UNDERSTANDING AND OPTIMIZING PYTHON-BASED APPLICATIONS - A CASE STUDY ON PYPY

YANGGUANG LI

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILMENT OF THE REQUIREMENTS
FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

GRADUATE PROGRAM IN ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE
YORK UNIVERSITY
TORONTO, ONTARIO
JULY 2019

Abstract

Python is nowadays one of the most popular programming languages. It has been used extensively for rapid prototyping as well as developing real-world applications. Unfortunately, very few empirical studies were conducted on Python-based applications. There are various Python implementations (e.g., CPython, IronPython, Jython, and PyPy). Each has its own unique characteristics. Among them, PyPy, which is also implemented using Python, is generally the fastest due to PyPy's efficient tracing-based Just-in-Time (JIT) compiler. Understanding how PyPy has been evolved and the rationale behind its high performance would be very useful for Python application developers and researchers.

This thesis is divided into two parts. In the first part of the thesis, we conducted a replication study on mining the historical code changes' of PyPy and compared our findings against Python-based applications from five other application domains: Web, Data processing, Scientific computing, Natural Language Processing, and Media. In the second part of the thesis, we conducted a detailed empirical study

on the performance impact of the JIT configuration settings of PyPy. The findings and the techniques in this thesis will be useful for Python application developers as well as researchers.

Table of Contents

List of Tables

List of Figures

1 Introduction

Python is nowadays one of the most popular programming languages [51]. Due to its dynamic features, rich set of library and framework, and large community, Python is also used quite often during rapid prototyping [77], especially in the fields of web (e.g., [4, 6]), data analysis (e.g., [11, 9]), and machine learning (e.g., [20, 18]) as well as developing many real-world business systems in many large scale and mission-critical systems inside companies like Facebook [66], Google [26], and PayPal [55].

There are many empirical studies done for other programming languages, like Java [79, 45, 42, 38, 59] and JavaScript [53]. However, very few works focused on Python, except the works of [69] and [71]. Studying Python applications have been increasing important, as many popular applications and libraries were written in Python. There are various Python implementations (e.g., CPython, IronPython, Jython, and PyPy). Each implementation has its own unique characteristics. For example, Jython is a Python interpreter implemented in Java and it can be

fully integrated into Java applications, IronPython is tightly integrated with .NET Framework. Among those implementations, PyPy is generally the fastest [14] due to PyPy's efficient tracing-based Just-in-Time (JIT) compiler [31] and it is implemented with Python. Understanding how PyPy has been evolved and the rationale behind its high performance would be very useful for Python application developers and researchers.

There have been very few studies on the evolution of Python-based applications, except the work of Lin et al. [69], the authors conducted an empirical study about fine-grained source code changes on ten Python application across five domains (Web, Data processing, Science computing, NLP, Media). However, the study did not included any applications from the domain of compiler, whose evolution patterns may or may not be the same as other application domains. For example, dynamic features bring great flexibility during development, but are generally slower to execute. PyPy, which is focused on high performance, would its JIT compiler use many dynamic features compared to other application domains? The first part of this thesis will be focused on replicating the above study on the PyPy development history.

PyPy mainly gains its performance via its JIT complier [14]. For dynamic languages, JIT compiler is being used to improve the efficiency of compiling and executing the source code. For example, Java's HotSpot JVM [10] can compile

methods based on its execution frequency. JIT compiling is also used in Chrome's V8 engine [2], which can compile JavaScript code into machine code during runtime. Existing works on the JIT compilation focus on the jitting strategies (e.g., method [38] vs. trace-level based code jitting [31]), speeding up the process of the JIT compilations [63, 47, 70], optimizing the performance of the underlying virtual machines [88, 89, 90], and detecting JIT unfriendly code [53]. Unfortunately, there are very few existing studies which investigate the impact of the JIT configurations on the system performance. Software configuration is one of the main sources of software errors [92]. The configuration settings of a software system can significantly impact its performance. Optimizing the configuration settings may result in great performance gain. Hence, the second part of the thesis will be focused on investigating the performance impact of PyPy's JIT configurations.

The contributions of this thesis are:

1. We have developed a change extraction tool, PyDiff, which can extract fine-grained historical changes from the Python source code.

2. We have replicated the empirical study in [69] by comparing our results from PyPy (a language interpreter written in Python) against 10 Python-based applications from five other domains (Web, Data processing, Science computing, NLP, Media) and found that 6 out of 9 findings are different from the

original study.

3. This is the first empirical study on assessing and optimizing the impact of the tracing-based JIT configuration settings on system performance.

4. In this thesis, we have detailed our search-based configuration tuning approach, (ESM-MOGA) for tuning applications running PyPy.

5. Our experiments on JIT configurations are carried out on both the synthetic benchmarks as well as real systems. The empirical findings can be useful for both software engineering and programming language researchers as well as practitioners.

Thesis Organization:

The thesis is organized as follows: Chapter 2 introduces the related work. Chapter 3 presents our empirical studies on the historical code changes in PyPy and compared our findings to the findings in the original study. Chapter 4 studied and optimized PyPy performance by tuning its JIT configuration settings. Chapter 5 concludes the thesis and discusses some future work.

2 Related Work

In this chapter, we discuss prior research works that are related to this thesis.

2.1 Code Diffing

Software projects are changed rapidly to meet the constantly evolving customer requirements and deployment environments. To help developers better understand the changes in different code commits, many tools have been developed to extract the find-grained code changes by comparing the historical versions of the source code.

J-REX [79] is an evolutionary extractor which can analyze Java-based projects. It uses the token-based technique to compare the changes between each consecutive file revisions and outputs the code revisions in terms of function additions, function deletions and function updates. Fluri and Gall presented a taxonomy of source code changes to be used for change coupling analysis in his work [43]. Fluri et al. [45] also presented the original Abstract Syntax Trees (AST) differencing algorithm,

ChangeDistiller, and described inadequacies concerning the extraction of source code changes. The work of Falleri et al. [42] mainly described their novel tree differencing algorithm, the GumTree differencing algorithm, which is more efficient and accurate. However, neither GumTree nor ChangeDistller provided a mechanism for diffing among Python-based source code. Lin et al. [69] implemented an automatic tool to extract fine-grained source code changes in Python code, named PyCT, based on ChangeDistiller. However, the implementation of PyCT is not available for download. In this thesis, we have developed a fine-grained code diffing tool, called PyDiff. Our tool extracts the ASTs from different versions of the Python-based applications and leverages the GumTree's differencing algorithms to output the fine-grained code changes between two commits.

2.2 Code Change Analysis

In the work of Lin et al. [69], the authors conducted an empirical study on 10 Python projects across 5 domains. They tried to investigate the pattern of change types across project, revisions and maintenance activities. And they provided insights about the change of Python dynamic features. Similar as the work of Lin et al. [69], Romano et al. [75] developed the tool WSDLDiff to extract fine-grained source code changes between different versions of WSDL interfaces and they analyzed the evolution of WSDL interface based on the fine-grained source

code changes in four real world web services. [44] used ChangeDistiller to extract change types from each version of the software and formed a matrix with which they can group changes between versions into different clusters and explore the patterns in each cluster. Thung et al. [86] combined code analysis and machine learning approaches to analyze the bug fix changes. Their approach can predict the code line that caused the bug and the result out-performed the state-of-art. However, there is no existing evolutionary study on Python-based compilers, which is the focus of Chapter 3 of the thesis.

2.3 JIT Compiler

JIT is introduced as a technique to improve the system behavior during runtime by compiling the frequently executed (a.k.a., "hot") code snippets into binaries [31, 73]. Currently, there are two general approaches on recognizing and compiling hot code: (1) the method-based jitting approach [38], which compiles the whole hot method; and (2) the trace-based jitting approach [31], which only compiles the frequently executed code path(s) within one method. Both techniques have their pros and cons and are adopted by different programming languages. The code jitting process takes a while to recognize and compile the hot code snippets [30]. Hence, various techniques have been proposed to speed up the JIT compilations [63, 47, 70]. Since only portions of the source code are jitted, during runtime, depending on the ac-

tual execution path(es), the system may switch between the native mode (a.k.a., executing the compiled binaries) and the interpreter mode. Gong et al. [53] developed a technique to detect performance anti-patterns that prohibit the system to execute certain portions of the code natively. Our paper differs from the above works, as it focuses on the configuration settings of the JIT compiler. The closest work was done by Hoste et al. [59], which proposed a search-based technique to automatically tune the Java compiler. Although the two programming languages differ in their jitting techniques (method-based JIT for Java and tracing-based JIT for PyPy), both [59] and this paper reported the need to automatically tune JIT configurations, as the optimal JIT configurations are system and workload dependent. [59] even found that the JIT configuration tuning is hardware dependent. In this paper, we further studied the characteristics of the PyPy jitting behavior and tried to derive general patterns/guidelines on tuning the JIT configurations. For example, we have found a high correlation between the amount of jitted code and memory utilization. Generally, the configuration parameter *decay* should be set with a small value and a large value in the *trace_limit*.

2.4 Understanding and Tuning the Configuration Settings

Software configuration settings play an important role in the performance of a software system. However, there can be many configuration parameters, each of

which has various possible settings. Hence, the overall configuration space for one system can be huge. In this subsection, we will discuss the related works in the area of assessing and optimizing the configuration settings for one system. We have divided the existing techniques into the following three categories:

- **Understanding the Performance Impact of Various Configuration Parameters Through Experimentation**: Not all configuration parameters can impact the system performance. Hence, researchers have devised a set of experiments with various combinations of the configuration settings to assess the impact of the configuration parameters [33, 84]. Various experimental design techniques (e.g., screening design [93], and covering array [58]) have been applied to assess the performance impact of various configuration parameters. These techniques require a much smaller set of experiments than exhaustively enumerating all the possible combinations of the configuration settings, while still able to identify the high performance impacting configuration parameters.

- **Modeling System Performance Under Different Configuration Settings**: Instead of isolating the impact of various configuration parameters, another approach to assess the performance impact of configuration settings is through performance modeling. Siegmund et al. [81] predicted the system

performance by detecting performance-relevant feature interactions. In their later work [80], Siegmund et al. leveraged machine learning and sampling heuristics to build performance models, which can describe the performance influences among different configuration options and their interactions, from a small set of experiments. Libič et al. [68] used queuing theory to model the performance of the JVM garbage collectors (GCs). Singer et al. [82] built machine learning models using the data from the existing configurations of the GCs. Recently, Jamshidi et al. [61] proposed to use the transfer learning technique to model and infer the system performance under different configuration settings.

- **Automated Tuning of the Configuration Settings**: There are two general approaches to automatically tuning the configuration settings of a software system: (1) through reduction of the possible candidates of optimal configurations (e.g., based on the similarities among configuration settings [74] or through iterative experimentations [85]); (2) through the use of the search-based algorithms (e.g., hill-climbing [91, 87], ParamILS [67], or multi-objective genetic algorithms [59, 83]).

However, there is no existing works focusing on assessing and tuning the performance of PyPy. This is the foucs of Chapter 4 of the thesis.

3 The Replication Study

In this study, we have performed an evolutionary study on the PyPy project. In particular, we compared our findings against one existing evolutionary study on Python-based appications [69].

3.1 Introduction

It has been well understood that software has to be adapted to changing requirements and environments. The source code changes between different versions can provide developers insight into how a software project evolves. Researches in mining software repositories are becoming more and more popular.

There are many works which studied the evolution in source code changes in projects. Romano et al. [75] analyzed the evolution of the WSDL interface based on the fine-grained source code changes in four real-world web services. [44] extract change types from each version of the software and formed a matrix with which they can group changes between versions into different clusters and explore the patterns

in each cluster. However, few of these work studied the evolution of Python-based applications, except the work of Lin et al. [69], the authors conducted an empirical study about fine-grained source code changes on ten Python application across five domains (Web, Data processing, Science computing, NLP, Media). However, the study did not include any applications from the domain of compilers, whose evolution patterns may or may not be the same as other application domains. For example, dynamic features bring great flexibility during development but are generally slower to execute. PyPy, which is focused on high performance, it would be interesting to verify if the JIT compiler use as many dynamic features as other application domains.

In order to study the evolution of the Python-based applications, tools to extract the fine-grained differences between different commits are needed. For example, J-REX [79], ChangeDistiller [44], and GumTree [42] can extract fine-grained Java source code changes. Unfortunately, they do not support Python. PyCT [69] can be used to extract fine-grained changes from Python source code and is based on ChangeDistiller's AST differencing algorithm. But its implementation is not available publically. In this project, we leveraged the framework from PyCT and the GumTree differencing algorithm and developed PyDiff, which can extract fine-grained historical changes from python source code. We conducted a replication study which analyzes the source code evolution for PyPy, which is a Python-based

application in the domain of interpreter. The contributions of this chapter are:

1. We have developed a change extraction tool, PyDiff, which can extract fine-grained historical changes from the Python source code. This tool is publicly available for use [13].

2. We have replicated the empirical study in [69] by comparing our results from PyPy (a language interpreter written in Python) against Python applications from five other domains (Web, Data processing, Science computing, NLP, Media) and we found that 6 out of 9 findings are different from the original study.

3.1.1 Chapter Organization

The rest of the chapter is organized as follows: we introduced the implementation of our Python source code diffing tool, PyDiff, in detail in section 3.2. Section 3.3 gives a summary of the original study and an overview of the findings. Section 3.4 describes the findings in our replication study and discusses the implications. Section discusses the threats and validity and Section 3.6 concludes the chapter.

3.2 Python Source Code Diffing

PyDiff analyzes two consecutive revisions of source code and provides fine-grained change information between these two revisions. The source code changes are categorized into many change types to help software engineer understand how the code got changed. Fig 3.1 gives us an overview of PyDiff. The input of PyDiff is historical revisions of a project, which is usually stored in the Code Repositories (e.g., CVS, SVN, and git.). Within PyDiff, there are three components: the Code Parser (Section III-A) parses Python source code into abstract syntax tree (AST) that can be processed by GumTree; the GumTree Differencing (Section III-B) performs the GumTree differencing algorithm on two ASTs and generates an edit script which contains Actions must be performed on the first AST to obtain the second one; and, the Action Parser (Section III-C) groups these Actions according to their line number and identifies the change type for each group by analyzing the Actions in that group. The output of PyDiff is a CSV file which contains change types for each changed line between two consecutive revisions.

3.2.1 Code Parser

The Code Parser is responsible for converting the Python source code to AST that could be analyzed by GumTree. GumTree has the functionality to directly convert

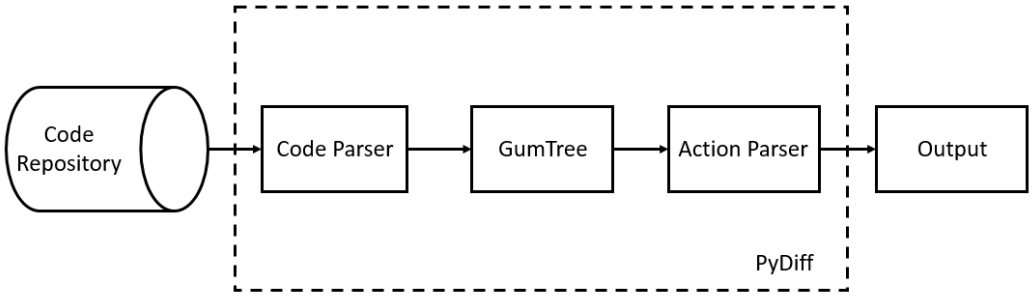

Figure 3.1: Overall process of PyDiff.

Java, C, and C++ source code into a Tree that it can process. However, no such a function has been developed for Python source code. Fortunately, GumTree can accept JSON data as input and can convert JSON data into a Language-Agnostic Tree (LAT) format which it can also analyze. Therefore, we can bring the gap between Python source code and LAT by converting Python AST into JSON AST and use the JSON AST as input for GumTree. The Code Parser takes the following four steps to convert Python source code to LAT that can be analyzed efficiently by GumTree:

3.2.1.1 Python Source Code to Python AST

Python has a module, named "ast", which can generate an AST from python source code. We can use it to parse the Python source code to Python AST easily. The result will be a tree of objects whose classes all inherit from AST. Figure 3.2 gives

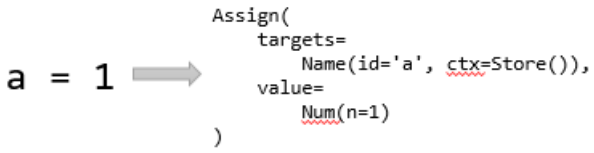

Figure 3.2: An example of converting Python source code to Python AST

an example about how to convert a Python statement "a = 1" to Python AST.

