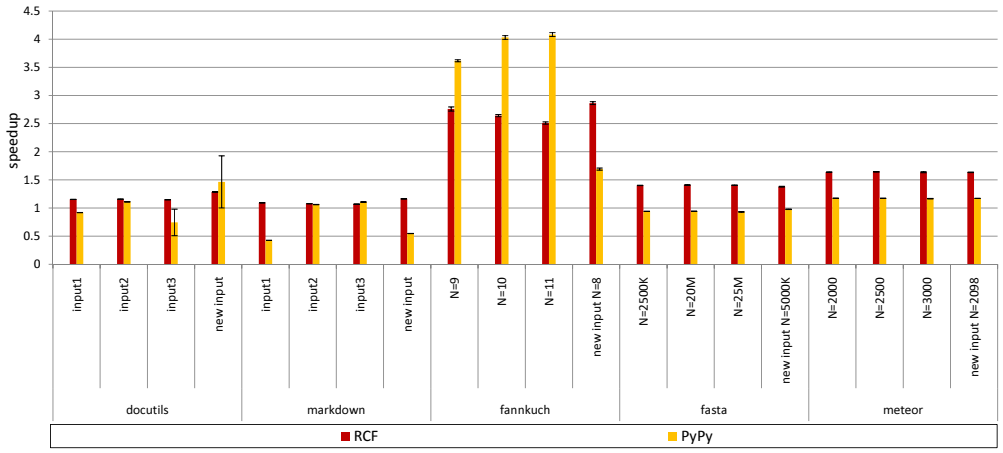


Figure 6.26: Speedup comparison of RCF to PyPy-1.5 for cold runs only. We average execution time over 10 cold runs to measure the effect of startup cost.

PyPy-1.6

In Figure 6.27, except *fannkuch*, for which the differences are now smaller, RCF and PyPy show similar speedup. In half of the cases, RCF outperforms PyPy. This result suggests that RCF may be more suitable for short running and user-interactive applications than dynamic compilation systems that are unable to quickly amortize their optimization overhead.

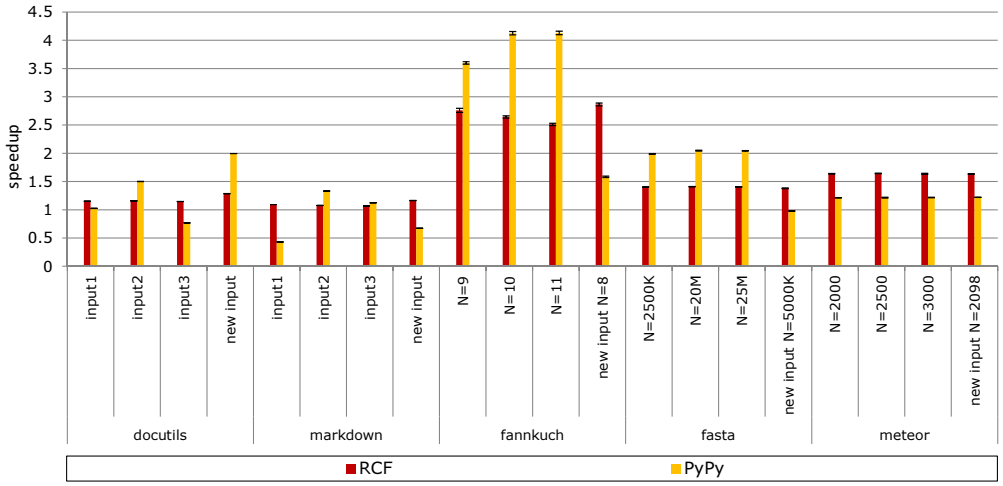


Figure 6.27: Speedup comparison of RCF to PyPy-1.6 for cold runs only. We average execution time over 10 cold runs to measure the effect of startup cost.

6.7.7 Memory Footprint

Since our goal with RCF is to achieve performance benefits without the memory footprint of dynamic compilation systems, we next investigate the impact of RCF on footprint versus that of PyPy.

Figures 6.28 and 6.29 show the memory footprint of CPython (solid), RCF (dashed), and PyPy (dotted), for PyPy-1.5 and 1.6, respectively. The X-axis is the percentage of execution time and the Y-axis is the memory in KiloBytes. We approximate memory footprint by measuring virtual memory resident set size via the Linux `ps` command. We query this value approximately every 0.1 seconds to generate this data.

RCF shows a similar memory usage pattern as CPython while PyPy footprint is significantly larger (and keeps increasing for all programs except *markdown*). The

footprint of the dynamic compiler is larger due to the code objects generated by PyPy as well as the code required to implement the compilation system. The memory usage of PyPy is 2×-7× greater than that of baseline and RCF for the programs we evaluated. This is also on par with that reported on the Unladen Swallow project website [115, 116].

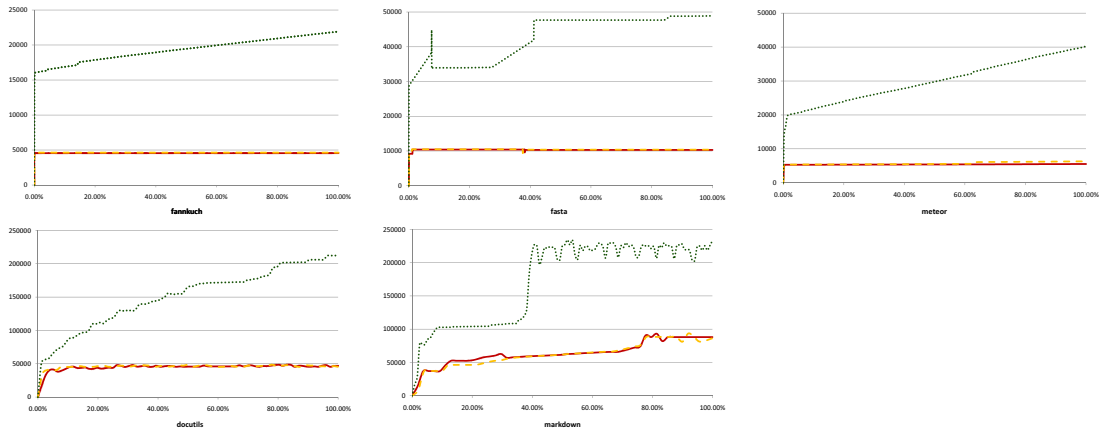


Figure 6.28: Memory footprint of CPython (solid), RCF (dashed), and PyPy-1.5 (dotted). The X-axis is the percentage of execution time and the Y-axis is the memory in KiloBytes

6.7.8 Phase Detection

In this section, we study how well RCF can react to larger Python programs that demonstrate phases in their behavior through its phase detection and recompilation technique. We first conduct a sensitivity analysis of the system to choose a suitable

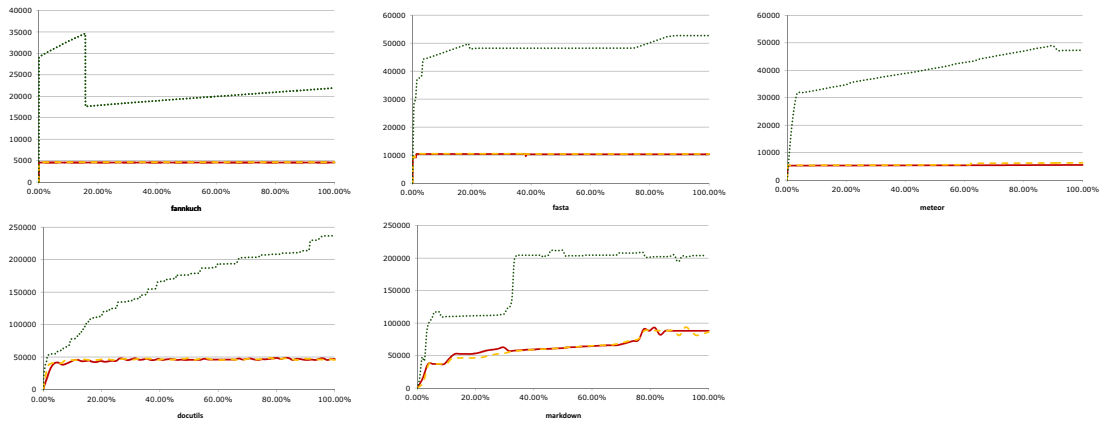


Figure 6.29: Memory footprint of CPython (solid), RCF (dashed), and PyPy-1.6 (dotted). The X-axis is the percentage of execution time and the Y-axis is the memory in KiloBytes

range for its parameters. We then evaluate the efficacy of the system, using the chosen parameters, on a multi-user experiment.

Sensitivity Analysis

RCF phase detection operation is controlled by a set of parameters that we presented earlier in Section 6.5. To understand the effect of those parameters on the system and the best range of values for each, we conduct a sensitivity analysis of the system. For each parameter, we vary it over a range of values while holding other parameters constant to a default value. Table 6.5 summarizes the system parameters and their default values.

For each experiment, we observe the effect on the following metrics:

Name	Description	Value
Convergence Threshold (CT)	Overlap threshold to exceed for a CC to converge	0.4
Convergence Cycle Duration (CCD)	Number of snapshots a CC must remain convergent before it ends	20
Phase Threshold (PT)	Overlap threshold to fall under for a phase to be detected	0.3
Phase Convergence Duration (PCD)	Number of CCs a phase must stay convergent before it is considered stable	4
Snapshot Distance (SD)	Distance between two snapshots compared during a CC	10
Snapshot Interval (SI)	Number of samples between snapshot recordings	100

Table 6.5: RCF phase detection parameters. For each sensitivity experiment, one parameter is varied while other parameters maintain their default values.