3.2.1.2 Python AST to JSON AST

JSON data structure is similar to XML. In JSON, each node is constructed by a name/value pair, in which name is a string and value could be number, string, list or another node. For each node in Python AST, we can get its class name as well as its child nodes. This structure is very similar to JSON data structure. Therefore, when we covert Python AST to JSON AST, we use the class name of the node as name and use its child nodes as value to construct a field in JSON data structure. By converting all nodes in Python AST, we can obtain a JSON AST that keeps the original structure of Python AST. Figure 3.3 shows the JSON AST for statement "a=1".

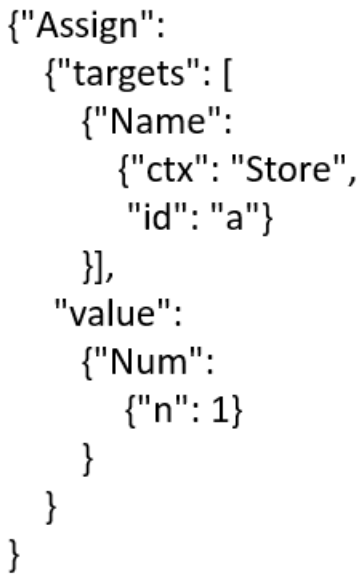

Figure 3.3: An sample converting Python AST to JSON AST

3.2.1.3 JSON AST to LAT

As we mentioned above, GumTree provides an API called TreeGenerator which can take JSON data as input and convert JSON data into a LAT. We can use TreeGenerator from GumTree to generate LAT from JSON AST. Figure 3.4 shows a LAT generated by TreeGenerator from JSON AST of statement 'a = 1'. In the LAT, each node has two fields: node type and label. Node type 4, 10, 15 16 do not have a label because they donate array, field, object, and number in JSON data separately. And node type 17 means string and node type 14 means numbers in JSON.

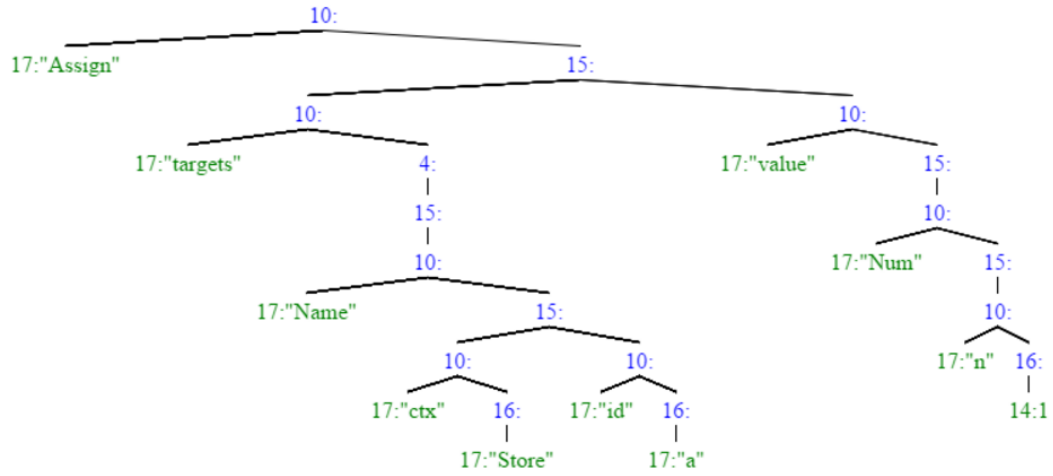

Figure 3.4: An example of converting JSON AST to LAT

3.2.1.4 Simplify LAT

As shown in Figure 3.4, the above parsing steps could introduce many uninformative nodes. For example, a field node will be added to LAT for each field in JSON AST, an object node will be added to LAT for each object in JSON AST. These uninformative trivial nodes make our LAT redundant. They can also introduce misclassification in the GumTree algorithm and bring a large number of uninformative Actions in GumTree's differencing results which are hard to interpret. To address this issue, we simplify the LAT by eliminating the uninformative nodes, like field nodes, object nodes, array nodes, number nodes, string nodes, etc. The simplified LAT of statement 'a = 1' can be seen in Figure 3.5. In the simplified LAT, each node has two fields: node id and label.

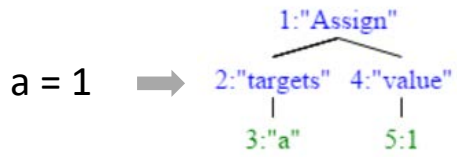

Figure 3.5: Two sample statement with their corresponding ASTs and mappings

3.2.2 GumTree Differencing

The GumTree Differencing component takes two LATs as input and applies the GumTree differencing algorithm to these two LATs. It can compute a sequence of tree edit actions that transform a LAT into another. GumTree Differencing works in two steps: establishing mappings and deducing an edit script. The mapping between two LATs will be computed by two successive phases. 1. A greedy top-down algorithm to find isomorphic subtrees of decreasing height. Mappings are established between the nodes of these isomorphic subtrees. They are called anchors mappings. On the sample trees of Fig 3.6, this step finds the mappings shown with dashed lines. 2. A bottom-up algorithm where two nodes match (called a container mapping) if their descendants (children of the nodes, and their children, and so on) include a large number of common anchors. When two nodes match, it finally applies an optimal algorithm to search for additional mappings (called recovery mappings) among their descendants. On the sample trees of Fig 3.6, this step finds the container mappings shown using short-dotted lines. And there are no recovery

mappings in this example. Then the mapping results can be used by the RTED algorithm [36] to compute the actual edit script. It will generate a tree edit script which contains Actions that must be performed on the first LAT to obtain the second LAT. And those Actions can be used to identify fine-grained source code change types. There are four types of tree edit actions: Insert, Delete, Update and Move. In the example of Fig 3.6, the edit script generated contains only one Action, which is "update label '1' to label '2' in node 5".

Then the mapping results can be used by the RTED algorithm [36] to compute the actual edit script. It will generate a tree edit script which contains Actions that must be performed on the first LAT to obtain the second LAT. And those Actions can be used to identify fine-grained source code change types. There are four types of tree edit actions: Insert, Delete, Update and Move. In the example of Figure 3.6, the edit script generated contains only one Action, which is "update label '1' to label '2' in node 5".

3.2.3 Action Parser

GumTree differencing can only give us changes between two ASTs, because GumTree ignores the semantic meanings of each node in the LAT when it's performing the differencing. Therefore, those changes cannot tell us anything about the source code changes, which is useless in software analysis. To gather source code level

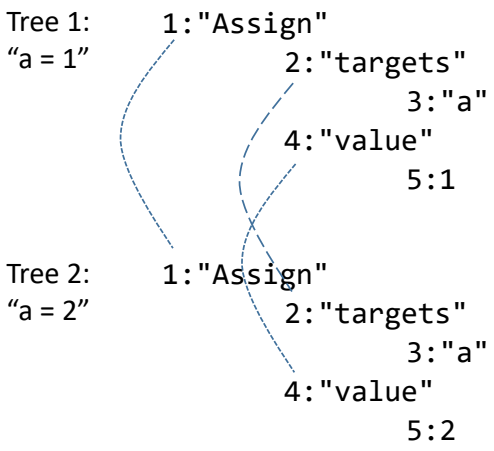

Figure 3.6: Two sample statement with their corresponding ASTs and mappings

change types, we introduced Action Parser to interpret Actions from the GumTree differencing result and obtain source code level change types for all changed lines. This can be done in two steps:

3.2.3.1 Map source code line with Actions

Edit script contains a sequence of Actions from different code lines. With just one Action, we cannot identify the change type for a code line correctly. Therefore, we need to aggregate Actions that are taken in each code line by its line number. For each Action in the edit script, it contains node information of the node that has been changed. And the line number of that node can help us identify which statement that action belongs to.

However, the line number can represent the line in the source code before a

change and it can also represent the line in the source code after a change. Therefore, we cannot simply map the line number back to the before source code line number. In GumTree differencing, the source of the line number is determined by the type of action. The line numbers of changed nodes in Update, Delete and Move Actions represent line number in before source code. The line numbers of changed nodes in Insert Actions represent line number in after source code. In order to deal with the difference, Action Parser processes Insert Actions and other Actions separately by dividing them into before Actions and after Actions. Therefore, it has two groups of Actions which represent the changes made to before and after AST. And it aggregates those two groups of Actions by their line number and builds a map of the source code line and the Actions.

3.2.3.2 Identify change type

In Fluri's work [43], they presented a taxonomy of source code changes that can be used for change coupling analysis. The source code change types are defined according to tree edit operations in the AST. And this classification allows one to assess error-proneness of source code entities, qualify change couplings, or identify programming patterns. The defined change types are shown in Table 3.7. Although the change types are mainly defined for Java programming language and there are many change types not applicable for Python, we took it as a guideline and defined

our change type taxonomy base on the characteristic of Python. The third column in the Table 3.7 indicates whether a change type is applicable for Python or not.

As shown in Table 3.7, 'Additional Object State' and 'Removed Object State' are not applicable because Python doesn't have attribute declaration. Since Python don't have attribute declaration in Python, change types, like 'Decreasing Accessibility Change', 'Increasing Accessibility Change', 'Attribute Type Change', 'Attribute Renaming', 'Final Modifier Insert' and 'Final Modifier Delete' are also not applicable. In method declaration of Python, we cannot define the return type of a method, thus 'Return Type Insert', 'Return Type Delete' and 'Return Type Update' are also not applicable. Base on Fluri's taxonomy of change types, we defined our own fine-grained change types according to our needs. For example, in Fluri's work, he identifies a class declaration statement as a normal statement and inserting or deleting the class declaration will be regarded as 'statement insert' or 'statement delete' action. Therefore, we defined two additional actions 'Additional Class' and 'Removed Class' to help me understand insert and delete changes made to class declaration statements. And we grouped defined fine-grained change types into eight groups: *Class* Change, *Function* Change, *Statement* Change, *Selection Structure* Change, *Loop Structure* Change, *Exception Handling* Change, *Import* Change, and *Others* Change.

For each changed statement, Action Parser identifies the change type for the

changed statement by analyzing Actions taken under the statement. For each changed statement, it can analyze its aggregated Actions by two steps. It firstly identifies if there is any Action made to the root node of that statement. If the root node of the statement is changed, the change type of the statement would be statement level Insert, Delete or Move, which is determined by Action type. If it's not a statement level change, it will identify a detailed change type according to the type of statement and the action type. If these Actions do not contain changes to the statement's root node, the change type would be statement Update. And a more detailed change type can be determined by doing an analysis of the Actions.

From the example in Fig 3.6, Action Parser can obtain an action "update '1' to '2' in node 5". According to the strategy, node 5 is not a root node for the variable assignment statement. Therefore, it can roughly infer that the change type for the statement is statement update. Then, it does further analysis on the change type by combining the statement type, which is assigned statement in our case, and the action made to the statement. And it can determine the change type for that statement is 'statement update'.

3.2.4 Output

In order to help researchers understand the changes in the software, PyDiff will output all changes made to a file throughout the software's life circle into a CSV

Change Type	Description	Applicability
Additional Functionality	add new method	T
Additional Object State	add new attribute	F
Condition Expression Change	change a condition	T
Removed Functionality	delete a method	T
Removed Object State	delete an attribute	F
Statement Delete	delete a statement	T
Statement Insert	insert a statement	T
Statement Order Change	statement moved inside the parent	T
Statement Parent Change	statement moved to another parent	T
Statement Update	statement got updated	T
Class Renaming	a class name got updated	T
Decreasing Accessibility Change	Change a accessibility to a lower level	F
Attribute Type Change	Change the type of an attribute	F
Attribute Renaming	update a attribute name	F
Final Modifier Insert	insert final modifier to a attribute declaration	F
Final Modifier Delete	delete final modifier in a attribute declaration	F
Increaseing Accessibility Change	Change a accessibility to a higher level	F
Method Renaming	rename a method	T
Parameter Delete	delete a paremeter in method declaration	T
Parameter Insert	add a new parameter in method declaration	T
Parameter Ordering Change	change the order of parameters in method declaration	T
Parameter Type Change	change the type of paramter	F
Parameter Renaming	update the name of a parameter	T
Parent Class Delete	delete parent in class declaration	T
Parent Class Insert	insert a new parent in class declaration	T
Parent Class Update	update the parent in class declaration	T
Return Type Delete	delete the return type in method declaration	F
Return Type Insert	add a return type in method declaration	F
Return Type Update	change the return type in method declaration	F

Figure 3.7: Fluri's Taxonomy of Change Type

file. For each change, it will contain information about change type, file path, before revisionID, after revisionID, changed line number, changed location.

3.2.5 Tool Evaluation

In this section, we will explain how our experiment was designed to evaluate PyDiff. In order to test the robustness of PyDiff, we need to choose well-maintained, long history project to evaluate PyDiff. Thus we chose the Python project PyPy to evaluate PyDiff. PyPy has more than 17 years of development history. And it's open-source on Bitbucket, so we can easily obtain all the historical revisions. Based on SVN logs, PyPy was first committed on Feb 24, 2003. With more than 17 years' development history, PyPy contains 97,000 committed revisions and about 4000 python source files. And the evaluation process is as follows:

1. Extracting PyPy history source code from SVM: In order to extract all changed files in each commit, we implemented a small tool to extract changed source files in the current commit and its parent commit by three steps: a. Extract the change log, which contains information about changed files, change type of the file, commit id, revision id, etc., for each commit. b. Recursively extract changed file's source code for each commit and its parent commit. c. Create a JSON file for each commit in PyPy and store changed source code in the JSON file. In this way, we can extract the changed source code for all

commits, on which we can apply PyDiff.

2. Applying PyDiff on PyPy source code: After we obtained the historical source code, we used them as input for PyDiff. PyDiff will run the differencing algorithm on two consecutive source code and generate the change type for each changed line between these two source code files. Then, PyDiff will store all changes in that commits into a CSV file.

3. Manual Examine: After obtaining the output for all commits in PyPy, we evaluated the differencing results by conducting a manual examination. Because there are more than 97,000 commits in PyPy, it is not practical to examine the output of all commits. Therefore, we conducted a manual examine on 10 sample commits. For each selected commit, we manually checked if the identified change type matches the source code change or not and record the number of correctly identified changes.

After manually checking the source code change result from 10 randomly selected commits, we measured the number of changes correctly identified and misclassified in each change type. Table 3.1 shows us the result of the examination. As we can see, PyDiff identified 293 lines of changes among the 10 randomly selected commits. Statement change is the most common change type which takes up 59 percent of all source line changes and achieved 98.8 percentage of accuracy. And

Table 3.1: Identified changes in 10 sampled commits of PyPy.

Change Type	**Total**	**Misclassified**	**Accuracy(%)**
Class	3	1	66.6
Function	31	1	96.7
Statement	174	2	98.8
Selection Structure	32	1	96.8
Loop Structure	6	0	100.0
Exception Handling	43	0	100.0
Import	2	1	50.0
Others	2	0	100.0
All	293	6	97.9

Import change and Other changes occurred the least in the examined commits. Overall, the PyDiff can achieve a precision of 97.9 percent for all changes types.

3.3 Summary of original study

In the work of Lin et al., the authors conducted an empirical study about fine-grained source code changes in Python applications (Django, Tornado, Pandas, Pylearn2, Numpy, Scipy, Sympy, Nltk, Beets, Mopidy) from different domains

(Web, Data processing, Science computing, NLP, Media). In their study, they first defined 77 kinds of fine-grained changes types and then they developed an automatic tool - PyCT, which can quickly compare Python source code and extract fine-grained source code changes. The author proposed four research questions covered different aspects of the source code change behavior throughout the project's history. Then, the authors applied PyCT on ten open source Python projects from five different domains and answered the proposed research questions. Based on the source code diffing result, the author reported 11 major findings, as shown in the Table 3.4, summarised below (Note: Original Fining is denoted as F and New finding is denoted as NF).

First, they compared the change type frequency between projects and domains. They found that *Function* and *Statement* changes are the most frequent change types and *Loop Structure* changes is the least frequent (Finding 1). Also, the distribution of change types shares similar trends between projects (Finding 2) and domains (Finding 3).

Second, they use Scipy and Pandas as case study and compared the change type frequency between different versions of a project. In project Scipy, they found that the change type frequency distribution is significantly different between bug-fix and non-bugfix versions (Finding 4) and language evolution may have a big impact on software evolution (Finding 5). In project Pandas, they found the change type

frequency distribution are similar across different versions (Finding 6).

Then, they conducted a detailed comparison for change type frequency between bug-fix and non-bugfix commits, in which they found that *if structure* related change types are more related to bug-fix activities (Finding 7). And the non-bugfix activity related change types are *import statement* and *function structure* changes (Finding 8).

Finally, they investigated the dynamic feature changes. The result shows that *isinstance*, *type*, *hasattr*, and *getattr* are the most frequently changed dynamic features (Finding 9). The most common change action on dynamic features is Update (Finding 10). 23% dynamic feature changes are related to bug-fix activities and 77% dynamic feature changes are related to non-bugfix activities (Finding 11).

We applied the PyDiff tool on PyPy to get the answer for each research question. And we compared our findings against the findings in the original study and report whether they are similar or different (as shown in Table 3.4). In summary, we found that PyPy, as a compiler, has very different code change behavior compared to the applications from other domains and 6 out of 9 findings are different compared with the original study. In the following sections, we detailed our approach for replication and results for each research question. And we summarised our findings and discussed the implications.

Table 3.2: Comparisons between the original study and the current study

Research Questions	Finding Comparison	Implications	Comparison
Across Projects	**F1**:*Function* Change and *Statement* Change are the most common change types, whereas *Loop Structure* Change is the least common change type. **NF1**: *Statement* Change and *Function* Change are the most common change types, whereas *Import* Change is the least common change type.	PyPy as an implementation of a language rarely uses third party dependencies. Therefore, the Import change is the least common change type.	Different
	F2: The distributions of change type frequency share similar trends across studied projects. **NF2**: The distribution of change type frequency is significant different between PyPy and the projects in the original study.	PyPy, as a Python written compiler, is different from other Python projects in change behavior. More replication studies are needed to generalize the evolutionary behavior of Python applications.	Different
	F3: There are no significant differences among the distributions of change type frequency across studied domains. **NF3**: PyPy is significantly different from the studied Python-based applications in other application domains in terms of the distribution of change type frequency.	The domain of compiler does not share similar change type frequency distribution with the projects from the other application domains. It is worthwhile to study other Python-based compilers to see whether our findings would generalize in this domain.	Different

Table 3.3: Comparisons between the original study and the current study (continued)

Research Questions	Finding Comparison	Implications	Comparison
Across Versions	**F4**:Considering project Scipy, the distributions of change type frequency are different significantly between bug-fix version and non-bugfix version.	Not applicable	Not applicable
	F5: Language evolution may have a huge impact on software evolution.	Not applicable	Not applicable
	F6: The distributions of change type frequency across versions show no significant difference.	Similar to Project Pandas, the distributions of change type frequency across versions in PyPy are similar.	Similar
Maintenance Activities	**F7**: *If structure* related change types are more related to bug-fix, especially *Condition Expression Update* and *If Insert*. **NF7**: *Statement Update* is the only change type that is related to bug-fix.	In PyPy, bugs are more likely to be fixed by *Statement Update*. Bug prediction models for Python-based applications trained in one domain cannot be easily transferred to other domains.	Different
	F8: *Import statement* and *Function Structure* change types are mainly related to non-bugfix activity.	Like the original study, developers are most likely to involve changes like *Additional Function*, *Function Renaming*, and *Parameter Insert* to added new features to the project.	Similar

Table 3.4: Comparisons between the original study and the current study (continued)

Research Questions	Finding Comparison	Implications	Comparison
Dynamic Features	**F9**: In their studied projects, *isinstance*, *type*, *hasattr*, and *getattr* are the top four dynamic features that change frequently.	Similar to other projects, PyPy practitioners should pay more attention to dynamic features like *isinstance*, *type*, *getattr*, *hasattr* and *delete* as well.	Similar
	F10: The change actions on dynamic features follow the pattern that Update is the most common type, followed by Insert and Delete respectively. **FN10**: The most common change action for dynamic features in PyPy is Insert. Delete changes in more than Update changes in most of the dynamic features.	PyPy might involves more dynamic features than normal Python projects. The high performance of PyPy is not a result of constraining the use of dynamic features.	Different
	F11: Among the changes containing dynamic features in the studied projects, about 23% are related to bug fixes, while 77% are related to other changes. **NF11**: Among the changes containing dynamic features in PyPy, about 19% are related to bug fixes.	In PyPy project, bug-fix changes do not involve as many dynamic feature changes as the originally studied projects do.	Different

3.4 Replication Results

In our replication study, we selected PyPy as our subject, because PyPy, as a compiler written in Python, is from a domain that was not studied in the original study. Since the first commit on Feb 24, 2003, there are about 97,000 commits submitted to the project after 16 years of development. About 44 releases published. And the earliest release (pypy-2.1-beta1-arm) was published on June 12th, 2013 with 4180 Python files and 1,393,401 lines of code. The latest release (release-pypy3.6-v7.1.1) was published on Apr 14th, 2019 with 3930 Python files and 987,939 lines of code. The size of the project has been expended for about 10% over the past years. We applied PyDiff on PyPy for source code change extraction. And we answered each of the research questions proposed by the original study in the subsections below. For each RQ in the original study, we described the approach that we conducted the experiment and compared our findings with the findings in the original study and discussed the implications.

3.4.1 (RQ1) Across Projects

In order to get the diffing result for all PyPy commits, we first extracted the changed source code files for each commit and its parent commit. Then we applied the PyDiff in each commits to extract the source code changes within each commit. For each

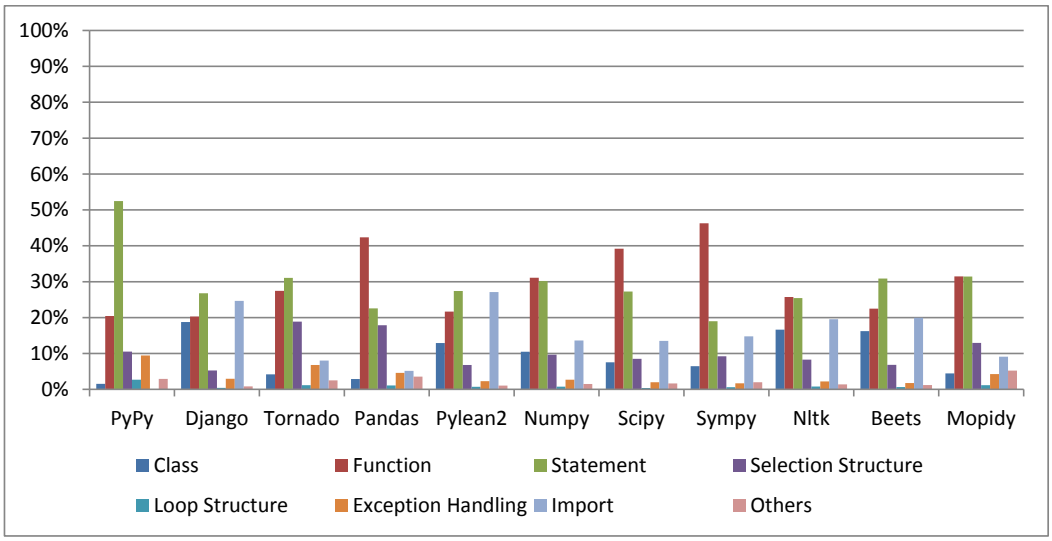

Figure 3.8: The distribution of change type frequency across projects.

change type, we first counted the number of changes of the type in the history, and then we divided it by the total number of changes and got the change type frequency. Figure 3.8 shows the change type frequency for all studied projects.

As shown in Figure 3.8, the *Statement* change has the highest frequency (75%) and the frequency for *Function* change and *Selection Structure* change ranked at the second and third place with a frequency of 52% and 44% respectively. On the other hand, the most uncommon change type in PyPy is *Import* changes.

New Finding 1: *Statement* Change and *Function* Change are the most common change types, whereas *Import* Change is the least common change type.

Implications: PyPy as an implementation of a language rarely uses third party dependencies. Therefore, the Import change is the least common change type.

Based on the measured change type frequency, we applied Wilcoxon rank sum (WRS) test [27] to statistically compare the change type frequency distribution between PyPy and the originally studied projects. A p-value larger than 0.05 indicates there is no significant difference in change type frequency distribution between PyPy and the originally studied projects. In addition, we group the change type frequency based on the application domains. And we applied the WRS test to statistically compare the change type frequency distribution between the domain of compiler with five other domains in the original study.

The statistical result indicates that the change type frequency distribution in PyPy is significant different from the projects in the original study. And the change type frequency in the domain of compiler is significantly different from all the five domains in the original study.

New Finding 2: The distribution of change type frequency is significant different between PyPy and the originally studied projects.

Implications: PyPy, which is a compiler written in Python, is different from other Python projects in terms of change behavior. More replication studies are needed in order to generalize the evolutionary behavior of Python-based applications.

New Finding 3: PyPy is significantly different from the studied Python-based applications in other application domains in terms of the distribution of change type frequency.

Implications: The domain of compiler does not share similar change type frequency distribution with the projects from the other application domains in the original study. It is worthwhile to study other Python-based compilers to see whether our findings would generalize in this domain.

3.4.2 (RQ2) Across Versions

Base on the source code diffing result for each commit in RQ1, we grouped the commits into releases. For all commits in each release, we measured the change type frequency. Figure 3.9 shows the change type frequency for the 11 PyPy releases.

As we can see from Figure 3.9, the change type frequency distribution across different versions show a similar trend. To further examine the finding, we conducted the WRS test between the change type frequency distribution between different PyPy releases. The statistic result shows that there is no significant difference in change type frequency distribution across different PyPy releases.

In addition, we looked into the release logs and check if any of the releases is purely a bug-fix release. But the release log shows that all the 11 release are mixed with changes about bug-fix, new features, and improved test coverage. Therefore,

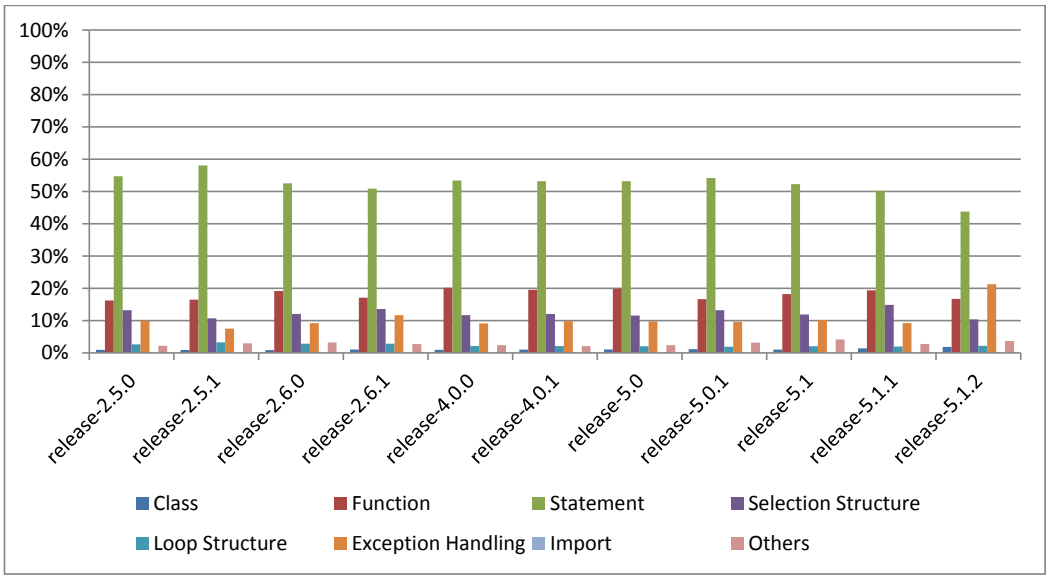

Figure 3.9: The frequency comparison between different versions of PyPy.

Finding 4 and Finding 5 in the original study do not hold in the PyPy project.

Finding 6: The distributions of change type frequency across versions show no significant difference.

Implications: Similar to Project Pandas, the distributions of change type frequency across versions in PyPy are similar.

3.4.3 (RQ3) Maintenance Activities

In the original study, they found that the distribution of change type frequency in pure bug-fix versions are different from others. In order to understand the code changes, they conducted a study comparing changes types frequency in bug-fix activities and non-bugfix activity. First of all, we divided the commits into bug-fix

and non-bugfix by mining the commit messages. For each commit, we got the commit message and convert the words into word stems. Then, we checked if the word stems contain the bug-fix related keywords. In particular, we identified a commit as a bug-fix commit if the commit message contains word stems for the following bug-fix related keywords: 'error', 'bug', 'fix', 'issue', 'mistake', 'incorrect', 'fault', 'defect' and 'flaw'. Figure 3.10 provides an overview of the change type frequency among bug-fix and non-bugfix commits. And then, for each change type, the bug-fix and non-bugfix commits are divided into four groups (1) bug-fix commits containing this change type; (2) bug-fix commits not containing this change type; (3) non-bugfix commits containing this change type; (4) non-bugfix commits not containing this change type. At last, we applied Fisher's exact test on the four groups which can measure the significance of the association between two classifications. In our case, a p-value smaller than 0.05 indicates that the change type differs between bug-fix and non-bugfix behavior. And the Odds Ratio from Fisher's exact test can quantify the strength of the association between two events. An Odds Ratio equals one means the change type behave the same in bug-fix and non-bugfix commits. If the odd ratio is larger than 1, the change type is more related to bug-fix commits and vice versa.

In the result of Fisher's exact test, only *Statement Update* is is related to bug fixes with a p-value equals 3.54e-48 and an Odds Ratio of 1.27. However, there are in

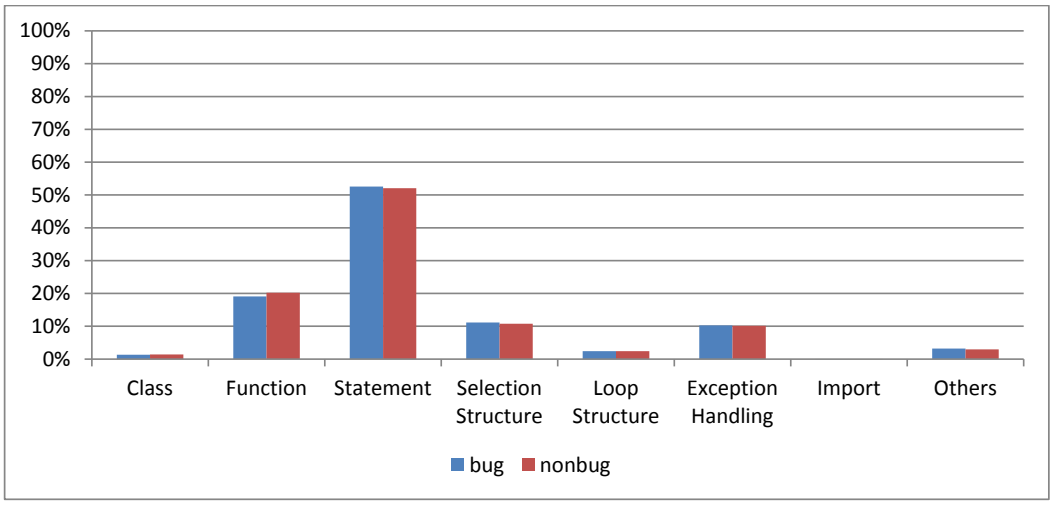

Figure 3.10: The frequency comparison between bug-fix commits and non-bugfix commits of PyPy.

total 62 fine-grained changes types which is not related to bug fixes. In particular, many function level changes, e.g. *Additional Function*, *Function Renaming*, and *Parameter Insert* are non-bugfix related.

New Finding 7: *Statement Update* is the only change type that is related to bug-fix.

Implications: In PyPy project, bugs are more likely to be fixed by *Statement Update*. Bug prediction models for Python-based applications trained in one domain cannot be easily transferred to other domains (e.g., compilers).

Finding 8: *Import statement* and *Function Structure* change types are mainly related to non-bugfix activity.

Implications: Like the originally studied projects, developers are most likely to involve changes like *Additional Function*, *Function Renaming*, and *Parameter Insert* to added new features to the project.

3.4.4 (RQ4) Dynamic feature

To study the pattern of dynamic feature changes in PyPy project, we further implemented the PyDiff tool and included the functionality to identify dynamic feature changes in the source code files. And then, we re-applied the PyDiff in all commits with the dynamic feature change function turned on and got the dynamic feature changes in each commit. Based on the dynamic change results in PyPy, we measured the dynamic feature change type frequency by dividing the number of commits containing this dynamic feature change with the total number of commits containing dynamic feature changes.