- **Average length of convergence cycle:** The average length of convergence cycles measured in samples.
- **Number of convergence cycles:** The number of convergence cycles generated.
- **Average recompilation delay:** The average delay, in samples, between the position where an actual phase shift occurs and where RCF triggers a recompilation.
- **Number of false positive/negative recompilations:** False positive are recompilations that cannot be associated with a phase shift. A false negative is a phase shift that goes undetected and no recompilation is triggered for it.

Evaluation of RCF phase detection demands a baseline for comparison. For that purpose we devise workloads for which we know where phase shifts occur. We call these *true phase shifts*. For each workload, we generate a trace of samples that contains a mark for every true phase shift that occurred during execution. We then feed the trace to RCF phase detection module. We finally compare the *detected phase shifts* and recompilation points with the true phase shifts marks in the trace. This enables us to find the number of false recompilations, recompilation delays and other metrics.

Our workloads consist of some of the applications described in Table 6.3 cascaded to execute back-to-back. Every application is homogeneous by itself and thus constitutes a stable phase. A transition from one application to the next causes significant changes to the aggregate profile that lead to a phase shift. We run each workload to generate a trace of samples where points of transitions are marked. The workloads are in two groups: the first group consists of the applications *2to3* and *docutils* exercised back-to-back with three different inputs. We iterate over the execution 5 times to generate 9 phase shifts (10 phases). The second group consists of *pyparsing*, *wapiti* and *markdown*. They are also executed with three different inputs and iterated 5 times to generate 14 phase shifts (15 phases). Table 6.6 summarizes the workloads, the number of samples for each, number of iterations and phase shifts. In the rest of this section, we investigate the effect of all phase detection parameters.

short name	# of samples	phases	iterations	phase shifts
exp1_inp1	3,297,142	2to3, docutils	5	9
exp1_inp2	7,081,851	2to3, docutils	5	9
exp1_inp3	2,077,072	2to3, docutils	5	9
exp2_inp1	3,701,229	pyparsing, wapiti, markdown	5	14
exp2_inp2	2,521,970	pyparsing, wapiti, markdown	5	14
exp2_inp3	3,989,511	pyparsing, wapiti, markdown	5	14

Table 6.6: Attributes of the six experiments used for sensitivity analysis. The table shows, for each experiment, the number of samples it generates, the phases it consists of, how many times the phases are iterated and the number of phase shifts.

Convergence Threshold (CT)

Every snapshot recorded during a CC is compared to a snapshot that is SD snapshots

in the past, where SD is the *Snapshot Distance*. Only when the overlap exceeds CT and remains above it for a certain number of samples is the CC terminated. Therefore, CT should control the CC length. The higher the CT, the longer it takes the overlap to exceed it. Figures 6.30 (a) and (b) show the effect on the CC count and length, respectively, as we vary CT from 0.05 to 0.49. For all experiment, there is slight effect on both metrics until CT reaches 0.4. After 0.4, there is a sudden increase in CC length (and decline in count). We inspected the overlap rate of increase during a CC and found that it rises rapidly until it exceeds 0.4. After that it increases at a much slower rate with many fluctuations. Therefore, setting CT to 0.4 has no significant effect on the CC length, while values above that makes it harder and longer for a CC to converge.

Figure 6.30 (c) show that CT does not cause any false compilation. Although one might expect that the increase in CC length can also reduce the sensitivity of the system, and hence lead to undetected short phases, this is not the case. The increase in CC length is never big enough to affect sensitivity. In other words, even with high CT, CCs always converge fast enough to be able to detect all phases.

Finally, we investigate the recompilation delay in Figure 6.30 (d). As expected, the increase in CT causes an increase in recompilation delay as the CCs grow larger. Again, the increase is significant for CT above 0.4 with the exception of *exp2_input2* where CT equals to 0.1. At this point there is a spike in delay from 17,000 to 20,000 samples. This increase is due to how, by affecting the CC length, the CT also affects

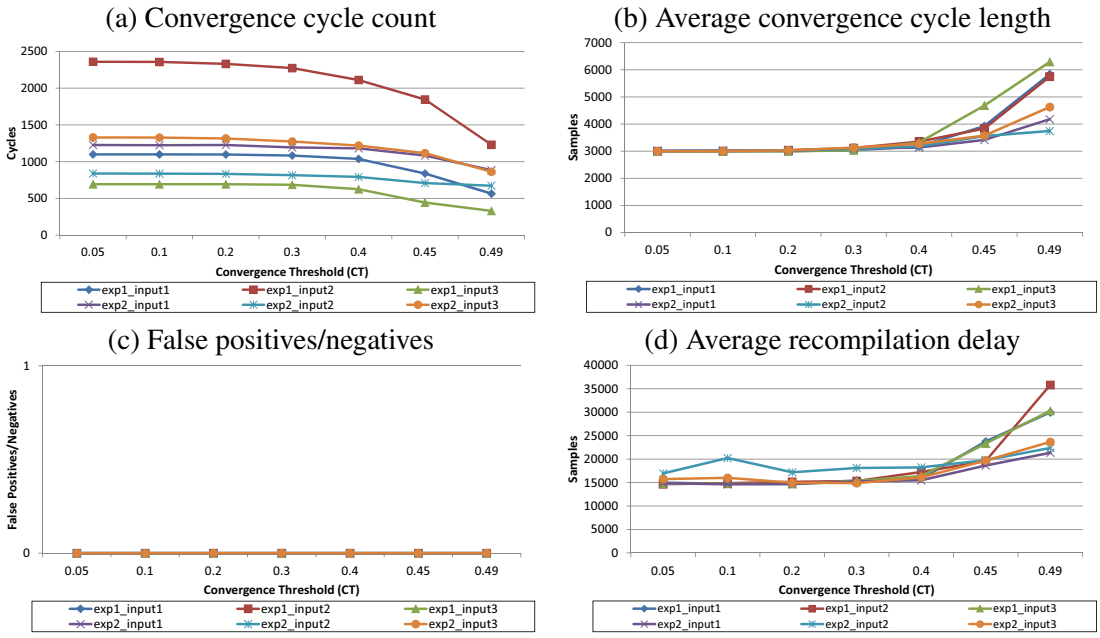


Figure 6.30: Effect of convergence threshold (*CT*) on number of convergence cycles, average convergence cycle length, false positives/negatives and average recompilation delay.

CC termination points and how they align with phase shifts. If a phase shift occurs very close to a CC end, the last snapshot (LS) of that CC will not reflect the shift, and we have to wait for another CC before the shift is detected. Given that the CC length for this data point is around 3000 samples, the increase is reasonable. This is an artifact of the randomness of the sampling process.

Convergence Cycle Duration (CCD)

CCD dictates how long the CC overlap must remain above CT before a CC terminates. It has a direct impact on the CC length. Figure 6.31 (a) and (b) show that impact when the CCD is varied from 5 to 200 snapshots. A snapshot interval is fixed at 100 samples.

By elongating CCs, CCD also indirectly increases the phase convergence duration (PCD) which is the number of CCs RCF must wait before it declares a phase stable. Longer PCD makes the system less sensitive to phase shifts. In other words, a PCD can outgrow a phase length which will make the system consider a whole phase as a temporary transient behavior before an actual stable phase is reached. On the other end, a short CCD causes premature termination of CCs which leads to randomness in the last snapshot (LS) of CCs. Figure 6.31 (c) summarizes this effect. At CCD below 10, there is an increase in false positives (false phase detections) due to randomness in CCs. Starting at 80, the number of false negatives (undetected phases) grows significantly. Finally, we study the effect on the recompilation delay. Figure 6.31 (d) shows the effect. At CCD equals to 60, the delay starts growing significantly. Although, for this

experiment, a reasonable range for CCD is between 10 and 60, our experiment with a larger number of users (discussed later in Section 6.7.8) reveals that CCD must scale with the number of users to achieve accurate phase detection.