Figure 3.11 shows the distribution of the frequency of various change types related to the dynamic features in PyPy and the originally studied projects. Among all dynamic feature changes, more than 60% commits contain *isinstance* change. The second most frequently changed dynamic feature in PyPy is *getattr*, with a percentage of 22%. In addition, dynamic features *type*, *hasattr*, and *del* are also

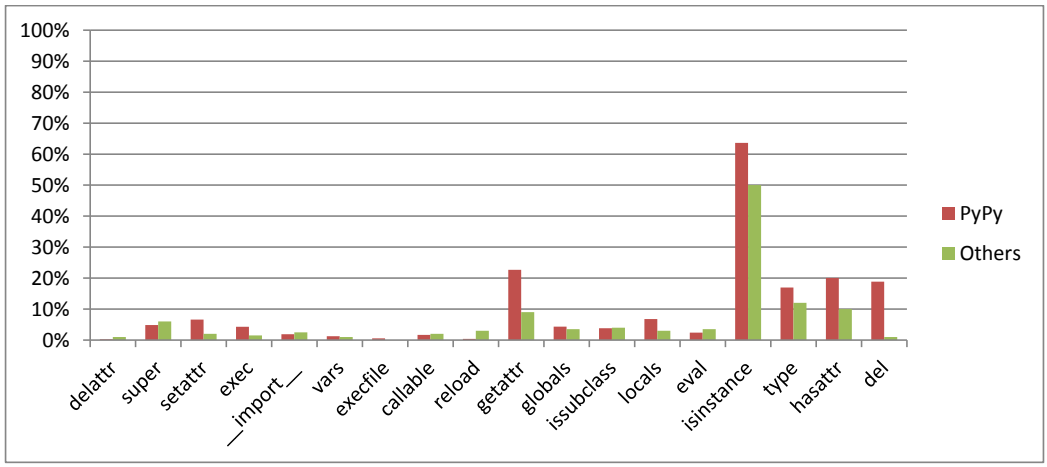

Figure 3.11: The change frequency of each dynamic feature.

commonly changed in PyPy.

Finding 9: In their studied projects, *isinstance*, *type*, *hasattr*, and *getattr* are the top four dynamic features that change frequently.

Implications: Similar to other projects, PyPy practitioners should pay more attention to dynamic features like *ininstance*, *type*, *getattr*, *hasattr* and *delete* as well.

We grouped the dynamic feature changes into three categories based on the change type: Insert, Delete, and Update. Figure 3.12 shows the percentage of each change actions within a dynamic feature change for the most commonly changed five dynamic features. As shown in Figure 3.12, Insert is the most common change action for all the five dynamic feature changes, with a percentage from 48% to 54%. And the second common change action is Delete for most of the dynamic features

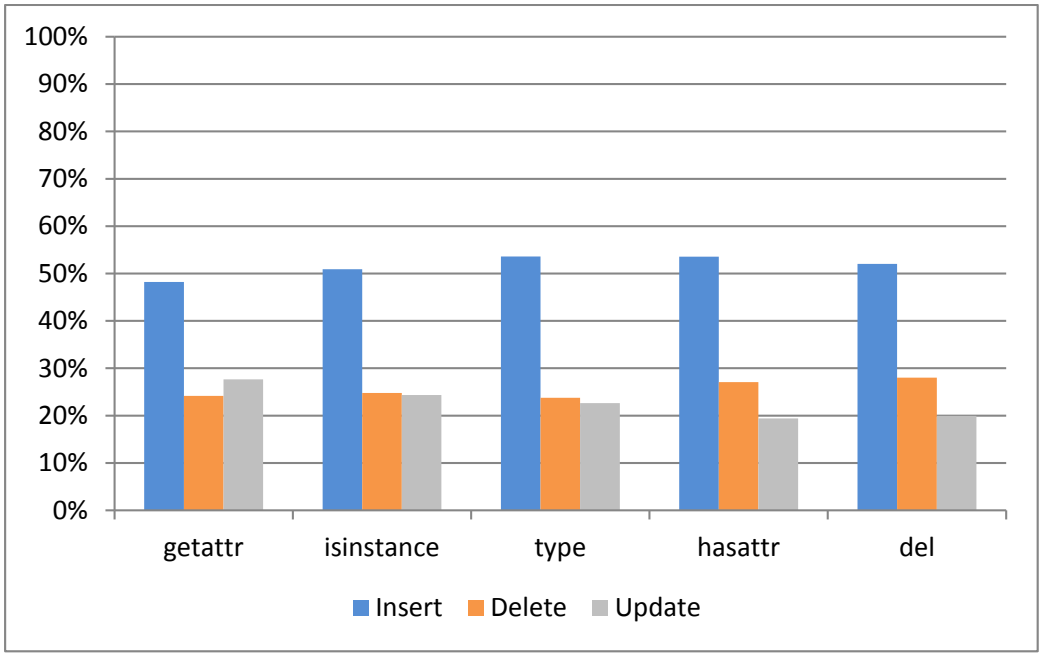

Figure 3.12: The change frequency of each dynamic feature.

changes except *getattr*.

New Finding 10: The most common change action for dynamic features in PyPy is Insert. Delete changes in more than Update changes in most of the dynamic features.

Implications: PyPy might involves more dynamic features than normal Python projects. The high performance of PyPy is not a result of constraining the use of dynamic features.

In order to understand the how the dynamic features changed under different maintenance activities, we leveraged the identified bug-fix commits and non-bugfix commits in RQ3 and divide the commits that contain dynamic features into two

groups: commits containing dynamic feature change that are bug-fix related and commits containing dynamic feature change that are non-bugfix related.

New Finding 11: Among the changes containing dynamic features in PyPy, about 19% are related to bug fixes.

Implications: In PyPy project, bug-fix changes do not involve as many dynamic feature changes as the originally studied projects do.

3.5 Threats to Validity

In this section, we will discuss the threats to validity.

3.5.1 Construct Validity

Since the source code change can be very complicated, correctly identifying code changes between two commits can be hard. To make sure that our source code diffing tool work properly, we manually examined 10 commits which contains more than 10 changes. The results show that all the code changes that we identified have more than 98% confidence level.

3.5.2 Internal Validity

The taxonomy of Python source code is defined by ourselves. In order to make a proper comparison with the original study, we must make sure the change types we defined are the same as the original study. Therefore, we defined our classification

scheme based on the Fluri's [43] taxonomy for source code change in object-oriented language, which is also used by the original study.

3.5.3 External Validity

The study is conducted in Python-based applications, the findings may not be generalizable to applications in other programming languages (e.g. Java, C). The findings we got from the source code change result of PyPy project from the domain of compiler is very different from the other Python applications and domains. We plan to extend our study to the applications in other domains as well as other programming languages in the future.

3.6 Conclusion

Lin et al. [69] conducted the first work in studying the evolution of Python-based projects by studying 10 Python applications from 5 different application domains. They studied the Python source code change behavior from four different angles: across projects, across versions, maintenances, and dynamic features and got 11 interesting findings. We performed a replication study on the PyPy project which belongs to the domain of compiler that has never been studied. We found that 6 out of 9 findings are different which indicates the evolution of projects from the domain of compiler is very different from the project. In the next chapter, we will dig more

into the reason for PyPy's fast speed and find ways to optimize the performance of Python-based applications running under PyPy.

4 Assessing and Optimizing the Performance Impact of the Just-in-time Configuration Parameters

In the previous chapters, we have studied the evolution of PyPy. In this chapter, we assess and optimize the performance of applications running under PyPy by tuning its JIT configuration parameters.

4.1 Introduction

Software performance is one of the crucial factors related to the success and the sustainability of large scale software systems, which serve hundreds or even millions of customers' requests every day. Failure to provide satisfactory performance would result in customers' abandonment and loss of revenue. For example, Amazon reported that one second delay in loading their webpages could result in $1.6 billion loss in their sales revenue annually [41]. BBC has also recently found that 10% of

the users will leave their website even if there is merely one additional second of performance delay [37]. Various strategies (e.g., asynchronous requests [60], data compression [54], just-in-time (JIT) compilation [10], load balancing [25], and result caching [34]) have been developed to further enhance the performance of these systems.

In general, there are three types of system executions depending on the programming languages: (1) executing natively on top of the operating systems (e.g., C and C++), (2) executing the source code by the interpreters (e.g., Python, PHP, and JavaScript), and (3) executing compiled intermediate artifacts on the virtual machines (e.g., Java and C#). Compared to the native execution mode, systems executed under the interpreted mode (a.k.a., by interpreters or virtual machines) are generally slower due to their additional layers. To cope with this challenge, the JIT compilation is introduced so that frequently executed code snippets are compiled into binaries, which can be executed natively.

Existing works on the JIT compilation focus on the jitting strategies (e.g., method [38] vs. trace-level based code jitting [31]), speeding up the process of the JIT compilations [63, 47, 70], optimizing the performance of the underlying virtual machines [88, 89, 90], and detecting JIT unfriendly code [53]. Unfortunately, there are very few existing studies which investigate the impact of the JIT configurations on the system performance. Software configuration is one of the main sources of

software errors [92]. The configuration settings of a software system can significantly impact its performance. Most of the existing configuration tuning and debugging studies are focused on the configurations of the studied systems [40, 93, 84, 61]. For tuning the configuration settings of interpreters or virtual machines, the focus is mainly on optimizing the performance of the garbage collectors [82, 67, 33], except the work by Hoste et al. [59]. In [59], Hoste et al. provided an automated approach to tuning the JIT compiler for Java, which is a method-based JIT. Hence, in this paper, we seek to investigate the impact of the tracing-based JIT configurations on the system performance by using PyPy as our case study subject.

Python is nowadays one of the most popular programming languages [51]. Python has been used extensively to develop real-world business systems, including many large scale and mission-critical systems inside companies like Facebook [66], Google [26], and PayPal [55]. Among various implementations of the Python programming language (e.g., CPython, IronPython, Jython, and PyPy), PyPy is generally the fastest [14] mainly due to PyPy's efficient tracing-based JIT compiler [31]. Hence, in this paper, we focus on assessing and optimizing the performance impact of PyPy's JIT configuration settings. The contributions of this chapter are:

1. This is the first empirical study on assessing and optimizing the impact of the tracing-based JIT configuration settings on system performance.

2. Our experiments are carried out on both the synthetic benchmarks as well as real systems. The empirical findings in this paper can be useful for both software engineering and programming language researchers as well as practitioners.

3. Compared to [59] which also used a search-based approach to automatically tuning the JIT configuration settings, many of the details (e.g., the initial setup, the configuration settings, and the evaluation details) are not clear. It is not easy to reapply the approach to other Java-based systems or other JIT compilers. In this paper, we have detailed our search-based configuration tuning approach, (ESM-MOGA) to ease replication.

4. To enable replication and further research on this topic, we have provided a replication package [16] which includes the implementation for our configuration tuning framework, PyPyJITTuner, as well as the experimental data.

4.1.1 Chapter Organization

The organization of this chapter is as follows: Section 4.3 describes the exploratory study that we have conducted to understand the relationship between JIT configuration settings and its performance. Section 4.4 proposes our approach to automatically tuning the JIT configuration parameters. Section 4.5 shows the result from

```
     6 def test():
     7   even = 0
     8   oddLarge = 0
     9   oddSmall = 0
    10   c = 0
(*) 11   for i in range(1000000):
(*) 12     if i%2 == 1:
(*) 13       even += 1
(*) 14     elif i * i <= 100:
    15       oddSmall += 1
(*) 16     elif i * i > 100 and i < 1000000:
(*) 17       oddLarge += 1
    18     else:
    19       c += 3
    20   return c
```

Figure 4.1: A sample PyPy code snippet with the jitted code marked as "(*)".

the case study conducted on three real word systems and evaluate the effectiveness of our approach. Threats and validity is discussed in Section 4.7. Section 4.8 concludes this chapter.

4.2 Background

In this section, we will first give an overview of the JIT compilation process in Section 4.2.1. Then we will explain PyPy's JIT configuration setting in Section 4.2.2.

4.2.1 An Overview of the JIT Compilation Process

JIT compilers are introduced for systems executed by interpreters (e.g., Python, PHP, and Ruby) or virtual machines (e.g., Java and C#) to further speed up the system performance during runtime. By default, there are no JIT compilations

upon the initial system startup and these systems are executed under the interpreted mode by their interpreters or virtual machines. Hence, their performance is usually slower than natively executed systems (e.g., systems programmed in C or C++), whose binaries are executed directly on top of the operating systems. To cope with this limitation, the JIT compiler is introduced so that, during runtime, various parts of the systems are converted into machine executable code (a.k.a., code jitting). However, the code jitting process is usually slow, as it takes time to load and compile the corresponding code snippets. Hence, only the commonly used (a.k.a., "hot") code snippets are usually jitted [31, 73]. For such systems, there is usually a warmup period after the initial system startup before these systems reach the peak performance [30]. During the warmup period, the frequently executed code will be profiled to locate the "hot" spots and various code snippets are being jitted [70]. In general, there are two approaches for code jitting depending on their granularity:

- **Method-based JIT Compiling**: if one method has been used many times (a.k.a., "hot method"), the method-based JIT will compile this entire method into the binary executable format. Hotspot (Oracle's implementation of the Java Virtual Machine) and Chakra (Microsoft's JavaScript engine) use the method-based JIT Compiling.

- **Trace-based JIT Compiling** is more fine-grained, in which only the commonly executed code path (a.k.a., "hot path") inside a method is compiled into the binary executable format. PyPy and TracingMonkey (Mozilla's JavaScript engine) use the trace-based JIT Compiling.

For the method-based JIT compiler, there will be a threshold value (e.g., 1500 as the default value for the configuration parameter *-XX:CompileThreshold* in Oracle's HotSpot JVM), which defines the number of invocations for a particular method before this method is considered to be hot. As soon as a method has been called 1500 times, the whole method will be compiled into the binary executable format.

For the trace-based JIT compiler, the process is a bit more complicated. We will use the sample code snippet shown in Figure 4.1 to explain. There can be various configuration parameters which define a particular code path to be hot. For example, in PyPy, there is a configuration parameter, called *threshold*, which defines the number of times a loop has to be run before it can be considered hot. During the system execution, the PyPy JIT compiler counts the number of iterations for each loop and all code paths in the loop will be potential candidates for code jitting. For example, the code lines marked with star (*) in Figure 4.1 are the resulting jitted lines, if the method *test* is executed in PyPy under the default configuration setting. After the loop reaches 1039 (the default value for *threshold*) iterations, the JIT compiler starts to trace the code path in the next iteration. And the code

path which contains the if branch, and the second $elif$ branch will be recorded and compiled into efficient machine code. The first $elif$ and the $else$ branches are not jitted, as they are not in the code path of the traced iteration.

4.2.2 PyPy's JIT Configuration

The types and the values of the JIT configuration parameters vary depending on the programming languages and the compilers. For example, there are more configuration parameters for PyPy's JIT compiler than Java's JIT compiler. Even within the same programming language, different language implementations may use different configuration parameters. For example, in Java, the configuration parameter which indicates the threshold value for the number of invocations for a method before code jitting is called *-XX:CompileThreshold* in Oracle's HotSpot JVM [10], and *-Xjit:count* for IBM's JVM [7]. In this paper, we have selected PyPy's JIT configuration parameters as our case study subject, due to the popularity of the Python programming language [51] and the fast execution under PyPy with its efficient JIT compiler [31, 14]. The list of JIT configuration parameters can be obtained through running the `pypy --jit help` command. For PyPy version 5.7.1, which is the PyPy version used in this paper, there are 19 of them.

4.3 Exploratory Study

To motivate the importance of this work, we have conducted an exploratory study on the performance impact of PyPy's JIT configuration settings. We seek to answer the following three research questions:

- **RQ1:** *How different is the system performance before and after its code has been jitted?*

  When the system initially starts up, all of its code is executed under the interpreted mode. The code jitting process will not start, until certain regions of code have been repeatedly executed many times. In this RQ, we want to quantify the performance differences between the warmup and the warmed up phases.

- **RQ2:** *What is the performance impact by varying JIT configurations?*

  The system after the warmup phase would achieve its peak performance. But would the peak performance be different among different JIT configurations settings (e.g., the default config, random configurations, or disabling JIT)? In this RQ, we seek to find the performance impact of different JIT configurations.

- **RQ3:** *Do systems containing more jitted lines yield better performance?*

Different JIT configuration settings would result in different amount of source code been jitted. Intuitively, a higher portion of the jitted code could lead to more code being executed natively, and hence result in better performance. However, the code jitting process is very resource heavy, which involves profiling system executions and compiling the hot code path into the binary executable format. In addition, the systems may need to constantly switch between the two running modes (the interpreted vs. the native execution mode). The goal of this RQ is to examine whether there is any relation between the portions of the jitted code and the system performance.

The remaining three subsections in this section will address the above three research questions. For each research question, we will first explain the experimentation process. Then we will describe the data analysis techniques, present the result findings, and discuss their implications.

(RQ1) How different is the system performance before and after its code has been jitted?

During the benchmarking and the performance testing processes, it is considered as a common practice to wait for a period of time (a.k.a., the warmup phase) for the system to stabilize [32, 8], before starting the actual benchmarks or the performance tests. During the warmup phase, various regions of the code are getting jitted and

the system caches are slowly being filled up. Hence, the performance of the warmup phase is generally considered as suboptimal and is discarded during the subsequent performance analysis. In this RQ, we want to quantitatively compare the system performance during and after the warmup phase.