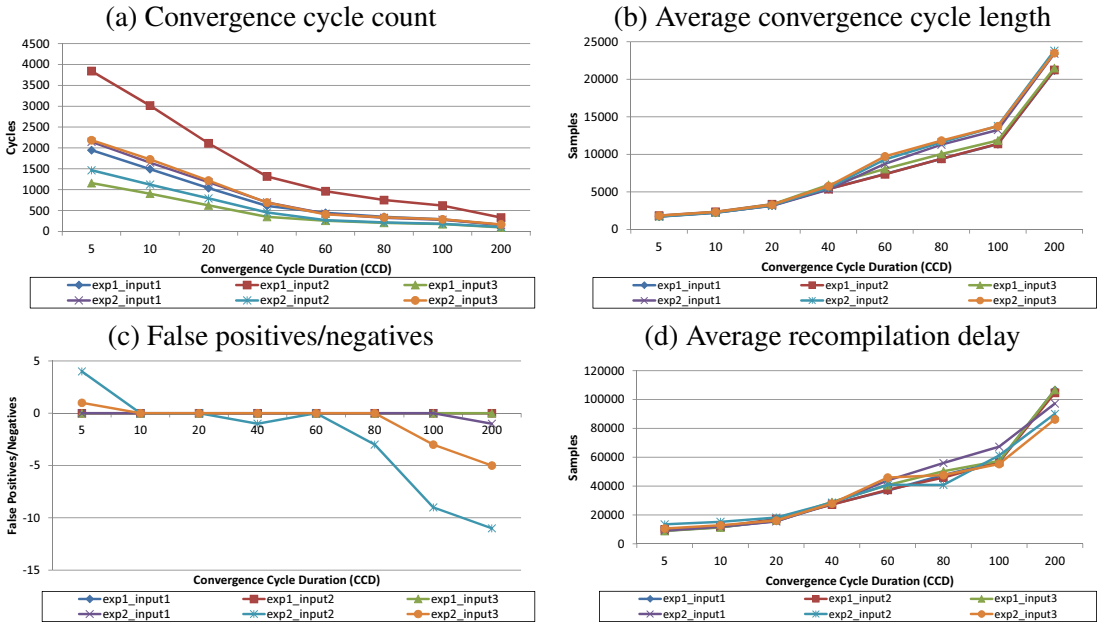


Figure 6.31: Effect of convergence cycle duration (*CCD*) (in snapshots) on number of convergence cycles, convergence cycle length, false positives/negatives and average recompilation delay. Infinite average recompilation delays are due to lack of phase detections and recompilations.

CCD is an important parameter as it can control the system's sensitivity to phase changes. By increasing it, the system can be tuned to be less sensitive to shorter phases. By lowering it, the system can detect subtle changes in behavior. CCD is also important as it controls the amount of randomness among consecutive CCs. A longer CCD makes each CC more stable and representative. In Section 6.7.8, we will show that with more

users contributing samples to the system, more randomness is present. Increasing CCD dampens this randomness.

Phase Threshold (PT)

Phase threshold (PT) is the overlap threshold that the last snapshot (LS) of the current CC and the LS used in latest optimization must fall under for a phase shift to be detected. This parameter affects operation at the CC level, not snapshot level, and is therefore orthogonal to both CC length and count. However, PT has a direct effect on false recompilations and recompilation delay. With low PT, the system becomes less sensitive to phase shifts which can lead to false negatives. With high PT, two contradicting effects take place. On one hand, the system is more sensitive which can cause false positives. On the other hand, increased sensitivity causes more interruptions during the PCD period and thus elongates it (refer to Figure 6.11). Elongated PCD can cause relatively shorter phase shifts to be overlooked as transient fluctuation hence causing false negatives. Figure 6.32 (a) summarizes these effects. We start seeing false negatives at

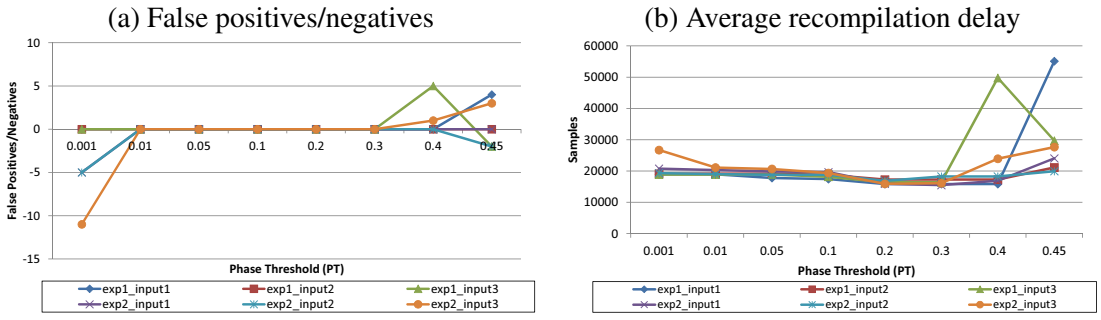


Figure 6.32: Effect of phase threshold (*PT*) on false positives/negatives and average recompilation delay.

very low PT values (0.001). No false recompilations happen in the range from 0.01 to 0.3. Starting from 0.4, both positive and negative false recompilations arise. Figure 6.32 (b) shows the effect on recompilation delay. The delay remains fairly constant until PT equals to 0.4. After that, there is increase in delay due to the longer PCD. Reasonable PT value should be in the 0.01 to 0.3 range, depending on how sensitive we want the system to be.

Phase Convergence Duration (PCD)

PCD parameter tells RCF how many CCs the profile needs to be stable before a stable phase is declared. It has the intuitive effect of increasing the PCD period and decreasing the system sensitivity to relatively shorter phases. It also causes the recompilation delay to be longer, since recompilations happen only after stable new phases are declared. Figure 6.33 (a) and (b) show the effects. At PCD equals to 8, we start seeing false negatives. Nearly all experiments have false negatives at PCD equals to 128. There is also a consistent increase in recompilation delay for higher PCDs. A reasonable range for our experiment is 4 CCs.

Snapshot Distance (SD)

The distance between every two snapshots compared during a CC is the snapshot distance (SD). A low SD can cause a CC to terminate early leading to randomness in CC endings which can possibly lead to false positives. A high SD will expand the CC length, and consequently the PCD, and lead to false negatives. Figure 6.34 (a) and (b)

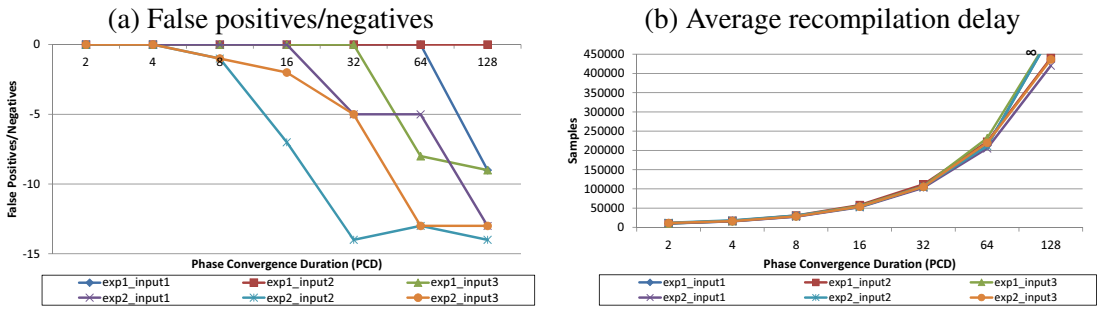


Figure 6.33: Effect of phase convergence duration (*PCD*) in CCs on false positives/negatives and average recompilation delay.

show constant increase in CC length and decrease in count. Figure 6.34 (c) shows false negatives at values larger than or equal 60. At 100, RCF does not detect any phases for *exp2_input2* due to the long PCD. At 1000, all experiments, except for *exp1_input2*, have no phases detected. A good range for this parameter is 10 to 40 which avoids false recompilations and retains low recompilation delay (Figure 6.34(d)).

Snapshot Interval (SI)

Snapshot interval (SI) determines the frequency by which snapshots are recorded. Although snapshots collected at higher rate (low SI) provide finer granularity for profile comparison, they need to be accompanied by an increase in the CCD. Otherwise, the snapshot comparisons will yield a false indication of CC convergence leading to randomness in LSs and false positives. On the other extreme, higher SI will cause longer CCs and consequently longer PCD and false negatives. Figure 6.35 (a) and (b) show the effect on convergence length and count, respectively. Figure 6.35 (c) shows how SI can generate false compilations. On the high end, we see false negatives, which is

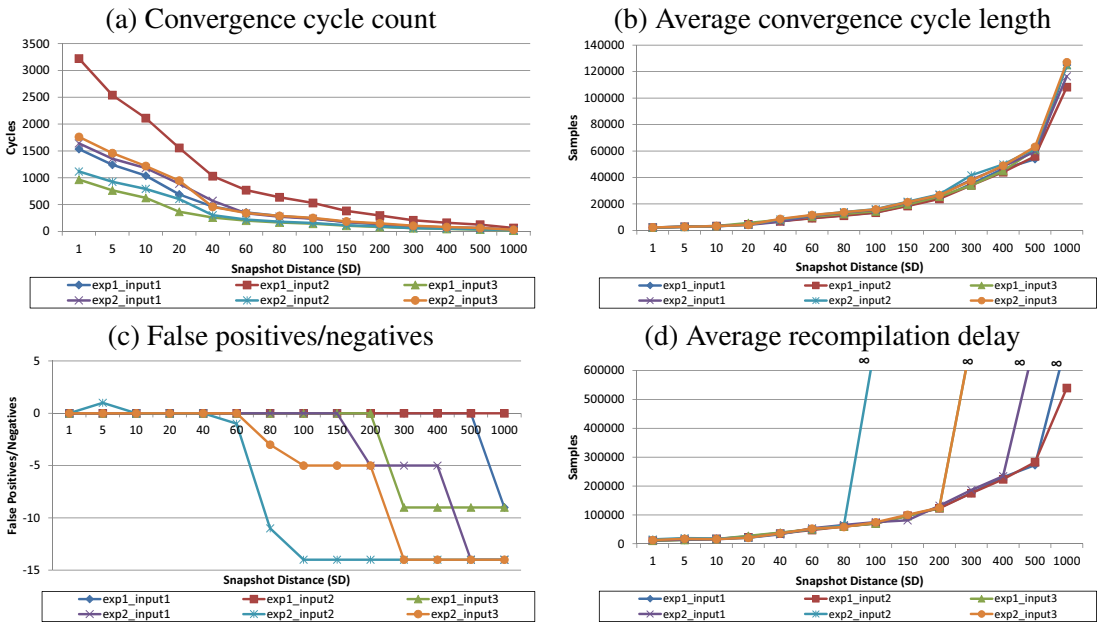


Figure 6.34: Effect of snapshot distance (*SD*) on number of convergence cycles, convergence cycle length, false positives/negatives and average recompilation delay.