Experiment

To tackle this research question, we selected the following two microbenchmark suites, which assess the performance of different software systems:

- *The PyPy benchmark suite* is run daily on PyPy's nightly builds and is mainly used to compare the performance of various Python implementations (PyPy vs. cPython). The benchmark suite consists of about 60 small Python programs, which perform various computation tasks like the n-queens solver, HTML table building, etc. For each run of the benchmark, the same benchmark programs will be run under PyPy and cPython (the default Python implementation). The performance results are uploaded and visualized in the PyPy's Speed Center [14]. In this exploratory study, we randomly selected seven benchmark programs as shown in Table 4.1 for experimentation. We further instrumented these benchmark programs to gather additional performance information (e.g., individual request response time).

- *The TechEmpower Web Framework Benchmark suite*[1] [19], whose main objective is to evaluate among various web frameworks, consists of more complicated web application-related tasks like JSON serializations, database accesses, and server-side template compositions. Different from the PyPy benchmark suite, whose programs are usually short-lived and computation intensive, the TechEmpower benchmark suite executes on long running web application servers built with various frameworks. For example, the benchmark includes Java-based web frameworks (e.g., Jetty), as well as Python-based web frameworks (e.g., Tornado and Flask). In this paper, we only focus on the Django web application frameworks.

We ran the two microbenchmark suites under the default PyPy configuration setting. To avoid measurement bias and errors [50], for the PyPy benchmark, in which the studied programs are short-lived and computational intensive, we repeated the benchmark for 30 times. For the TechEmpower benchmark, which examines the performance of processing web requests for long-running servers, we set the duration for each benchmark task to be two hours. During the benchmarking process, resource utilizations (e.g., CPU, memory, and disk) for the servers were monitored and recorded using pidstat [12]. We also added additional instrumentation using the JIT logging function from PyPy's *jitlog* module to record the JIT

[1]To ease explanation, we will call this the TechEmpower benchmark in the rest of this paper.

Table 4.1: PyPy benchmark programs description.

Program	**Description**
ai	Test the performance of simple AI solvers.
bm_mako	Benchmark for test the performance of Mako templates engine.
chaos	Test the performance of the Chaos benchmark. Create chaosgame like fractals.
django	This will have Django generate a 100x100 table as many times as you specify.
rietveld	This will have Django render templates from Rietveld with canned data as many times as you specify.
html5lib	Test the performance of the html5lib parser.
pidigits	Test the pidigit calculation performance.

logs to the disk. The recorded JIT logs can be further parsed with VMProf [23] to obtain the exact lines of code that were jitted. However, the recorded JIT logs do not have timestamps to mark when a code snippet is jitted. To estimate the exact timing when individual code snippets are jitted, we decided to periodically take snapshots of the JIT log files. We took snapshots of the JIT log files after each iteration of the PyPy benchmark and every minute for the TechEmpower benchmark. These snapshots would help us gain insights on the time and the location of the jitted code regions. Finally, we also archived the benchmarking logs, so that we can extract the response time for each individual request.

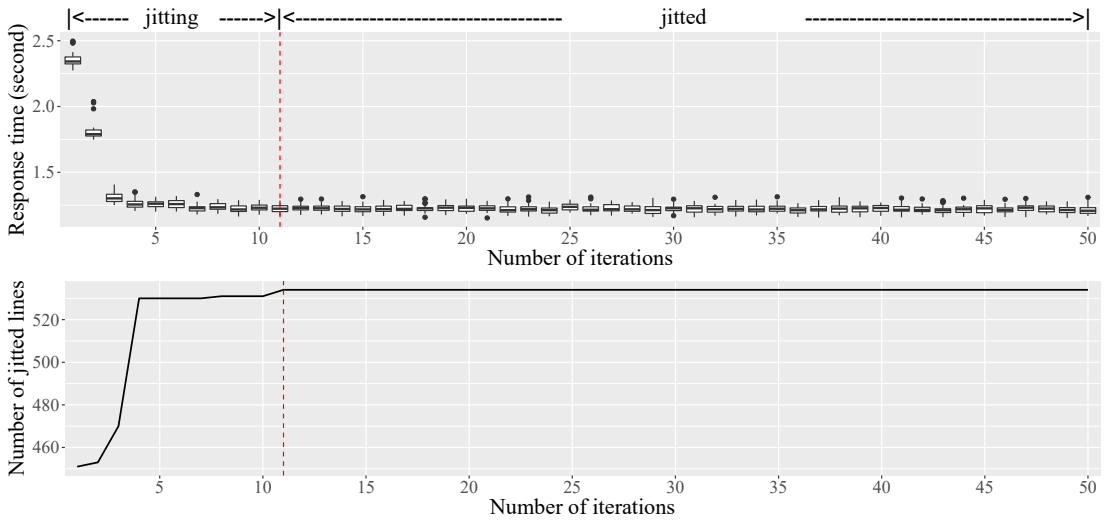

Figure 4.2: Number of jitted lines and response time over 50 iterations for the *html5lib* program from the PyPy benchmark suite.

Data Analysis

We parsed the JIT logs using VMProf to obtain the regions of the jitted code during each snapshot period. We also processed the benchmarking logs to extract the response time for each iteration of the PyPy benchmark and the response time for individual requests in the TechEmpower benchmark.

To understand the performance of the systems during and after the warmup phase, we need to determine the duration of the warmup phase. For the PyPy benchmark, we kept track of the amount of the jitted code during each iteration. We considered the warmup phase to be completed, when the amount of the jitted code remains stable during the remaining of the benchmarking run. Figure 4.2 shows the result for the *html5lib* program from the PyPy benchmark. The topper subgraph of Figure 4.2 shows the how the response time evolve over different number of iterations. Since each program within the PyPy benchmark is repeatedly executed 30 times, we aggregated the response time for that iteration across the 30 runs using boxplots. For example, the first boxplot contains all the response time values for the first iteration during the 30 runs. The bottom subgraph of Figure 4.2 shows the evolution of number of jitted lines across different iterations. The red dotted lines in both subgraphs indicates the iteration when the number of the jitted lines becomes stable. Hence, we considered the first 11 iterations as the warmup phase ("jitting")

and the remaining iterations (a.k.a., the 12th to the 50th iterations) as the warmed up phase ("jitted"). The response time is the highest during the initial iteration, and gets slowly improved when the amount of the jitted code increases. After the 11th iteration, the response time stabilizes. For the TechEmpower benchmark, we used a similar approach as the PyPy benchmark and divided response time into the warmup phase and the warmed up phase based on the time when the number of jitted lines gets stabilized.

We applied statistical techniques to rigorously compare and quantify the differences between the response time distributions from the two phases. Statistical test like the Wilcoxon Rank Sum (WRS) test would give us a rigorous measurement if the distributions of the performance data from these two phases are different. However, in some cases, even if the distributions are different, the differences between the two distributions can be small. For example, if the response time for one request is long (e.g., more than five minutes) and the differences of the response time between the two experiments are very small (e.g., one millisecond), such performance differences would not be useful for our study, as it will not be noticed by the end-users. Hence, we also need to quantify the strength of the differences between the two distributions, called the *effect size* [65]. In this paper, we used Cliff's Delta (CD) as our effect size measures. Both CD and WRS are non-parametric techniques. Hence, they do not hold any assumptions regarding the distributions

of the data. We consider two datasets as statistically different, when the p-value from the WRS test is lower than 0.05. The strength of the differences and the corresponding range of CD values [76] are shown below:

$$\text{effect size} = \begin{cases} \text{trivial} & \text{if } |CD| < 0.147 \\ \text{small} & \text{if } 0.147 \leq |CD| < 0.33 \\ \text{medium} & \text{if } 0.33 \leq |CD| < 0.474 \\ \text{large} & \text{if } 0.474 \leq |CD| \end{cases}$$

We used the following criteria to judge if the response time from the warmed up phase (denoted as B) is getting better ($>$), worse ($<$), or relatively the same ($\sim$) as the warmup phase (denoted as A):

$$\text{difference} = \begin{cases} \text{A} > \text{B} & \text{if } CD \leq \text{-0.33 and } p\text{-value} < 0.05 \\ \text{A} \sim \text{B} & \text{if } |CD| < 0.33 \text{ or } p\text{-value} \geq 0.05 \\ \text{A} < \text{B} & \text{if } CD \geq 0.33 \text{ and } p\text{-value} < 0.05 \end{cases}$$

The p-values shown above are calculated from the WRS test. In other words, B improves over A ($B > A$) when the WRS test and the CD value satisfy the following three conditions: (1) the two distributions are statistically significantly different (p-value < 0.05), and (2) the differences between the two distributions have medium or large effect sizes, and (3) CD value is positive, indicating B is smaller than A. The conditions for B degrades from A ($B < A$) is similar, except

the CD value is negative, indicating B is bigger than A. If A and B are relatively the same ($B \sim A$), when there is no statistical difference between the two distributions (a.k.a., p-value ≥ 0.05) or the effect size between A and B is small or trivial.

We extracted the performance data from the warmup and the warmed up phases based on the time when the number of jitted lines stabilizes for all the runs of the two microbenchmark suites. We compared the response time between the two phases for each run. Table 4.2 shows the results.

Table 4.2 shows almost all the programs/scenarios (except one) for both microbenchmark suites exhibit better performance during the warmed up phase. However, in the PyPy benchmark suite, the *ai* program is not showing significant performance improvement. This is because at the end of the the first iteration while running the *ai* program, the majority of the code jitting process has already been completed. Only a few lines from the test library, which does test setup, got jitted during the benchmarking process (at the 34th iteration). These additional jitted lines have no impact on the actual performance of the benchmark program. Thus, the performance differences between the two phases are very small.

Table 4.2: Comparing the response time between the warmup phase (A) and the warmed up phase (B).

Performance Difference	**# of scenarios in the PyPy benchmarks**	**# of scenarios in the TechEmpower benchmarks**
B < A	0	0
B ∼ A	1	0
B > A	6	6
Total # of scenarios	7	6

Findings: For most of the studied programs/scenarios, the performance in the warmed up phase is statistically much better than the warmup phase. This clearly highlights the huge performance gain contributed by the JIT compilations.

Implications: Only performance data from the warmed up phase can be representative of the performance of systems due to the big difference in performance between the warmup and the warmed up phase. Thus, performance analysts should be careful when conducting the analysis and focus on the data after the warmup phase. In addition, the warmup phase for each system should be as short as possible, due to its inferior performance. Existing techniques for speeding up the jitting processes [63, 47, 70] can be very useful in this aspect.

(RQ2) What is the performance impact by varying JIT configurations?

In RQ1, we have found that the system performance significantly improves after the warmup phase. Hence, the data during the warmup phase is normally discarded during the performance analysis phase for a benchmark or a performance test. In this RQ we only focus on the system performance after the warmup phase. We study the performance impact of various JIT configuration settings. In particular, we would like to (1) verify whether the system configured with the default configuration setting would yield the optimal performance amongst other configuration settings, and (2) measure the performance impact of the jitting process (a.k.a., comparing the performance against completely disabling the jitting process).

Experiment

Similar to RQ1, we still used the same two microbenchmark suites as our experimental subjects. However, instead of keeping the default JIT configuration setting, we varied the values for the following six JIT configuration parameters: *decay*, *function_threshold*, *threshold*, *loop_longevity*, *trace_eagerness*, *trace_limit*. Table 4.3 shows the detailed information about these six configuration parameters. We picked these six parameters because we think they are tunable and can have an impact on where and when certain regions of the source code will be jitted. Other pa-

Table 4.3: List of relevant PyPy's JIT configuration parameters and their information.

Parameter	**Range**	**Default**	**Descriptions**
decay	$[0, 1000]$	40	The amount that PyPy decrease the counters for each loop or function periodically
function_threshold	$(0, \infty)$	1619	Number of times a function must run for it to become traced from start
loop_longevity	$(0, \infty)$	1000	A parameter controlling how long loops will be kept before being freed
threshold	$(0, \infty)$	1039	Number of times a loop has to run for it to become hot
trace_eagerness	$(0, \infty)$	200	Number of times a guard has to fail before we start compiling a bridge
trace_limit	$[0, 16385]$	6000	Number of recorded operations before we abort tracing with ABORT_TOO_LONG

Table 4.4: The JIT configurations chosen for performance evaluation.

Group	Config	threshold	function_threshold	decay	trace_limit	trace_eagerness	loop_longevity
Default	X	1039	1619	40	6000	200	1000
	$\frac{1}{4}X$	260	405	10	1500	50	250
	$\frac{1}{2}X$	520	810	20	3000	100	500
Group1	X	1039	1619	40	6000	200	1000
	$2X$	2078	3238	80	12000	400	2000
	$4X$	4156	6476	160	12000	800	4000
	R_1	64	101	120	375	200	2000
	R_2	519	809	5	375	200	2000
Group2	R_3	519	101	20	1500	200	4000
	R_4	3117	1619	2	1500	25	2000
	R_5	259	4857	120	375	200	2000
Group 3	JIT Off	-	-	-	-	-	-

rameters like *enable_ops*, *inlining*, and *off* need to be kept as default to enable the jitting process; whereas other parameters: *vec*, *vec_all*, *vec_cost* are not included in our study, as they are not relevant to the selected microbenchmark suites. Since there can be many possible combinations of these parameter settings, due to time constraints, we decided to run the two microbenchmark suites under the following eleven configuration settings from three different groups:

1. *Group 1 (Varying Default Configurations)* consists of five configuration settings ($\frac{1}{4}X$, $\frac{1}{2}X$, X, $2X$, and $4X$) by mutating the default configuration setting.

X refers to the default configuration setting. $2X$ means doubling the default configuration values, whereas $\frac{1}{2}X$ means cutting the default configuration values by half and rounding to the nearest integer values. As shown in Table 4.4, the default PyPy JIT configuration setting, X, is: $(1039, 1619, 40, 6000, 200, 1000)$, which corresponds to the configuration parameters *(threshold, function_threshold, decay, trace_limit, trace_eagerness, loop_loogevity)*. Hence, the $2X$ setting would be: $(2078, 3238, 80, 12000, 400, 2000)$ and the $\frac{1}{2}X$ setting would be: (519, 809, 20, 3000, 100, 500). To avoid PyPy command line parsing errors, when the value of parameter *trace_limit* exceeds the upper bound, we just set it to be two times of the default value (12000).

2. *Group 2 (Randomly Generated Configurations)* consists of five randomly generated configuration settings (R_1, R_2, R_3, R_4, and R_5). For each parameter in the configuration setting, we randomly generated an integer value within the defined boundary. As we can see from Table 4.4, the randomly generated JIT configurations in Group 2 are very different from the JIT configurations from Group 1.

3. *Group 3 (JIT Off)* consists of only one configuration setting, which sets the parameter *off* to be true. This setting will completely disable the jitting process.

Similar to RQ1, to avoid the measurement errors and noise, we repeatedly executed each PyPy benchmark program for 30 times, and ran each TechEmpower benchmark scenario for two hours. We also collected the same kind of performance data (a.k.a., the resource utilization metrics, the JIT logs, and the benchmarking logs) for further analysis.

Data Analysis

For each experiment, we first parsed the JIT log snapshots. Based on the time when the number of jitted lines stabilizes, we divided the benchmark runs into the warmup and the warmed up phase. We extracted response time from the warmed up phase for further analysis. We used the same statistical analysis techniques as in RQ1 to compare the response time among all the runs. For each program inside the PyPy benchmark suite, we compared the performance between each pair of the JIT configuration settings and identify the best performing configuration setting. Similarly, we also located the best performing JIT configuration settings for each scenario inside the TechEmpower benchmark suite. Table 4.5 shows the results. There are ties when ranking the top performing configuration settings across different programs/scenarios. We noted with a "*" besides a configuration setting if it shares the first place with other configuration settings in any programs/scenarios.

As shown in Table 4.5, Only 3 out of the 7 programs from the PyPy benchmark

Table 4.5: Number of best performing programs/scenarios under each JIT configuration setting.

Settings	**# of best performing programs/scenarios**	
	PyPy Benchmark	TechEmpoer Benchmark
$\frac{1}{4}X$	1*	0
$\frac{1}{2}X$	0	1*
X	3*	6*
$2X$	2*	1*
$4X$	1*	0
R_1	0	0
R_2	0	0
R_3	2*	0
R_4	1	0
R_5	1	0
$JITOff$	0	0

suite, where the default configuration setting yields the best performance. Furthermore, there are many configuration settings which perform best for some PyPy benchmarks (e.g. R_4, R_5) or share the top performance with other configurations (e.g. $\frac{1}{4}X$, $4X$). For the TechEmpower benchmark suite, all of the scenarios perform the best (with one scenario tied with $2X$ and $\frac{1}{2}X$) under the default configuration setting.

To quantify the performance impact of the jitting process, we also compared the performance of different JIT configuration settings against the JIT off setting. For each JIT enabled configuration setting, we measured the number of programs/scenarios that perform worse ($<$), similar ($\sim$), or better ($>$) than the JIT off setting. Table 4.6 shows the result. The number of programs/scenarios whose performance under jit enabled configurations is worse or no different than jit off is marked as bold.