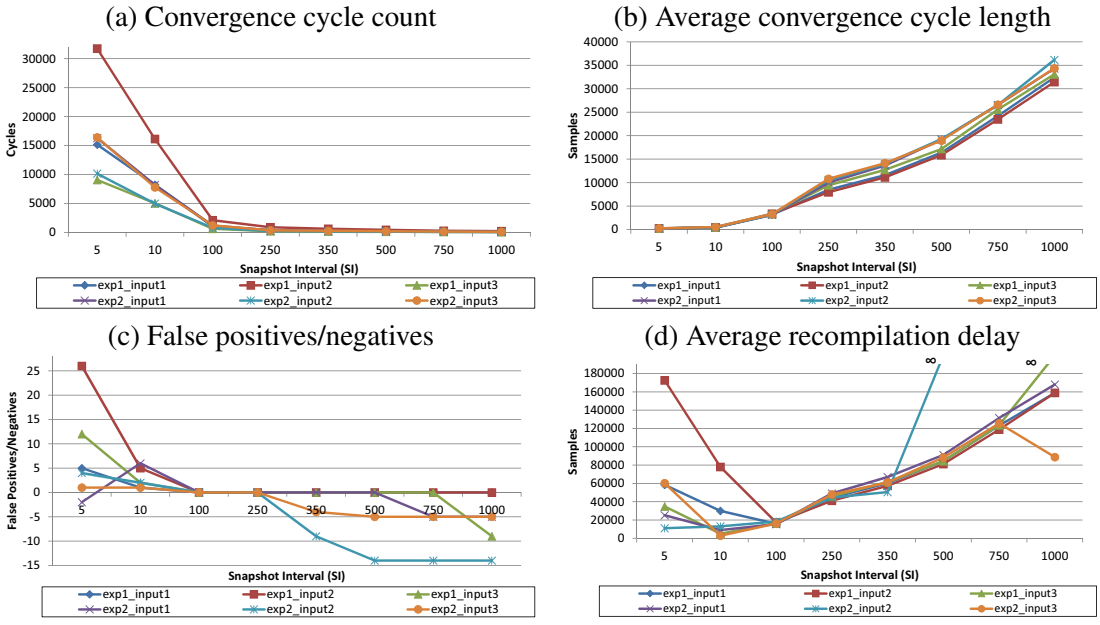


Figure 6.35: Effect of snapshot interval (*SI*)on number of convergence cycles, convergence cycle length, false positives/negatives and average recompilation delay.

explained by the undetected phase shifts due to a long PCD. On other low end, there is a mix of false positives and negatives. This is induced by the randomness in CCs which can either cause the overlap to drop below PT, hence leading to false phase detection (false positives), or overlap drops during PCD which can lead to excessively long PCD and hence undetected phases (false negatives). This randomness also causes an increase in recompilation delay on both ends in Figure 6.35 (d). A reasonable choice of SI range is from 100 to 250. The minimum recompilation delay for all experiments with no false recompilation happens at SI equals to 100.

Multi-user experiments

In this section, we evaluate the efficacy of RCF phase detection for large multi-user experiments. We set the system parameters to the values shown in Table 6.5. For CCD, however, we increase its value with the number of users to overcome the randomness in CCs with the increased number of users. We will explore the effect of CCD on accuracy later in this section.

To be able to assess RCF accuracy, we devised a workload consisting of five applications (components): *markdown*, *wapiti*, *pyparsing*, *docutils* and *2to3* executed in this order back-to-back. Each transition from one application to the next is a phase shift. We divide the users into three equal groups, each executing with a specific input. For every

group of users, we collect a trace of samples that is fed into RCF. We experimented with 100, 500 and 1000 users.

Figure 6.36 illustrates the behavior of the workload and the RCF response to it. For every input (bottom three bars), we show the phase shift points at which the workload moves from one component to the next to form five phases for each. The phases are plotted against the snapshot count. Since the number of users in each group (input) is equal, a phase shift in one group always has an impact on the overall phase of the workload. The top bar shows the RCF reaction to every phase shift. There are 13 workload phase shifts; RCF detects all of them accurately. No false positives or negatives exist for all users.

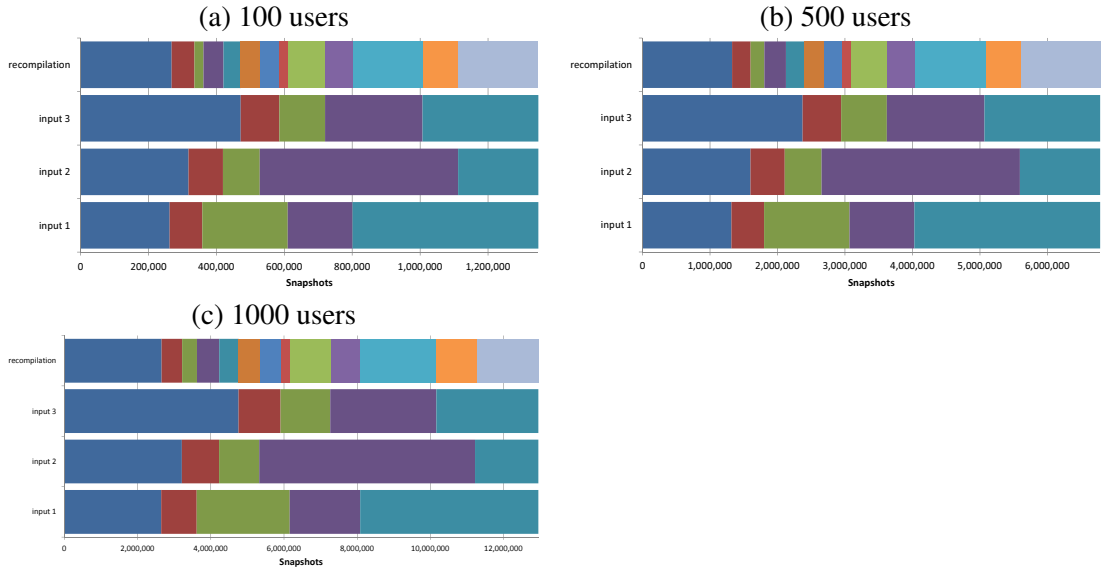


Figure 6.36: Illustration of phase shifts and recompilation points plotted against snapshots count for 100, 500 and 1000 users experiments.

# of users	100	500	1000
Samples	134,826,000	678,215,300	1,294,850,600
Avg CC length (samples)	14,799	61,556	259,033
CC count	9,110	11,017	10,402
Avg phase length (samples)	9,050,700	45,528,000	85,858,100
Avg recompilation delay (%)	3.88%	3.68%	2.08%

Table 6.7: Difference in behavior for the multi-user experiment when varying the number of users.

Table 6.7 shows the effect of increasing the number of users on the workload. The number of samples nearly scales linearly with the number of users to reach almost 1.3 billion samples at 1000 users. The average CC length increases with the number of users, this is partially because we scale CCD to be equal to the number of users and also because with more users comes more variation in the profile, thus it takes longer for a CC to stabilize. The table also shows average phase length, which is the average number of samples between consecutive phase shifts. The phase length scales linearly with the number of users. Finally, the average recompilation delay is shown, which is the distance between when a phase shift occurs and when RCF reacts with recompilation. The number shown is a percentage of the new phase length. For all users, on average, recompilation happens before 4% of the phase samples are read. The delay declines with increasing number of users and longer phases. This indicates that although the phases are longer, only a small percentage is needed to recognize the shift. Figure 6.37 and Figure 6.38 show a breakdown of phase length and recompilation delay, respectively.

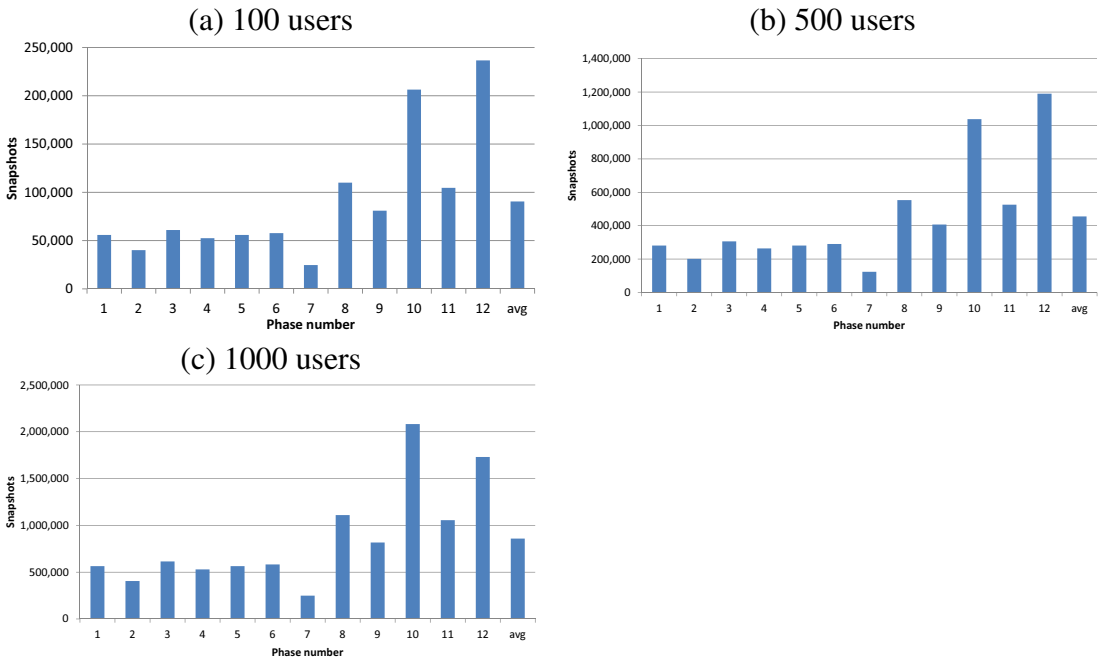


Figure 6.37: Phases length for multi-user experiment for 100, 500 and 1000 users.

Effect of Convergence Cycle Duration

We mentioned earlier that we need to scale the CCD to be equal to the number of users. Because the CCD defines the length of a CC, if we keep the CCD constant then the number of samples received during each CC will be roughly the same. Now, if we increase the number of users connected to RCF while CCD is fixed, then, during one CC, we will receive less samples from each user on average. In other words, we will learn less about what each user is doing since the number of users is increasing and the amount of samples collected per CC is the same. Since users are using the program differently, it is imperative that we know enough about every use case. Hence, it is important to allow CCD to grow with the number of users to allow RCF to learn

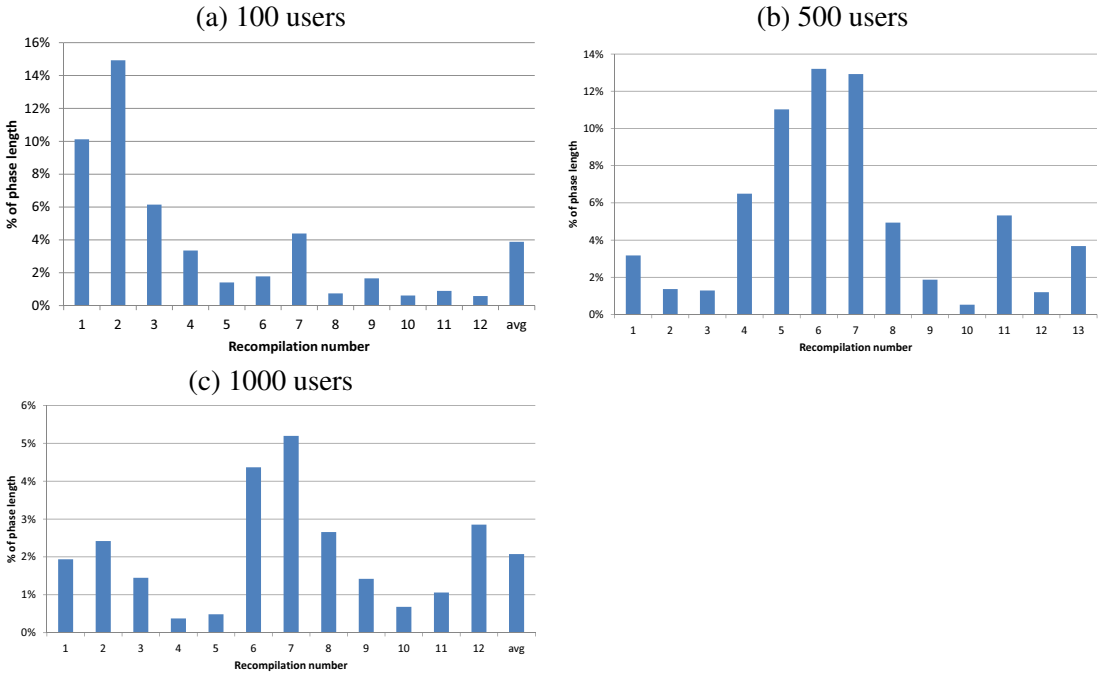


Figure 6.38: Recompilation delay normalized to new phase length.

enough information about every usage model. If the CCD remains fixed, then the last snapshot (LS) of every CC may not be sufficiently representative of that CC leading to fluctuations in LSs and false phases detections.

While increasing CCD increases the CC length in samples, it does not necessarily mean that the CCs will have longer time duration. With more users connected to the system, samples may be received at a higher rate, and since CCD is measured in samples, increasing it may not affect the CC time duration.

CCD is the only parameter that depends on the number of users aiming to reduce the fluctuations among CCs and hence false positives. Table 6.8 demonstrates the effect

of lowering the CCD for the 500 users experiment. There is an expected decrease in average CC length and increase in their count. When the CCD is lowered to a 100 instead of 500, many false recompilations occur.

CCD	100	500
Avg CC length (samples)	15114	61556
CC count	44872	11017
False positives	588	0

Table 6.8: Effect of decreasing CCD on number of false positives, average CC length and CC count

6.8 Related Work

The idea of distributed remote profiling has been employed before to serve different purposes. Liblit et al. [67] propose an approach to isolate bugs in a program by remotely collecting assertion outcomes from a large user community. Orso et al. [77] propose *GAMMA system*, a low-overhead software monitoring framework where program profiles are constructed by merging partial profiles gathered from many users connected through a network . Nagpurkar et al. [74, 73] present a phase-aware [100] profiling framework aimed at collecting accurate per-user profile at low overhead. These approaches are different from ours in that we are interested in gathering a global *performance* profile, not a per-user profile, that spans multiple users and runs of the program. This enables us to use extremely low sampling rates without sacrificing accuracy.

Tian et al. [112] use an approach similar to ours of cross-user profiling to gather program input information and loop trip counts with low overhead to enable input-aware optimizations. However, unlike RCF, the profiles analysis and optimizations happen locally on the user machine. The optimizations remain dynamic (happen at runtime), per-user and rely on a relatively shorter profiling period to identify the input features. Also, their work targets Java, which is a statically-typed language. In contrast, RCF is geared more towards DSL, performs AOT compilation and relies on cross-user aggregate profiles to devise optimizations for all users.

A remote compilation strategy is used by Lee et al. [66] for Java to offload compilation tasks to a remote server resulting in lower compilation overhead and footprint. Similar work by Palm et al. [78] propose a remote compilation service for hand-held devices in order to achieve a better battery consumption. Sirer et al. [105] factor runtime services such as verification, security enforcement, compilation and optimization to a cluster of machines serving as a distributed virtual machine (DVM) for Java. Although our work bears similarity in concept, we do not do code replacement while the client application is running. Instead, we compile offline and send program updates back to the user. Furthermore, to the best of our knowledge, no remote compilation framework exists for a dynamic scripting language.

Biggar et al. [19] present an ahead of time compiler for PHP called *phc* which generates code based mainly on the interpreter C APIs. That improves the compiler

portability across different releases of the language. Our compilation approach is no different, it relies on Cython which generates a sequence of C calls to the CPython APIs. Cython, however, can type-specialize either by using more type-specific calls or by generating its own specialized C code. Mauricio et al. [96] present quicksilver, a part ahead-of-time and part dynamic compiler for Java. They provide a mechanism in which Java class files were pre-compiled to binary and were stitched to the JVM instance. Our approach relies on profiling information collected from a distributed user environment whereas their approach relies on actual execution of the program and generation of quasi-static images.

Zhuang et al. [127] propose adaptive calling context profiling for Java. They describe an adaptive bursting technique where redundant profiling is avoided by using an adaptive algorithm. Serrano et al. [97] present a method of building approximate calling context from partial call traces. The partial call traces are merged together to generate the smaller partial context trees which were later merged to generate the approximate calling context tree. Arnold et al. [12] propose a novel technique of generating approximate CCT via sampling for Java. We use a similar approach to merge samples into the aggregate CCT.

There has been several prior work on understanding overall program behavior and phases of execution [118, 72, 102, 36, 101, 100]. In [102], Sherwood et al. present a hardware-based phase detection and prediction scheme. Their technique relies on

generating basic blocks execution frequency profiles at equal profiling intervals and classifying these intervals into unique phases. They use an encoding of phases history and duration to predict the next phase. In [118], Vijayn et al. reduce the hardware storage needed by identifying phases using branches footprint (sequence of branches addresses). Their technique also reduces the number of unique phases by collapsing phases with a small variation in footprint. Nagpurkar et al. [72] devise an online software-based phase detector for virtual machines. Their solution is based upon grouping profile elements into windows and doing a similarity check to detect a phase shift.

Several applications of phase detection and prediction have been presented in prior work. Barnes et al. [14] employ region-based compilation to generate phase-specific code. Huang et al. [57] use phases to perform power optimization reconfiguration. Shen et al. [99, 98] use re-use distance to perform offline signal processing to detect phases in traces of memory accesses. They use binary rewriting to insert phase markers into the code to perform phase prediction. Also, they employ phases to guide optimizations such as dynamic data packing, memory remapping, and cache re-configuration. Dhodapkar et al. [35] detect phases to drive reconfigurable hardware through a virtual machine monitor (VMM).