From Table 4.6, we can see that, among the PyPy benchmark suite, all the JIT enabled configuration settings perform better than the JIT off setting, except R_1 in which the performance of two PyPy benchmark programs is even worse than completely disabling the jitting process (a.k.a., JIT off)! Similarly, in the TechEmpower benchmark suite, R_1 is still the odd one, as none of its scenarios is better than the JIT off setting. In addition, only three of the TechEmpower scenarios under R_5 are better than the JIT off setting.

Table 4.6: Comparing the jitting performance against the configuration under JIT off. The number of programs/scenarios whose performance under jitted configuration is worse or no different than JIT off setting is highlighted in bold.

Configs	**PyPy Benchmark**			**TechEmpower Benchmark**		
	$<$	$\sim$	$>$	$<$	$\sim$	$>$
$\frac{1}{4}X$	0	0	7	0	0	6
$\frac{1}{2}X$	0	0	7	0	0	6
X	0	0	7	0	0	6
$2X$	0	0	7	0	0	6
$4X$	0	0	7	0	0	6
R_1	**2**	0	5	**5**	**1**	0
R_2	0	0	7	0	0	6
R_3	0	0	7	0	0	6
R_4	0	0	7	0	0	6
R_5	0	0	7	**2**	**1**	3

Findings: Programs/scenarios running under the default configuration setting do not necessarily yield the best performance when comparing to other configuration settings. The optimal JIT configuration setting can vary depending on the programs/scenarios. The performance of some of the JIT enabled configurations can be worse than turning JIT off.

Implications: PyPy's JIT configuration settings have a big impact on the system performance. It is important to find the optimal JIT configuration setting for each system to achieve the best performance.

(RQ3) Do systems containing more jitted lines yield better performance?

In RQ2, we have found that the default JIT configuration setting does not necessarily result in the optimal performance. Different JIT configuration settings would result in different portions of the code been jitted. However, does more jitted code always lead to better performance? In this RQ, we want to study the relationship between these two aspects.

Experiment

We used the same data from RQ2 and did not run any additional experiment for this RQ.

Data Analysis

We first selected the configuration setting that has the best performance for each program/scenario based on the results of RQ2. Then we also selected the configuration settings with the highest number of jitted lines. If there are two different configuration settings corresponding to the above two criteria, we further performed the WRS test and calculated the CD value between the performance data under those configuration settings.

Table 4.7 shows the effect size between the best performing and the most jitted configuration settings for each program/scenario. Since there can be ties in either category, we compared all pairs of configuration settings from the best performing category to the category of the largest portion of jitted code. We label $True$ at the third column, if there is at least one common configuration setting in both categories for one program/scenario. In 71% ($\frac{5}{7}$) of the programs in the PyPy benchmark suite and all the scenarios in the TechEmpower benchmark suite, the best performing configuration setting is different from the one that has the highest number of the jitted lines. When comparing the performance differences, we compared all the pairs of these configuration settings from the two categories. In the end, twelve of the programs/scenarios have a medium to large effect size differences. In other words, the results show that more jitted lines do not necessarily lead to better

Table 4.7: Comparison between the configuration settings yielded the best performance and the configuration settings resulted in the most jitted code.

PyPy Benchmark			TechEmpower Benchmark		
Programs	**Effect Size**	**Same**	**Scenarios**	**Effect Size**	**Same**
ai	large	False	db	large	False
bm_mako	trivial,medium	False	fortune	large	False
chaos	large	False	json	large	False
django	large	False	plaintext	large	False
html5lib	-	True	query	large	False
pidigits	large	True	update	large	False
rietveld	large	False			

performance. For the PyPy benchmark 'html5lib', we marked the effect sizes as '-', since the best performing configuration settings and the highest amount of the jitted code configuration settings are exactly the same. Hence, we do not calculate the effect sizes for this case.

Findings: JIT configuration settings, which resulted in the highest number of the jitted lines, do not necessarily yield the best performance.

Implications: We cannot just arbitrary choose the configuration settings that favor more jitted lines of code while tuning the system performance. A more sophisticated approach is needed to locate the optimal configuration setting(s) for one system.

4.4 Automatically Tuning the JIT Configuration Parameters

In the previous section, we have found that PyPy's JIT configuration settings do have a significant impact on the system performance. Furthermore, there is no straightforward way to recommend a performance-efficient JIT configuration setting, since such setting can be application-dependent and a higher portion of the jitted code does not necessarily result in better performance. Hence, in this section, we will propose our automated approach, ESM-MOGA (Effect Size Measurement-based Multi-Objective Genetic Algorithm), to tuning the JIT configuration parameters for one system.

Figure 4.3 provides an overview of our tool which we called the PyPyJITTuner. It leverages a search-based technique called Multi-Objective Genetic Algorithm (MOGA) [39] and a statical measure called effect size, for the exploration of the

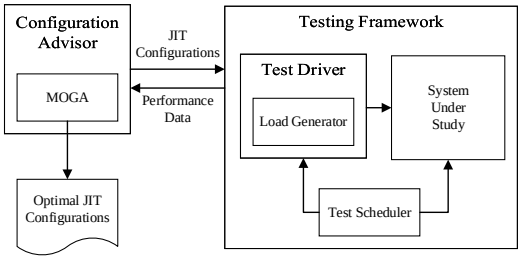

Figure 4.3: Overall process of PyPyJITTuner.

JIT configuration space. Genetic Algorithm (GA) is a search-based method inspired by evolutionary biology, in which a population of solutions is evolved during each generation. The solutions from the next generation should be generally better than the previous generations evaluated based on some objective functions. MOGA is a type of GA, in which multiple objectives are being considered. We chose MOGA, as there can be multiple objectives associated with a system's performance (e.g., optimizing the response time for multiple scenarios). One machine, which is deployed with the tailored-version of the MOGA, acts as the configuration advisor. When new solutions have been created, this machine continuously sends the JIT configuration settings (solutions) to the test scheduler machine, which will deploy and configure the system under study (SUS). Once the test scheduler machine receives these settings, it will reset the test environment (a.k.a., clean up the database, and remove the testing data from the previous run), and start up the SUS under the suggested JIT configuration setting. The same performance test (a.k.a., the same

workload) will be executed. Once the test is completed, the performance data will be collected and sent to the configuration advisor machine for further analysis. The configuration advisor machine will evaluate the newly received performance data against the data from other configuration settings and leverage the MOGA methods to select the best solutions and generate the next generation. If the solutions in the next generation are good enough, the MOGA will stop the evolution and output one or multiple "optimal" configuration settings. Otherwise, the MOGA will continue with another round of iteration. The newly generated JIT configuration settings will be sent to the test scheduler machine for another round of testing.

The rest of this section is organized as follows: Section 4.4.1 explains the general idea behind the ESM-MOGA approach. Section 4.4.2 presents our performance testing framework, and Section 4.4.3 describes briefly our implementation.

4.4.1 Tailoring MOGA for JIT Configuration Tuning

GA is a search-based method inspired by evolutionary biology. GA encodes the candidate solutions into a set of values, called "chromosomes". Inside the chromosomes, the set of values, which are to be optimized are called the "genes". GA starts off with a population of the initial solutions and keeps iterating until any of the termination criteria is met. During each iteration, GA improves the population via crossover (combining existing solutions to produce new solutions), mutation

(randomly changing some values in the solutions), and selection (picking the best candidate solutions). The termination criteria can either be the optimization conditions (e.g., the resulting solutions are better than a predefined threshold) or the maximum number of iterations. The MOGA, which is a type of GA, evaluates multiple objectives simultaneously. In general, as illustrated in Figure 4.4, the MOGA consists of six phases: the problem formulation phase, the initialization phase, the tournament phase, the evolution phase, the selection phase, and the stopping phase. The process of going through the tournament, the evolution, and the selection phase can be repeated multiple times, with each iteration called one generation. At the end of each generation, a new population will be produced. This process will be repeated until any termination criteria described in the stopping phase is met.

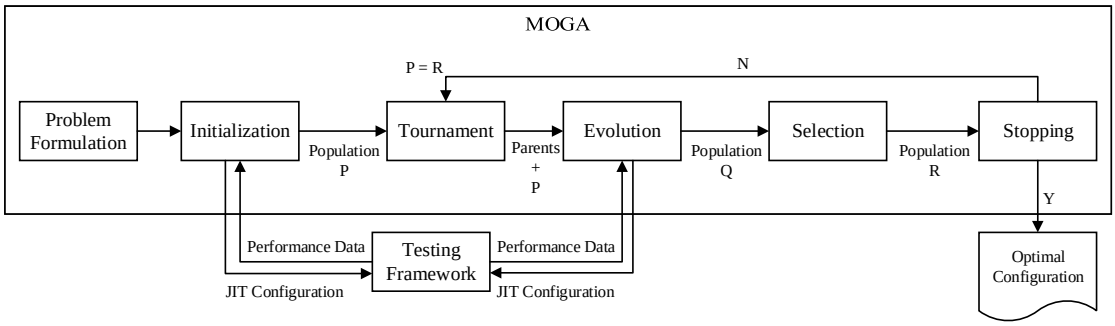

Figure 4.4: The workflow for our tailored version of the MOGA method.

In this subsection, we will explain the ESM-MOGA approach by using a running example. For illustration purposes, we assume the SUS in our running example is

a simple e-commerce system, which consists of only three scenarios: login, browse, and purchase.

4.4.1.1 Phase 0 - Problem Formulation

We formulated our problem of automated tuning of JIT configuration settings into a multi-objective optimization problem. Our objective is to find one or more optimal JIT configuration settings that yield the best performance in all the scenarios in the system. Below we define our solution encoding and objectives:

- **Solution Encoding**: The ESM-MOGA requires us to encode its solution into binary strings. As shown in Section 4.2.2, all the studied JIT configuration parameters are integers and have a large range (a.k.a., many possible values). Assume we use 2^{32} as the upper bound for unbounded parameters, we have to do 5.5e+45 performance tests if we want to try out all combinations which is apparently impossible.

  Hence, we decided to select eight representative values from the input domain of each configuration parameter: ($4X$, $3X$, $2X$, X, $\frac{1}{2}X$, $\frac{1}{4}X$, $\frac{1}{8}X$, $\frac{1}{16}X$), where X refers to the default value for that configuration parameter. We chose the above eight levels, as these eight values cover a wide range of the input domain and each configuration parameter value can be easily encoded into a binary string of length three. In this way, the smallest configuration value ($\frac{1}{16}X$),

the default configuration value (X), and the largest configuration value ($4X$) for each parameter are encoded as 111, 011, and 000, respectively. We set the largest configuration values to be $4X$, as Section 4.3 shows that large JIT configuration settings usually do not yield good performance. We set the smallest configuration values to be $\frac{1}{16}X$, as Section 4.3 shows very small configuration settings could result in a high number of jitted code, but worse performance. To ease explanation, in our running example, there are only two configuration parameters. Hence, the default configuration setting can be encoded as a binary string: 011011.

- **Objectives**: For a real world system, there can be more than one aspect associated with the performance of the system. Examples of optimizing performance aspects can be optimizing the resource utilizations (e.g., CPU, memory, and disk) or the responsiveness of different scenarios in a system. Some of these concerns can be conflicting with each other. In our approach, we focus on optimizing the response time for different scenarios in a system. Each objective refers to a list of response time for each scenario during the warmed up phase, measured through performance testing. In our running example, the objectives are to optimize the response time for the above three scenarios in the e-commerce system.

4.4.1.2 Phase 1 - Initialization

During the initialization phase, the ESM-MOGA defines an initial population (P) consisting of n solutions. In our approach, P consists of the default configuration setting, and $n-1$ randomly generated configuration settings. The ESM-MOGA will intentionally included the default configuration setting in the initial population, as we want to ensure the default configuration setting is evaluated among its alternatives and the final "optimal" setting(s) will be at least as good or better than the readily available default configuration setting. Once the initialization process is completed, the ESM-MOGA enters the iterative process of going through the tournament, the evolution, and the selection phase to refine and improve its population until any termination criteria is met.

To ease explanation, we set $n = 4$ in our running example and compose the initial population (P) with the following solutions:

$$\mathrm{P} = \begin{cases} C_1: & 011011, \\ C_2: & 001010, \\ C_3: & 000101, \\ C_4: & 101110 \end{cases}$$

Once the initial population is generated, the solutions in the initial population will be sent to the test scheduler machine in Section 4.4.2. Multiple performance

tests with the same workload will be conducted under each given configuration setting. The response time of the three scenarios under warmed up phase will be collected as performance data and assigned as the objectives for each solution.

4.4.1.3 Phase 2 - Tournament

During the tournament phase, the ESM-MOGA will first randomly select two solutions from a pool which contains all solutions of the current population. Then, a pairwise comparison is done to recognize the better solution from the two. The better solution will be used as one of the parents for the next phase. These evaluated solutions will not be put back to the pool for efficient concerns. The process will be repeated until all solutions in the pool have been evaluated pairwisely. In our approach, the pairwise comparison is done using a pre-defined dominant comparison function and the dominating configuration setting (a.k.a., the better solution) will be selected. In this dominant comparison function, the configuration setting A dominates the configuration setting B, if the response time distributions under the two configuration settings satisfy the following two criteria:

1. **The response time for all the scenarios under A are statistically no worse than under B**: The response time under configuration A for one scenario is statistically no worse than under B, if (1) the response time distributions for that scenario under the two settings are not statistically signifi-

cantly different under the WKS test, or (2) they are statistically significantly different under the WKS test, but there is only a trivial to small effect size calculated by the CD. This relation has to be held for all the scenarios when comparing the two settings.

2. **There is at least one scenario whose response time under A is statistically better than under B**: The response time under configuration A for one scenario is statistically better than B, if the response time distributions for that scenario under the two settings are statistically significantly different under the WKS test and there is a medium to large effect size calculated by the CD.

In other words, one configuration setting (A) only dominates the other one (B), if (1) the performance of all the scenarios under A is at least as good as B, and (2) there will be at least one scenario under A whose performance is better than B.

The dominance comparison among all the pairs of the configuration settings are shown in Table 4.8. Each row in Table 4.8 corresponds to the comparison results of one configuration setting pair. For example, the second row shows the comparison results between configuration setting C_1 and C_3. The response time for the login scenario is statistically better under C_1 than C_3. The performance of the other two scenarios are statistically not different between the two settings. Hence, the

Table 4.8: Dominance relations among the four configuration settings in our running example. "$\succ$" means the the left configuration setting dominates the right configuration setting, "$\prec$" means the right configuration setting dominates, and "$\approx$" means there is no dominance relation.

Config Pairs	**Login**	**Browse**	**Purchase**	**Dominance**
(C_1, C_2)	Better	Worse	Better	$\approx$
(C_1, C_3)	Better	Equal	Equal	$\succ$
(C_1, C_4)	Better	Better	Equal	$\succ$
(C_2, C_3)	Better	Equal	Equal	$\succ$
(C_2, C_4)	Better	Better	Equal	$\succ$
(C_3, C_4)	Equal	Worse	Worse	$\prec$

configuration setting C_1 dominates C_3. Assume, from the pool that contains all solutions of population (P), we selected (C_1 and C_3), (C_2 and C_4) for pairwise comparison. The two configuration settings, C_1 and C_2, will be selected as parents for the next phase.

4.4.1.4 Phase 3 - Evolution

During the evolution phase, the parents from the Tournament phase will undergo the following two actions to produce new solutions:

- **Crossover**: Two solutions from the Parents are randomly selected as parents. A new solution will be created by randomly selecting some bits from one solution and the remaining bits from another solution. In our running example, C_1 and C_2 will be selected as parents. A new solution C_5 (011010) can be created by inheriting the first three bits from C_1 and the remaining bits from C_2.

- **Mutation**: Some of the newly produced solutions will be mutated by randomly flipping some bits (a.k.a., turning 0s into 1s and 1s into 0s). For our running example, after flipping the first and the last bits of C_5, it becomes 111011.

Similar as the Initialization phase, a performance test with the same workload but a new configuration setting (C_5) will be conducted. Once completed, the performance data will be sent back as objectives for the new configuration setting. The overall population (Q) at the end of this phase will consist of the new solutions produced after the crossover and the mutation operations as well as existing solutions from P.

4.4.1.5 Phase 4 - Selection

In this phase, the "best" n solutions in Q will be selected with NSGA-II selection [39] [5]. It will first use the non-dominated sorting algorithm to sort the solutions into different levels (L_0, L_1, ...). Solutions that were dominated by the smallest number of solutions will be assigned to the top level (L_0). Solutions in L_0 are the "best" solutions in this iteration, followed by the solutions in L_1, and then L_2, and so on. The solutions within the same levels (e.g., L_0) are not dominant over each other. For example, if configurations settings A and B are both within L_0, it means that A and B are not dominant over each other. In other words, some scenarios are better performed under A and whereas some other scenarios are better performed under B. When selecting the top n solutions, we will first start picking solutions from the top level (L_0), followed by solutions from L_1, and so on. If there are more solutions in a level than we needed (a.k.a., exceeding the total n solutions), we will rank solutions within that level with crowding distance sorting and select the top ranked solutions in that level.