RCF phase detection scheme differs in several ways. First, RCF operates on a higher level profile of hot methods and their argument types to guide a type-specializing

compiler. Second, the profile samples are collected online from different instances of the application running with different inputs. Finally, although RCF still relies on the idea of a profiling interval (a profile snapshot in our case), we add a second layer of adaptive profiling interval: the convergence cycle. A convergence cycle is terminated only when the aggregate profile is stable long enough. This dampens the expected randomness in snapshots when sampling from multiple sources.

6.9 Conclusion

In this chapter, we present the Remote Compilation Framework (RCF) for improving the performance of Python programs. RCF decouples compilation from interpretation so that each can be performed at a different location. The RCF model is one in which a large number of users execute a particular program locally using an interpreter. A subset of these users do so using an extant interpreter system extended with support for collecting samples of program execution behavior unobtrusively. These runtimes communicate these samples to an optimization server. The server aggregates the samples into a calling-context-aware type profile that is used to guide translation of hot functions to C and to type-specialize, partially evaluate, and inline the code. In our current prototype, we use a combination of Cython translation and type specialization, by-hand optimization, and gcc compilation to perform this function. RCF returns the

optimized program to all users as a part of a software update; the optimized program code can be executed using any extant (unmodified) runtime.

RCF targets a sweet spot between interpretation and dynamic compilation. RCF retains the benefits of both: simplicity and ubiquity of user interpreter-based runtimes, and high-performance of dynamic-compiler-based runtimes. We evaluate RCF using community benchmarks and real applications. We find that RCF enables $1.1 \times -3.4\times$ performance gains, introduces very low overhead for performance sampling ($< 2\%$), and has a memory footprint that is $2.7 \times -7\times$ smaller than a compiler-based runtime. We compare RCF to PyPy, a popular dynamically optimizing and type/value specializing runtime. We find that the RCF achieves similar or better performance – especially when we consider PyPy compilation overhead.

We also propose a phase detection mechanism for RCF that enables it to optimize larger Python programs (e.g. server-side applications). RCF continuously compares snapshots of the aggregate profile to detect phase shifts and triggers recompilation if the program enters a new phase. We evaluate our techniques through an elaborated sensitivity analysis of the systems parameters and by trying our system on a large multi-user experiment. RCF is able to accurately detect all phase shifts and recompiles with a delay of 4% on average.

Chapter 7

Conclusion and Future Work

Providing software for today's computing platforms has become challenging. These platforms are heterogeneous in architecture, capabilities and resources ranging from low-end resource-constrained devices up to high-end desktops and servers. Software built for these devices must meet their particular design requirements and make the best use of their capabilities and features. Software, on the other hand, is growing in complexity and is usually divided into many components written in different languages. Efficient programming for these devices requires a diverse set of experiences and skills in both platforms and languages. Certain remedies have arisen that facilitate software development, deployment and maintenance. First, collaboration between a large, distributed and remote group of developers is made possible via Revision Control (RC) systems that allow incremental and concurrent contributions to the source code base. Second, software deployment and maintenance is enhanced by using software repositories to share, distribute, and automatically update applications and sys-

tems. Third, developer productivity and software portability are improved with the use of high-level managed languages. Despite their clear benefit, these remedies cause understanding software behavior and optimizing it to be challenging. It becomes difficult to reason about how changes made through an RC system to various parts of a code base shape overall performance. Additionally, software repositories allow programs to be downloaded and used by millions of users over different platforms in different ways which hinders reasoning about different usage models and devise optimizations that will benefit most users. Finally, managed languages runtimes abstract away the hardware, pose startup overhead and are hard to build especially for dynamic scripting languages. These disadvantages and others make it hard to understand the behavior of software, both during development and after deployment in the wild, and to extract performance dynamically and automatically.

7.1 Contribution

This thesis work contributes to addressing the above disadvantages by enabling better understanding and performance of managed languages using novel profile analysis and collection techniques. In particular, we present novel ways to represent and contrast performance of managed languages across software revisions. We characterize and optimize the performance of interpreter-based dynamic scripting language (DSL)

programs. We also contribute a new offline feedback-based approach of compiling and optimizing DSLs that is an intermediary point between simple interpretation and advanced dynamic compilation.

Revision control (RC) systems enable large-scale collaborative software development by tracking source code changes from different developers. This permits building large, complex and often multi-language applications. Since each developer modifies the code in isolation, it becomes difficult to reason how several incremental changes affect overall performance. Such understanding is essential for performance debugging and tuning. To tackle this problem, we present a Performance-Aware RC System (PARCS) for Java programs. Our PARCS prototype automatically contrasts performance profiles between revisions and exploits existing RC system services to attribute performance changes to source code modifications. PARCS models execution as Calling Context Trees (CCTs) that it automatically generates and compares in topology and annotations. Our evaluation of PARCS revealed that our incremental CCT differencing algorithm using source code differences provides better CCT differencing than existing solutions. We show via a case-study the applicability and effectiveness of PARCS in attributing performance differences to code modifications.

Lately, software developers have been increasingly relying on managed Dynamic Scripting Languages (DSLs). These languages have been gaining popularity because of their high-level concise syntax and dynamic features that boost developers' pro-

ductivity. DSLs, however, are challenging to implement efficiently. Due to their dynamic features, constantly changing semantics and the complexity of compiler-based runtimes, most DSLs standard implementations remain purely interpreted. Interpreter-based implementations are also favored for their low startup overhead, low footprint and accelerated development cycle.

In this work, we aim to contribute new techniques to improve DSLs performance while preserving the simplicity and flexibility of their runtimes. Our contributions start with characterization of Python performance as a representation of DSLs. Our findings show that existing static language interpreter optimizations, namely interpreter dispatch optimizations, are not suitable for DSLs. We identify sources of overhead for Python and target them with a set of novel interpreter-based optimizations: attributes caching, load/store elimination and inlining of runtime calls. We are able to improve performance by up to 28% and 15% on average.

Despite optimizations, interpreter-based managed runtime environments (MREs) are significantly slower than compiler-based ones. Compiler-based MREs, however, are quite difficult to build for DSLs. They must incorporate advanced dynamic profiling and compilation mechanisms in addition to type and value specializations to extract significant performance. Moreover, DSLs semantics are always in flux which makes the latest compiler-based MREs fall short from covering all of the language features. Additionally, compiler-based implementations have high footprint and high startup over-

head which can be problematic for interactive applications. To mitigate the drawbacks of compiler-based MREs while improving performance, we contribute a Remote Compilation Framework (RCF). RCF is a decoupled and distributed MRE. It consists of a pure interpreter-based MRE equipped with a sampling profiler running on the user side. Samples collected are communicated to a remote server where they are aggregated into a global profiler that is used to guide optimizations on hot code. Optimized code is sent back to the user in the form of application update and is run as a native code extension to the interpreter. RCF is also capable of detecting phase shifts in the global profile and can adapt by re-compiling the code. We evaluate RCF with a set of real applications and community benchmarks. We find that RCF improves performance by $1.1 \times -1.7\times$ for real applications and $1.3 \times -3.4\times$ for community microbenchmarks while maintaining $2.7 \times -7\times$ smaller memory footprint than PyPy.

7.2 Future Work

This dissertation is a step towards improving understanding and performance of managed languages. There is still more work needed towards this goal. In this section, we identify directions for future research work that we believe are worth exploring based upon our findings and results.

Although our Performance-Aware Revision Control System (PARCS) guides developers to where to look for code modifications causing a certain performance variation, it does that at the method-level. It then becomes the developer's task to manually drill down into the methods to pin-point the root cause of the variation. The set of probable code modifications can be narrowed down by employing dynamic program slicing [1, 119]. Program slicing will isolate only those modifications that lead to the performance difference (the method call in question). It is interesting to investigate the potential impact of program slicing on PARCS attribution of performance differences.

PARCS only operates at the method-level. That means it only compares call profiles (namely, Calling Context Trees (CCTs)). We believe PARCS accuracy can be improved by using lower level profiles such as basic blocks or edge profiles. Each CCT node can have a sub-profile describing performance within that method. Algorithms for differencing these sub-profiles in a meaningful manner and attributing difference to source code modifications are needed.

Our empirical results show that PARCS CCT differencing algorithm is efficient at tracking performance across close revisions of the code (with minor modifications). However, we currently have no automated means of combining profile differences from close successive revisions to understand the overall difference between two releases. Techniques are needed to combine these differences, analyze them and present them to the user.

In our work to improve Dynamic Scripting Languages (DSLs) performance via interpreter-based optimizations, the performance gain is limited by the size of the body of the dispatch loop. Both attribute caching and load/store elimination rely on adding extra bytecodes which increases the size of the dispatch loop. A larger dispatch loop causes more instruction cache and ITLB misses. It is interesting to explore solutions in software or hardware that mitigate this effect.

Our Remote Compilation Framework for DSLs optimizes code based on an aggregate profile. The aggregate profile is a combined profile from users in the wild and reflects the behavior of the majority of users. However, for larger applications with large behavior variation across inputs, a minority of users who use the application differently may not experience a significant gain in performance. It is of interest to investigate ways that enable RCF to address this problem. One technique that is worth exploring is to group users into clusters based on the samples collected and build an aggregate profile for each cluster. RCF can then optimize for each cluster of users independently.

Finally, our RCF phase detection algorithm depends on a set of parameters that are explicitly set by the user. It would be interesting to pursue ways of automatically and dynamically setting these parameters based on the frequency of the samples received and the amount of information they provide.

Bibliography

[1] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, PLDI '90, pages 246–256, New York, NY, USA, 1990. ACM.

[2] Z. G. Alberto Apostolico. *Pattern Matching Algorithms*. Oxford University Press, 1997.

[3] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 85–96, New York, NY, USA, 1997. ACM.

[4] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 85–96, New York, NY, USA, 1997. ACM.

[5] J. An, A. Chaudhuri, and J. S. Foster. Static typing for ruby on rails. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 590–594, Washington, DC, USA, 2009. IEEE Computer Society.

[6] J. An, A. Chaudhuri, J. S. Foster, and M. Hicks. Dynamic inference of static types for ruby. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 459–472, New York, NY, USA, 2011. ACM.

[7] T. Apiwattanapong, A. Orso, and M. J. Harrold. A differencing algorithm for object-oriented programs. In *ASE '04: Proceedings of the 19th IEEE international conference on Automated software engineering*, pages 2–13, Washington, DC, USA, 2004. IEEE Computer Society.

[8] APT: Debian advanced package tool. `http://wiki.debian.org/Apt`.

[9] M. Arnold, S. Fink, D. Grove, M. Hind, and P. Sweeney. Adaptive Optimization in the Jalapeño JVM. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Oct. 2000.

[10] M. Arnold and D. Grove. Collecting and exploiting high-accuracy call graph profiles in virtual machines. In *International Symposium on Code Generation and Optimization (CGO)*, pages 51–62, Washington, DC, USA, 2005. IEEE Computer Society.

[11] M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. *SIGPLAN Not.*, 36(5):168–179, 2001.

[12] M. Arnold and P. F. Sweeney. Approximating the calling context tree via sampling. Technical report, IBM Research, July 2000.

[13] J. Aycock. A brief history of just-in-time. *ACM Comput. Surv.*, 35(2):97–113, 2003.

[14] R. D. Barnes, E. M. Nystrom, M. C. Merten, and W. mei W. Hwu. Vacuum packing: Extracting hardware-detected program phases for post-link optimization. In *in the 35th International Symposium on Microarchitecture*, pages 233–244, 2002.

[15] Bytecode engineering library. `http://jakarta.apache.org/bcel`.

[16] J. R. Bell. Threaded code. *Communications of the ACM*, 16(6):370–372, 1973.

[17] G. Benson. Zero and shark: a zero-assembly port of openjdk, May 2009. `http://today.java.net/pub/a/today/2009/05/21/zero-and-shark-openjdk-port.html`.

[18] M. Berndl, B. Vitale, M. Zaleski, and A. D. Brown. Context threading: A flexible and efficient dispatch technique for virtual machine interpreters. In *International Symposium on Code Generation and Optimization (CGO)*, pages 15–26. IEEE Computer Society, 2005.

[19] P. Biggar, E. de Vries, and D. Gregg. A practical solution for scripting language compilers. In *Proceedings of the 2009 ACM symposium on Applied Computing*, SAC '09, pages 1916–1923, New York, NY, USA, 2009. ACM.

[20] BitTorrent. `http://www.bittorrent.com/`.

[21] Blackburn, Garner, Hoffmann, Khang, McKinley, Bentzur, Diwan, Feinberg, Frampton, Guyer, and Hosking. The dacapo benchmarks: java benchmarking development and analysis. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Oct. 2006.

[22] M. D. Bond and K. S. McKinley. Probabilistic calling context. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications*, pages 97–112, New York, NY, USA, 2007. ACM.

[23] M. Burke, J. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. Serrano, V. Shreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Dynamic Optimizing Compiler for Java. In *Proceedings of the ACM Java Grande Conference*, pages 129–141, June 1999.

[24] C. Chambers. *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for an Objected-Oriented Programming Language*. PhD thesis, Stanford University, Mar. 1992.

[25] C. Chambers, D. Ungar, and E. Lee. An efficient implementation of self a dynamically-typed object-oriented language based on prototypes. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 49–70. ACM, 1989.

[26] N. Chohan, C. Bunch, S. Pang, C. Krintz, N. Mostafa, S. Soman, and R. Wolski. Appscale: Scalable and open appengine application development and deployment. In *International Conference on Cloud Computing*, 2009.

[27] N. Chohan, C. Bunch, S. Pang, C. Krintz, N. M. S. Soman, and R. Wolski. AppScale design and implementation. Technical Report 2009-02, UCSB, Jan 2009.

[28] CPython. `http://www.python.org/`.

[29] Cython: C-Extensions for Python. `http://cython.org`.

[30] The DaCapo benchmark suite. `http://dacapobench.org/`.

[31] B. Davis and J. Waldron. A survey of optimisations for the java virtual machine. In *PPPJ*, pages 181–183, New York, NY, USA, 2003. Computer Science Press, Inc.

[32] E. H. Debaere and J. M. Campenhout. *Interpretation and Instruction Path Coprocessing*. MIT Press, Cambridge, Massachusetts, 1990.

[33] L. P. Deutsch and A. M. Schiffman. Efficient implementation of the smalltalk-80 system. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '84, pages 297–302, New York, NY, USA, 1984. ACM.

[34] R. B. Dewar. Indirect threaded code. *Communications of the ACM*, 18(6):330–331, June 1975.

[35] A. S. Dhodapkar and J. E. Smith. Managing multi-configuration hardware via dynamic working set analysis. In *Proceedings of the 29th annual international symposium on Computer architecture*, ISCA '02, pages 233–244, Washington, DC, USA, 2002. IEEE Computer Society.

[36] A. S. Dhodapkar and J. E. Smith. Comparing program phase detection techniques. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 217–, Washington, DC, USA, 2003. IEEE Computer Society.

[37] K. Driesen and U. Hölzle. Accurate indirect branch prediction. In *25th Annual International Symposium on Computer Architecture*, pages 167–178, 1998.

[38] M. A. Ertl and D. Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. *SIGPLAN Not.*, 38(5):278–288, 2003.

[39] M. A. Ertl and D. Gregg. The structure and performance of efficient interpreters. *The Journal of Instruction-Level Parallelism*, vol. 5, November 2003.

[40] M. Evers, P. Chang, and Y. Patt. Using hybrid branch predictors to improve branch prediction accuracy in the presence of context switches. In *23rd Annual International Symposium on Computer Architecture*, pages 3–11, 1996.

[41] P. T. Feller. Value profiling for instructions and memory locations. Master's thesis, University of California, San Diego, April 1998.

[42] FindBugs. `http://findbugs.sourceforge.net/`.

[43] S. Fink and F. Qian. Design, Implementation and Evaluation of Adaptive Recompilation with On-Stack Replacement. In *International Symposium on Code Generation and Optimization (CGO)*, Mar. 2003.

[44] The Forth programming language. `http://www.phact.org/e/forth.htm`.

[45] FreeCAD. `http://sourceforge.net/apps/mediawiki/free-cad/`.

[46] M. Furr, J.-h. An, and J. S. Foster. Profile-guided static typing for dynamic scripting languages. In *Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, pages 283–300, New York, NY, USA, 2009. ACM.

[47] M. Furr, J.-h. An, J. S. Foster, and M. Hicks. Static type inference for ruby. In *Proceedings of the 2009 ACM symposium on Applied Computing*, SAC '09, pages 1859–1866, New York, NY, USA, 2009. ACM.

[48] M. Furr, J.-h. D. An, J. S. Foster, and M. Hicks. The ruby intermediate language. In *Proceedings of the 5th symposium on Dynamic languages*, DLS '09, pages 89–98, New York, NY, USA, 2009. ACM.

[49] Y. Futamura. Partial evaluation of computation process - an approach to a compiler-compiler. *Systems, Computers, Controls*, 2:45–50, 1999.

[50] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based just-in-time type specialization for dynamic languages. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 465–478, New York, NY, USA, 2009. ACM.

[51] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 465–478, New York, NY, USA, 2009. ACM.

[52] Google App Engine. `http://code.google.com/appengine/`.

[53] J. Ha, M. R. Haghighat, S. Cong, and K. S. McKinley. A concurrent trace-based just-in-time compiler for single-threaded JavaScript. In *Proceedings of the Second Workshop on Parallel Execution of Sequential Programs on Multi-core Architectures*, Jun 2009.

[54] K. Hansson. Java: Trees vs bytes. BComp Honours Thesis: `http://central.kaserver5.org/Kasoft/Typeset/JavaTree/index.html`.

[55] A. Holkner and J. Harland. Evaluating the dynamic behaviour of python applications. In *In Australasian Computer Science Conference. (ACSC'09)*, 2009.

[56] U. Hölzle and C. C. D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP '91, pages 21–38. Springer-Verlag, 1991.

[57] M. Huang, J. Renau, and J. Torrellas. Profile-based energy reduction for high-performance processors. In *4th Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, December 2001.

[58] International Data Corporation. `http://www.idc.com`.

[59] Ironpython. `http://ironpython.codeplex.com/`.

[60] D. Jackson and D. A. Ladd. Semantic diff: a tool for summarizing the effects of modifications. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, Victoria, BC, Canada, 1994. IEEE Press.

[61] N. D. Jones, C. K. Gomard, and Sestoft. *Partial evaluation and automatic program generation.* Prentice Hall, 1993.