Suppose for our running example, after sorting the solutions in Q using the non-dominated sorting algorithm, they are divided into the following levels:

$$\mathrm{Q} = \begin{cases} L_0 : & C_5, \\ L_1 : & C_1, C_2, \\ L_2 : & C_4 \\ L_3 : & C_3 \end{cases}$$

Since at the end of the selection phase, only n solutions will be kept. Hence, our resulting population (R) will be C_1, C_2, C_4, C_5.

4.4.1.6 Phase 5 - Stopping

During this phase, the resulting population Q formed during this generation will be evaluated to decide whether its solutions are good enough comparing to the previous generation. The main idea is to decide whether any progress has been made during this generation. In other words, we want to check whether there are any better solutions produced during this generation. We used the Mutual Dominance Rate (MDR) [72] to measure the improvements made between the current population B and the population A from the previous generation:

$MDR(B, A) = \frac{dom(B,A)}{\|B\|} - \frac{dom(A,B)}{\|A\|}$,

where $dom(A, B)$ is defined as the number of solutions in population A that are dominated by at least one solutions in B. Hence, for our running example, $dom(P, R)$ would be 4, since all four solutions in P are dominated by at least one

solutions in R. $dom(R, P)$ would be 1, since only C_4 in R is dominated by the solutions in P. Hence, $MDR(R, P) = \frac{1}{4} - \frac{4}{4} = -\frac{3}{4}$. The closer the MDR value gets to -1, the larger the improvement has been made in the current generation. If MDR is close to 0, it means little progress has been made to the population. The iteration should be stopped if the improvement between two generations is insignificant (a.k.a., $|MDR|$ is smaller than some threshold values), or it has been running for too long (e.g., over 100 iterations).

When the termination criterion is met, we will output the top configuration settings for the current generation with the NSGA-II selection. Since there is only one solution (C_5) at L_0 for our running example, C_5 will be the optimal configuration setting outputted.

4.4.2 Our Performance Testing Framework

For any newly generated solutions, we need to measure their performance using a performance test. Each solution, which is sent to the test scheduler machine inside the Testing Framework, will first be parsed into the corresponding JIT configuration setting. The test scheduler machine will then start the system with the new configuration setting and measure the system performance under a predefined workload. At the end of each performance test, performance data during the warmed up phase will be collected and sent to the configuration advisor machine, so that

they can be used as objectives to evaluate among solutions in the ESM-MOGA.

For our running example, a total of five performance tests with the same workload but different JIT configuration settings, which correspond to the four initial solutions in P and the new solution in Q, will be run. Once each test is completed, the test scheduler shuts down the SUS, collects the performance data (response time for the individual scenarios, the resource utilization metrics, and JIT logs) and sends the data to the configuration advisor machine.

4.4.3 Implementation

We implemented the ESM-MOGA using the NSGA-II algorithm [39], which is a fast and efficient multi-objective genetic algorithm, from the DEAP framework (Distributed Evolutionary Algorithms in Python) [3]. The framework contains the relevant library functions for NSGA-II, like assigning crowding distance, non-dominated sorting algorithm, and NSGA-II selection. We had to implement the dominance function, and input encoding ourselves to fit into NSGA-II algorithm. We re-implemented the non-dominate sorting and NSGA-II selection functions so that they can use our dominance function to compare among solutions.

We also implemented the automated performance testing framework, which leverages JMeter [1] as the load generator. The performance testing framework can startup, initialize, execute, and stop a performance test under one particular

Table 4.9: An overview of the three Python-based systems under study.

Name	Version	LOC	Application Domain	Technology Stack
Saleor	2017.07.0	48482	E-commerce	Gunicorn,Tornado,Postgres,Django
Wagtail	1.12.1	85006	Content Management	Gunicorn,Tornado,Postgres,Django
Quokka	0.2.1	34468	Content Management	Gunicorn,Tornado,MongoDB,Flask

configuration setting.

4.5 Case Study

In this section, we evaluated the performance of our automated approach to tuning the JIT configuration parameters on three Python-based open source systems: Saleor [17], Wagtail [24], and Quokka [15]. As shown in Table 4.9, these three systems vary from system sizes, application domains, and technology stacks. All three systems can be deployed on top of the Tornado WSGI server which uses Gunicorn for worker process management. And they all require a database to be functional. Saleor is an e-commerce system, built using the Django framework, and uses Postgres as its database. Wagtail shares the same technology stack as Saleor (a.k.a., Django and Postgres), but is from a different application domain: the Content Management System (CMS). Although Quokka is also a CMS, it is built with the

Flask framework and uses MongoDB as its database.

The rest of this section is organized as follows. Section 4.5.1 describes the case study setup. Section 4.5.2 explains the case study results.

4.5.1 Case Study Setup

We deployed the above three systems on the same physical machines, which have the following hardware configurations: *Intel i7-4790* CPU, 16 GB memory, and 2 TB hard-drive. JMeter was deployed on another physical machine with the following hardware configuration: *Intel(R) Core(TM)2 Duo* CPU, 4 GB memory, 160 GB hard-drive. And all machines have Ubuntu 14.04 deployed. The reason for the separate deployment of JMeter and the SUSs is to ensure no overhead caused by the load generator to the SUSs [64]. The version of the PyPy that we used for evaluation is *5.7.1* which corresponds to Python version *2.7.13*. Similar to the exploratory study, we focused on the same six JIT configuration parameters. Hence, each solution (a.k.a., configuration setting) requires 18 bits to be encoded into our tailored MOGA method. For example, the default configuration setting would be encoded as 011011011011011011. As for the rest of the MOGA configurations, we set the initial population (P) size as 40, and the mutation rate as 0.10 based on some small trials. We also configured our termination criteria to be either $|MDR| \leq 0.1$ holds for two consecutive generations or the MOGA has iterated for 10 generations.

Table 4.10: Workload description for the three systems.

System	Workload Mix	Workload Intensity
Saleor	load index page	10 req/sec
	load login page	
	login request	
	view category	
	view product	
	add product to cart	
	view cart	
	check out cart	
	select shipping method	
	select shipping address	
	payment	
	payment confirm	
Wagtail	add blog(1)	
	add event(1)	
	edit blog(2)	
	view blog page(3)	
	view event page(3)	
Quokka	add blog(1)	
	edit blog(2)	
	view blog(7)	

Table 4.10 shows the workload that we have set for the three systems. The workload tries to simulate how real users use the systems in the field. The overall workload intensity (10 requests/sec) is the same for all three systems and the workload mix for each system is shown below.

- For the e-commerce system, Saleor, the workload tries to mimic the purchasing workflow from a real customer. Hence, we divided this scenario into 12 actions, which correspond to 12 different webpage operations. The overall workload intensity (10 request/sec) corresponds to 10 different users performing the above 12 actions at the same time. Hence, all the actions in this workload are assigned with the same ratio in the workload mix.

- For the content management system, Wagtail, the workload tries to mimic users reading, posting, and editing blogs or events. Since, in the majority of the time, users will be viewing the blogs or events, we assigned a higher ratio for these two actions. The scenarios of adding a new blog or a new event happen the least frequently. Hence, they are assigned with the smallest weight in the workload mix.

- Quokka's workload is similar to Wagtail's, as they are in the same application domain. Since Quokka only supports reading/editing/adding blogs, we adjusted the workload mix accordingly.

As Section 4.3 indicates, only the performance data from the warmed up phase is representative of the actual system performance. Hence, in the case study, we want to make sure the system has been running long enough (a.k.a., finished the warmup up phase). To properly decide the test duration, we first did a test run with the default configuration setting, in which the predefined workload was executed for three hours. We leveraged a similar technique as Alghmadi et al. [29] to test when the system's performance behavior gets repetitive, so that we can identify the duration of the warmup phase. We divided the collected performance data into intervals of every 20 minutes. Then we performed statistical analyses with the WRS test and CD values on the response time between two adjacent time intervals. We considered the system to be fully warmed up when the response time from the two adjacent time intervals show insignificant difference for all scenarios (a.k.a., not statistically different by the WRS test or CD values show trivial to small effect sizes).

We performed the above process in all three case study systems. We found that all three systems finish the warmup phase in the first 40 minutes before their performance behaviors start to be repetitive. Hence, for consistency concerns, we set the test duration to be 50 minutes for each test and only used the data from the last 10 minutes (a.k.a., the data from the warmed up phase) for further analysis in the ESM-MOGA method.

Table 4.11: Statistics after running the MOGA approach on the three case study systems.

System	**# of Generations**	**# of Configurations Evaluated**	**Duration (hour)**
Saleor	3	100	36
Wagtail	7	199	67
Quokka	3	96	35

4.5.2 Case Study Results

For all the three systems, we applied our ESM-MOGA method with the aforementioned setup. Table 4.11 shows the runtime statistics for the ESM-MOGA method after running on the three systems. The search algorithm all terminates under termination criteria $|MDR| \leq 0.1$. For Saleor, it takes 36 hours and evaluated 100 solutions, 67 hours and 199 solutions for Wagtail, and 35 hours and 96 configuration settings for Quokka. For all three systems, the ESM-MOGA method found optimal solutions when it terminated.

For each system, we used the NSGA-II selection to select the top three configuration settings. We compared the response time and the resource utilization between the optimal configuration settings and the default configuration setting.

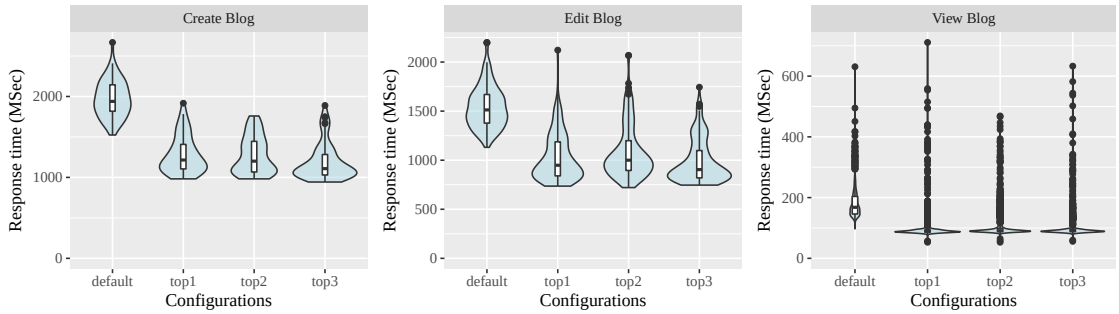

Figure 4.5: Visualizing the response time distributions of different scenarios under different configuration settings for Wagtail.

Table 4.12: Comparing the performance between the optimal configuration settings and the default configuration setting for the three systems. The optimal configurations are labelled as O_A, O_B, and O_C. WS stands for the web server, and DB stands for the database. "-" means the ESM-MOGA suggested optimal configuration setting outperforms the default setting and "+" means otherwise.

System	Top Configurations	Comparing scenarios			Average differences (%)		Average resource usage difference (%)			
		better	equal	worse	min	max	WS_CPU	WS_Memory	DB_CPU	DB_Memory
Saleor	O_A	12	0	0	-27.11	-56.17	-33.71	+24.80	-3.61	-0.24
	O_B	12	0	0	-27.32	-58.93	-25.68	+31.97	-7.59	+1.18
	O_C	12	0	0	-9.66	-60.28	-32.38	+12.74	-7.30	+0.84
Wagtail	O_A	5	0	0	-33.47	-45.10	-23.86	+158.71	-5.96	+1.55
	O_B	5	0	0	-29.59	-44.93	-18.81	+202.53	-2.90	-2.04
	O_C	5	0	0	-35.93	-44.28	-25.18	+157.94	-3.46	-3.34
Quokka	O_A	2	1	0	-9.39	-34.95	-15.39	+59.34	-13.80	-5.5688
	O_B	3	0	0	-13.45	-22.71	-16.51	+61.08	-21.89	+0.942
	O_C	1	2	0	-4.98	-25.44	-11.94	+62.03	-19.05	+2.21

Figure 4.5 visually compares the response time distributions between the default and the top three optimal configuration settings for Wagtail. Due to space limitations, it only contains the performance comparisons of the three blog-related scenarios inside Wagtail. Each sub-figure corresponds to one scenario. Within each sub-figure, the four violin plots correspond to the response time distributions of that scenario under the default and the three optimal configuration settings. Among all the sub-figures, the response time under the default configuration setting is significantly much higher than the optimal configuration settings. A similar trend also holds for the two event-related scenarios in Wagtail.

To quantify the differences between the optimal and the default configuration settings, we performed the WRS test and CD test between the response time distributions under each configuration pair. We used the same criteria as in Section 4.3 to judge whether one response time distribution is better, or same, or worse than the other one. Table 4.12 shows the results. Each row corresponds to one optimal configuration setting for one particular system. There are no orderings among the top three optimal configurations (O_A, O_B, and O_C). The second to the fourth columns contain the number of scenarios which show better, equal, or worse difference when comparing this configuration setting against the default. For Saleor and Wagtail, all the scenarios performed better under the suggested optimal configuration settings. For Quokka, at least one scenarios performed better under the

suggested optimal configuration settings, while the remaining scenarios performed no worse than the default configuration setting. The fifth and the sixth columns show the minimum and maximum percentage of differences when comparing the average response time under the suggested configuration setting with the average response time under the default for all scenarios. The percentage improvement in terms of the average response time can vary between 5% to 60%.

Since each system consists of a web server and a database, we further compared the CPU and the memory utilizations between the optimal and the default configuration settings. Last four columns in Table 4.12 shows the comparison results for the resource utilizations. The CPU usage for both components drops. The decrease in CPU is more significant in the web server, with the average improvement ranges between 12% to 33.7%. However, the memory usage for the web server dramatically increases (12.7% to 202.5%) across all the optimal configuration settings. We suspect this may be due to the storage of the complied jitted code. For a more detailed discussion, please refer to Section 4.6.4.

4.6 Discussions

In this section, we discussed the findings based on the case study results and their implications.

4.6.1 Optimal Configurations across Different Systems

As we can see from the previous section (Section 4.5), the optimal configuration settings obtained using ESM-MOGA significantly out-performed the default configuration. In this subsection, we would like to compare the optimal configuration settings against the default configuration setting in order to see if we can derive some rules or provide some guidance for PyPy users when tuning the JIT configuration settings for their systems.

For each of the studied systems above, we obtain its top three optimal configuration settings, whose values are shown in Table 4.13. We also included the default configuration setting in the table to ease comparison.

One common pattern as we can see from Table 4.13 is that *trace_eagerness* is significantly smaller in all the optimal configuration settings when comparing to the default ones. *trace_eagerness* refers to the eagerness to compile a non-jitted branch within a loop. A system can go through various branches within a loop. A smaller *trace_eagerness* is preferred, so that the branch(es) corresponds to frequently executed scenarios will be jitted faster. There is no pattern found in the other five configuration parameters, as they can be either bigger or smaller than the default values among the nine optimal configuration settings.

We are also interested in understanding the correlation of JIT configuration

Table 4.13: Top three optimal configuration settings for the three studied system.

System	Config.	decay	function_threshold	loop_longevity	threshold	trace_eagerness	trace_limit
Saleor	O_A	10	101	2000	129	25	12000
	O_B	2	1619	250	1039	12	12000
	O_C	2	1619	1000	3117	100	12000
Wagtail	O_A	10	101	1000	1039	12	12000
	O_B	2	404	250	259	12	12000
	O_C	2	101	4000	4156	50	12000
Quokka	O_A	80	404	500	519	12	12000
	O_B	2	3238	2000	4156	12	6000
	O_C	2	202	4000	4156	25	3000
Default	-	40	1619	1000	1039	200	6000

Table 4.14: Spearman correlation between configuration and response time. The large and very large correlation measures are shown in bold.