[62] Jython. `http://www.jython.org/`.

[63] H. Kim, J. A. Joao, O. Mutlu, C. J. Lee, Y. N. Patt, and R. Cohn. Vpc prediction: reducing the cost of indirect branches via hardware-based dynamic devirtualization. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 424–435, New York, NY, USA, 2007. ACM.

[64] P. M. Kogge. An architectural trail to threaded-code systems. *Computer*, 15(3):22–32, 1982.

[65] W. Laski and J. Szermer. Identification of program modifications and its applications in software maintenance. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 282–290, Victoria, BC, Canada, 1992. IEEE Press.

[66] H. B. Lee, A. Diwan, and J. E. B. Moss. Design, implementation, and evaluation of a compilation server. *ACM Trans. Program. Lang. Syst.*, 29, August 2007.

[67] B. Liblit, A. Aiken, A. Zheng, and M. Jordan. Bug Isolation via Remote Program Sampling. In *Conference on Programming Language Design and Implementation (PLDI)*, 2003.

[68] Lua programming language. `http://lua-users.org/lists/lua-l/2009-11/msg00089.html`.

[69] Microsoft. Microsoft Corporation, .Net. `http://www.microsoft.com/net/`.

[70] N. Mostafa and C. Krintz. Tracking performance across software revisions. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, PPPJ '09, pages 162–171, New York, NY, USA, 2009. ACM.

[71] E. W. Myers. An o(nd) difference algorithm and its variations. *Algorithmica*, 1:251–266, 1986.

[72] P. Nagpurkar, C. Krintz, M. Hind, P. F. Sweeney, and V. T. Rajan. Online phase detection algorithms. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, pages 111–123, Washington, DC, USA, 2006. IEEE Computer Society.

[73] P. Nagpurkar, C. Krintz, and T. Sherwood. Phase-Aware Remote Profiling. In *International Symposium on Code Generation and Optimization (CGO)*, Mar. 2005.

[74] P. Nagpurkar, H. Mousa, C. Krintz, and T. Sherwood. Efficient remote profiling for resource-constrained devices. *ACM Trans. Archit. Code Optim. (TACO)*, 3(1):35–66, 2006.

[75] OpenJDK. `http://openjdk.java.net/`.

[76] OProfile. `http://oprofile.sourceforge.net/news/`.

[77] A. Orso, D. Liang, M. Harrold, and R. Lipton. GAMMA System: Continous Evolution for Software After Deployment. In *International Symposium on Software Testing and Analysis*, pages 65–69, 2002.

[78] J. Palm, H. Lee, A. Diwan, and E. Moss. When to use a compilation service? In *LCTES*, 2002.

[79] Parrot virtual machine. `http://parrot.org`.

[80] perfmon2: the hardware-based performance monitoring interface for linux. `http://perfmon2.sourceforge.net/`.

[81] Perl programming language. `http://www.perl.org`.

[82] PHP programming language. `http://www.php.net/`.

[83] I. Piumarta and F. Riccardi. Optimizing direct threaded code by selective inlining. *SIGPLAN Not.*, 33(5):291–300, 1998.

[84] J. Plevyak and A. A. Chien. Precise concrete type inference for object-oriented languages. In *Proceedings of the ninth annual conference on Object-oriented programming systems, language, and applications*, OOPSLA '94, pages 324–340, New York, NY, USA, 1994. ACM.

[85] PyBench — a python benchmark suite. `http://svn.python.org/projects/python/trunk/Tools/pybench/README`.

[86] PyGTK: GTK+ for Python. `http://www.pygtk.org/`.

[87] PyPy: Python virtual machine. `http://pypy.org/`.

[88] Python programming language. `http://www.python.org`.

[89] PythonCAD. `http://sourceforge.net/projects/pythoncad/`.

[90] Rietveld, code review for subversion, hosted on google app engine. `http://code.google.com/p/rietveld`.

[91] A. Rigo. Representation-based just-in-time specialization and the Psyco prototype for Python. In *ACM SIGPLAN 2004 Symposium on Partial Evaluation and Program Manipulation - PEPM'04*, Aug 2004.

[92] A. Rigo and S. Pedroni. PyPy's approach to virtual machine construction. In *Proceedings of the Dynamic Languages Symposium*, Oct 2006.

[93] T. H. Romer, D. Lee, G. M. Voelker, A. Wolman, W. A. Wong, J. loup Baer, B. N. Bershad, and H. M. Levy. The structure and performance of interpreters. In *In Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII*, pages 150–159. ACM Press, 1996.

[94] Ruby programming language. `http://www.ruby-lang.org/`.

[95] Scientific python. `http://wiki.python.org/moin/NumericAndScientific`.

[96] M. Serrano, R. Bordawekar, S. Midkiff, and M. Gupta. Quicksilver: a quasi-static compiler for java. In *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '00, pages 66–82, New York, NY, USA, 2000. ACM.

[97] M. Serrano and X. Zhuang. Building approximate calling context from partial call traces. In *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09, pages 221–230, Washington, DC, USA, 2009. IEEE Computer Society.

[98] X. Shen, Y. Zhong, and C. Ding. Locality phase prediction. In *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XI, pages 165–176, New York, NY, USA, 2004. ACM.

[99] X. Shen, Y. Zhong, and C. Ding. Predicting locality phases for dynamic memory optimization. *J. Parallel Distrib. Comput.*, 67:783–796, July 2007.

[100] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, PACT '01, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society.

[101] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, ASPLOS-X, pages 45–57, New York, NY, USA, 2002. ACM.

[102] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *Proceedings of the 30th annual international symposium on Computer architecture*, ISCA '03, pages 336–349, New York, NY, USA, 2003. ACM.

[103] The computer language benchmarks. `http://shootout.alioth.debian.org/`.

[104] Sikuli. `http://sikuli.org`.

[105] E. G. Sirer, R. Grimm, A. J. Gregory, and B. N. Bershad. Design and implementation of a distributed virtual machine for networked computers. In *Proceedings of the seventeenth ACM symposium on Operating systems principles*, SOSP '99, pages 202–216, New York, NY, USA, 1999. ACM.

[106] J. Smith and R. Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufmann, 2005.

[107] Open-source code hosting web-service. `http://www.sourceforge.com`.

[108] SpamBayes. `http://spambayes.sourceforge.net`.

[109] SquirrelFish Extreme JavaScript engine. `http://webkit.org/blog/189/announcing-squirrelfish/`, 2008.

[110] Stackless Python. `http://www.stackless.com/`.

[111] Sun Microsystems Inc. The Java Hotspot Virtual Machine White Paper. `http://java.sun.com/products/hotspot/docs/whitepaper/Java_HotSpot_WP_Final_4_30_01.html`.

[112] K. Tian, E. Zhang, and X. Shen. A step towards transparent integration of input-consciousness into dynamic program optimizations. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 445–462, New York, NY, USA, 2011. ACM.

[113] Unladen-swallow project. `http://code.google.com/p/unladen-swallow/wiki/Benchmarks`.

[114] Unladen-swallow project 2009 q1 results. `http://code.google.com/p/unladen-swallow/wiki/Release2009Q1`.

[115] Unladen-swallow project 2009 q2 results. `http://code.google.com/p/unladen-swallow/wiki/Release2009Q2`.

[116] Unladen-swallow project 2009 q3 results. `http://code.google.com/p/unladen-swallow/wiki/Release2009Q3`.

[117] Google V8 JavaScript engine. `http://code.google.com/p/v8/`.

[118] B. Vijayn and D. V. Ponomarev. Accurate and low-overhead dynamic detection and prediction of program phases using branch signatures. In *Proceedings of the 2008 20th International Symposium on Computer Architecture and High Performance Computing*, pages 3–10, Washington, DC, USA, 2008. IEEE Computer Society.

[119] M. Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[120] J. Whaley. A portable sampling-based profiler for java virtual machines. In *JAVA '00: Proceedings of the ACM 2000 conference on Java Grande*, pages 78–87, New York, NY, USA, 2000. ACM.

[121] Wiki matrix – programming language comparison. `http://www.wikimatrix.org/statistic/Programming+Languages`, Sept 2009.

[122] Yum: Fedora software package manager. `http://fedoraproject.org/wiki/Yum`.

[123] M. Zaleski, K. Stoodley, and A. D. Brown. Yeti: a gradually extensible trace interpreter. In *VEE '07: Proceedings of the 3rd ACM/USENIX international conference on Virtual execution environments*, pages 83–93, New York, NY, USA, 2007. ACM Press.

[124] X. Zhang and R. Gupta. Matching execution histories of program versions. *SIGSOFT Softw. Eng. Notes*, 30(5):197–206, 2005.

[125] T. Zhao. Polymorphic type inference for scripting languages with object extensions. In *Proceedings of the 7th symposium on Dynamic languages*, DLS '11, pages 37–50, New York, NY, USA, 2011. ACM.

[126] X. Zhuang, S. Kim, M. io Serrano, and J.-D. Choi. Perfdiff: a framework for performance difference analysis in a virtual machine environment. In *International Symposium on Code Generation and Optimization (CGO)*, pages 4–13, New York, NY, USA, 2008. ACM.

[127] X. Zhuang, M. J. Serrano, H. W. Cain, and J.-D. Choi. Accurate, efficient, and adaptive calling context profiling. *SIGPLAN Not.*, 41(6):263–271, 2006.