System	Correlation	decay	function_threshold	loop_longevity	threshold	trace_eagerness	trace_limit
Saleor	corr. coeff.	-0.5120	-0.1338	0.0291	0.1869	-0.2961	0.6293
	$p-value$	6.057e-08	0.1864	0.7748	0.0638	0.0029	3.028e-12
	scale	**large**	small	trivial	small	small	**large**
Wagtail	corr. coeff.	-0.3719	-0.2750	0.1506	0.1392	-0.5446	0.5790
	$p-value$	6.346e-08	8.452e-05	0.0336	0.0498	2.2e-16	2.2e-16
	scale	moderate	small	small	small	**large**	**large**
Quokka	corr. coeff.	-0.0403	-0.1932	0.1157	0.0568	0.0506	0.8024
	$p-value$	0.7554	0.1323	0.3703	0.6608	0.6956	4.476e-15
	scale	trivial	small	small	trivial	trivial	**very large**

parameters to the system performance. We collected all the evaluated configuration settings and the corresponding response time for different scenarios. We summed up the average response time for all scenarios as the overall response time under a JIT configuration setting. Then we calculated the Spearman's rank correlation between each configuration parameter and the overall response time. Spearman is a non-parametric correlation metric measuring the strength of the relation between the two variables. The scale of the Spearman's ρ correlation coefficient is indicated below [57]:

$$\rho = \begin{cases} \text{trivial} & \text{if } 0 \leq \rho < 0.1 \\ \text{small} & \text{if } 0.1 \leq \rho < 0.3 \\ \text{moderate} & \text{if } 0.3 \leq \rho < 0.5 \\ \text{large} & \text{if } 0.5 \leq \rho < 0.7 \\ \text{very large} & \text{if } 0.7 \leq \rho < 0.9 \\ \text{near perfect} & \text{if } 0.9 \leq \rho \leq 1.0 \end{cases}$$

Table 4.14 shows the result of the correlation between each configuration parameter and the overall system performance. We highlight the cell in bold if the correlation measure is "large" or "very large". Each system has at least one configuration parameter which has a "large" or "very large" correlation measure. However, highly correlated configuration parameters vary among the three systems, ex-

cept *trace_limit*. A large *trace_limit* enables the system to compile large frequently executed loops, which can subsequently improve the system performance. Meanwhile, the three JIT configuration parameters *function_threshold*, *loop_longevity* and *threshold* have very low correlations (small or trival) with the overall system performance.

Findings: The configuration parameter *trace_eagerness* should be generally set lower than the default values in order to obtain better performance. *trace_limit* is highly correlated with the overall system performance, whereas *function_threshold*, *threshold* and *loop_longevity* have no or weak correlations.

Implications: There are some general guidance in terms of tuning the PyPy JIT configuration settings on web frameworks. However, the optimal configuration settings are still highly system dependent. In this paper, we only evaluated the impact of PyPy JIT configuration settings on benchmark programs or web applications. However, Python is also popular in data statistic analysis and machine learning (e.g. scipy, tensorflow), which could be time consuming to train a model. One of the interesting future research area would be to derive rules or general guidance to improve the performance of various machine learning or statistic analysis packages by tuning their JIT configuration parameters.

4.6.2 Top Configurations across Different Workloads

In the previous study, we compared the optimal JIT configuration settings and it's performance cross different applications under the same level of workload. In this subsection, we want to compare the optimal JIT configuration settings under different workloads. We selected Wagtail as our experiment subject.

In addition to the default Wagtail workload (10 req/sec), we generated two other workloads for comparison: 15 req/sec and 5 req/sec, while keeping the workload mixes. For each of newly generated workload, we ran PyPyJITTuner to derive the optimal JIT configuration settings. For 15 req/sec workload, the framework iterated for 5 generations before termination. And it takes 4 generations for the PyPyJITTuner to be terminated under 5 req/sec workload. The resulting configuration settings yield significant performance gain (20% - 50%) when compared to the default configuration setting.

Table 4.15 shows the actual performance for the top three configuration settings under each workload. It shows the average response time for each scenario in milliseconds, as well as various resource usage metrics like CPU and memory usage for the web server and the database, respectively. In addition, it also shows the number of jitted lines under each configuration setting. Although the workloads are different, all the nine top optimal configuration settings share similar perfor-

Table 4.15: Comparing the performance among the optimal configuration settings under different workloads for Wagtail.

Experiments	Workload	**5 req/sec**			**10 req/sec**			**15 req/sec**		
	Top configs	(O_A	O_B	O_C)	(O_A	O_B	O_C)	(O_A	O_B	O_C)
Response time per scenario (msec)	add blog	1316.00	1286.00	1237.00	1276.00	1275.00	1191.00	1379.00	1428.00	1366.00
	add event	1104.00	1073.00	1090.00	1033.00	1190.00	1133.00	1326.00	1409.00	1375.00
	edit blog	1077.00	1068.00	920.00	1025.00	1085.00	987.40	1080.00	1143.00	1100.00
	view blog	102.50	104.60	111.00	101.30	109.00	104.90	119.00	115.40	117.10
	view event	89.55	94.81	95.03	89.98	89.06	90.10	98.10	101.0	98.17
Resource usage	WS_CPU (%)	27.89	25.62	23.82	46.16	48.04	44.27	61.63	62.19	63.27
	WS_Memory (MB)	2532.00	2229.00	1848.00	3337.00	3770.00	3214.00	2998.00	3123.00	2944.00
	DB_CPU (%)	0.21	0.22	0.21	0.41	0.40	0.40	0.61	0.59	0.59
	DB_Memory (MB)	87.49	87.85	86.96	90.58	93.90	95.09	97.84	97.90	95.20
JIT	# of jitted lines	12987	9993	8780	11045	13065	12298	11002	11451	11402

mance in terms of response time for each scenario. The number of jitted lines are similar across different workloads. As workload increases, the CPU and memory consumption for both the web server and the database increases. Hence, in this case, the workload intensity is the main reason behind the increase of memory and CPU consumption of the two server components.

Table 4.16 shows the top three optimal JIT configuration settings under different workloads. The optimal JIT configuration settings under different workloads share very similar properties (e.g., large *trace_limit* and small *decay* and *trace_eagerness* values).

Table 4.16: Top three optimal configuration settings for Wagtail under different workloads.

Workload	decay	function_threshold	loop_longevity	threshold	trace_eagerness	trace_limit
	2	404	250	129	12	12000
5 req/sec	40	101	250	519	12	12000
	20	404	500	1039	25	12000
	10	101	1000	1039	12	12000
10 req/sec	2	404	250	259	12	12000
	2	101	4000	4156	50	12000
	2	1619	125	3117	25	12000
15 req/sec	2	1619	62	2078	25	12000
	10	101	62	2078	25	12000
Default	1039	1619	40	6000	200	1000

Findings: Varying the workload intensity would not impact the optimized JIT performances. And different workloads would result in different optimal configuration values. However, there are some common properties (e.g., large *trace_limit* and small *decay* and *trace_eagerness* values) shared across them.

Implications: Researchers could further improve the efficiency of the ESM-MOGA by developing machine learning algorithms to proactively eliminate some of the performance deficient configuration settings from each generation.

4.6.3 Code Jitting vs. Performance

In Section 4.3, we have shown that more jitted lines do not necessarily lead to better performance. Furthermore, the performance under some of the JIT configuration settings are even worse than turning the JIT completely off! In this subsection, we would like to perform a more in-depth study to find out the reasons.

In Figure 4.6, we plotted the number of jitted lines with respect to system performance across all the runs we did for the Saleor system. And the red dotted line shows the overall average response time under the JIT off configuration setting. As the number of jitted lines increases, the average response time for the system gradually decreases. As we can see from the figure, there are a few JIT configuration settings which are even worse than turning the JIT off! We conducted further analysis to understand the reason why some jitted code would even lead to worse

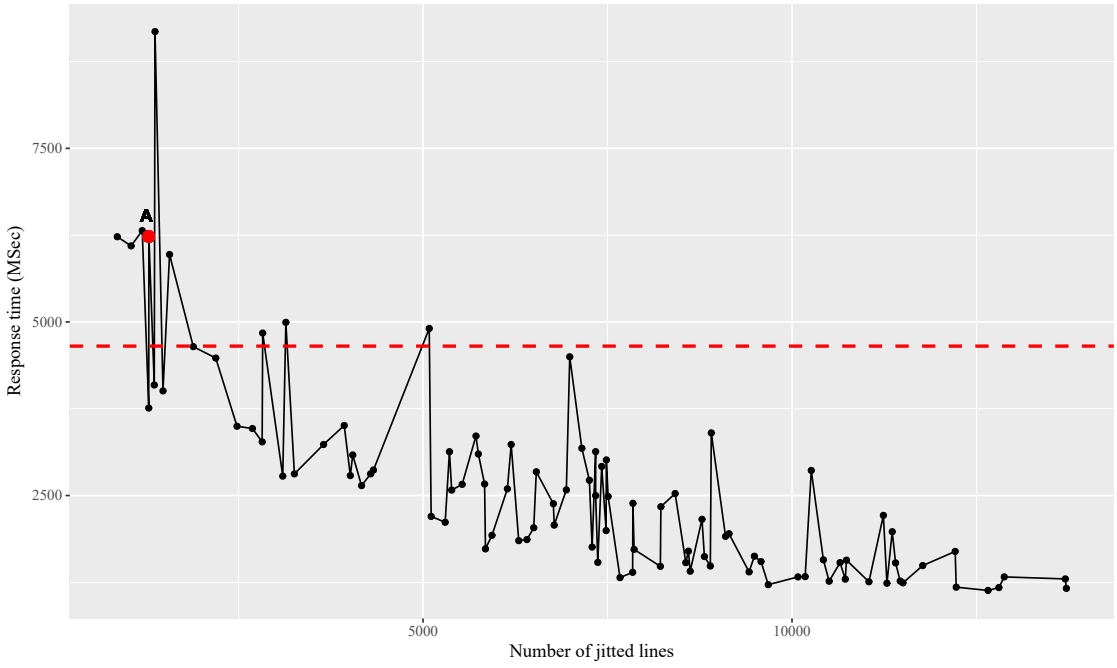

Figure 4.6: Number of jitted lines and overall average response time among all evaluated JIT configuration settings in Saleor. The red dotted line shows the overall average response time with JIT turned off.

performance.

We focused on comparing the code structure under two configuration settings: configuration A, which is a configuration with JIT enabled. As shown in Figure 4.6, the performance of configuration A is worse than turning JIT off. For brevity, we call the JIT off configuration as configuration B.

We first applied cProfile [21] to gather the high-level performance numbers for Saleor. cProfile is a profiling tool for Python-based systems. It can provide information like the execution time, the number of execution for each of the executed functions. We enabled cProfile and ran the Saleor under the default workload twice: one run under configuration A, and the other run under configuration B. After the profiling, we extracted the total execution time and the frequency of the executions for each function. Although the cProfile can provide us with function level profiling, it cannot provide information on which lines are executed during runtime. Hence, we implemented a simple tracer based on Python's tracing library [22]. Since both runs executed exactly the same workload, the lines of the executed source code should be the same. Hence, we only ran our tracer once. Finally, we parsed the jitted logs we collected for configuration A in order to know the exact lines of source code that were jitted.

Based on the cProfiling results, we calculated the average execution time for each function and computed their differences between the two configuration settings.

We sorted the differences in decreasing order and selected top 30 functions whose performance is worse in configuration A for manual examination.

We found that the main reason behind the worse performance is configuration A is due to the overhead of time switching between the two execution modes (interpreted vs. native execution). When a function is executed each time, PyPy will be running under the default interpreted mode. When a code region is marked as jitted, PyPy will switch from the interpreted mode to the native execution mode and executed the compiled binary code. After the binary code is executed, PyPy will have to switch back to interpreted mode to execute the rest of the function. Executing the compiled binary code is much faster than running the same code under the interpreted mode. However, the switching between the two executing mode takes time. Figure 4.7 shows two such examples. The different text styles are defined as follows:

- **grey**: not executed;
- **bold**: executed but not jitted;
- **bold & highlighted**: jitted under configuration A.

As we can see from the Figure 4.7, only a single line is jitted in both functions. Both lines are related to Python list comprehension, which internally execute a *for* loop. In both cases, PyPy has to switch the execution mode twice: starting

```
def patch_vary_headers(response, newheaders):
      """
    Adds (or updates) the "Vary" header in the given HttpResponse object.
    newheaders is a list of header names that should be in "Vary". Existing
    headers in "Vary" aren't removed.
    """
    # Note that we need to keep the original order intact, because cache
    # implementations may rely on the order of the Vary contents in, say,
    # computing an MD5 hash.
    if response.has_header('Vary'):
        vary_headers = cc_delim_re.split(response['Vary'])
    else:
        vary_headers = []
    # Use .lower() here so we treat headers as case-insensitive.
    existing_headers = set(header.lower() for header in vary_headers)
    additional_headers = [newheader for newheader in newheaders
                          if newheader.lower() not in existing_headers]
    response['Vary'] = ', '.join(vary_headers + additional_headers)
```

(a) Code snippet 1

```
def get_javascript_catalog(locale, domain, packages):
    app_configs = apps.get_app_configs()
    allowable_packages = set(app_config.name for app_config in app_configs)
    allowable_packages.update(DEFAULT_PACKAGES)
    packages = [p for p in packages if p in allowable_packages]
    paths = []
    # paths of requested packages
    for package in packages:
        p = importlib.import_module(package)
        path = os.path.join(os.path.dirname(upath(p.__file__)), 'locale')
        paths.append(path)

    trans = DjangoTranslation(locale, domain=domain, localedirs=paths)
    trans_cat = trans._catalog
```

(b) Code snippet 2

Figure 4.7: Two code snippets showing the executed code and the jitted code under the two configuration settings: A vs. B. Configuration A is a jit-enabed configuration shown in Figure 4.6. It has worse performance than configuration B, which is JIT off. The colour scheme is defined as follows: grey coloured code is for

from the interpreted mode to the native execution mode and back. The time saved under the native execution mode is much smaller than the time takes for switching between the two modes, which causes the performance degradation in configuration A (enabling JIT) when comparing against configuration B (JIT off).

Such jitting behavior can be explained using the configuration settings. The configuration A is shown in Table 4.17. For reference, we also included the default configuration values in the table. The 'trace_limit' in configuration A is set to be a very small value, which would only allow a small region of code to traced and jitted. The smaller the jitted code region is, the less the performance gain code jitting can bring. In this case, the overhead of frequently switching between the two modes out-weights the gain from code jitting.

Findings: Enabling code jitting does not necessarily lead to good performance. Some JIT configuration settings can perform worse than the disabling the JIT completely. This is mainly due to the overhead of switching between the interpreted and the native execution mode.

Implications: Programming language researchers may look into adaptive JIT compilation techniques, which can disable inefficient code jitting behavior during runtime.

Table 4.17: Configuration A and the default configuration.

JIT Configuration Parameter	A	Default
decay	5	40
function_threshold	404	1619
loop_longevity	1000	1000
threshold	4156	1039
trace_eagerness	100	200
trace_limit	374	6000

4.6.4 JIT vs. Memory Usage

The case studies have shown that by using our automated approach, we are able to locate JIT configuration settings whose performance significantly outperform the default configuration setting. The CPU for all the components are better or no worse in the optimal configuration settings than the default. This is mainly because the CPU can process the same amount of work much more efficiently when more code is compiled into efficient machine code. However, the memory usage for the Tornado web servers are much worse. The memory usage for the worker processes for Wagtail even tripled in the optimal configuration settings. Hence, we want to investigate whether there is any relation between the amount of code jitted and the

Table 4.18: Spearman correlation between number of jitted line and memory usage for each system.

System	**correlation**	$p - value$	**scale**
Saleor	0.8244878	2.2e-16	very large
Wagtail	0.882144	2.2e-16	very large
Quokka	0.8549971	2.2e-16	very large

amount of memory used in the worker processes.

For all the performance tests in each case study, we processed the JIT logs to obtain the amount of the jitted source code and extract the memory usage at the end of the test. Then we calculated the Spearman's rank correlation between the lines of the jitted code and the amount of memory usage.

As shown in Table 4.18, there is a very large correlation between the memory usage of the web server processes and the amount of the jitted code. In other words, the larger the amount of the jitted code, the higher the memory usage for the worker processes. As more code is jitted, a larger amount of compiled machine code is generated. The generated machine code will be kept in the memory during the system execution.

Table 4.19: Runtime statistics for ESM-MOGA under different termination criteria.

	Saleor			**Wagtail**			**Quokka**		
Stoppage Criteria	T_0	T_1	T_2	T_0	T_1	T_2	T_0	T_1	T_2
# of Generation	3	3	3	7	3	3	3	3	3
Evaluated Configurations	100	100	100	199	97	97	96	96	96
Duration (hours)	36	36	36	67	34	34	35	35	35

Findings: The improvement in response time using the JIT compilation process is at the cost of higher memory usage.

Implications: More jitted code can generally lead to more responsive system. However, the number of jitted lines should be kept in a moderate range as: (1) more jitted code means higher memory usage, and (2) the configuration settings with the highest amount of jitted lines will not guarantee the best performance. One of the interesting future research work would be incorporating memory usage as one of the objectives in the ESM-MOGA while searching for the optimal configuration settings.

4.6.5 Termination criteria

The above case studies show that among the three studied systems, all top three configuration settings significantly outperform the default configuration setting. We set (T_0:) $|MDR| \leq 0.1$ in the hope that there is a higher chance to obtain the optimal configuration settings, as solutions in the previous generation are good enough so that little optimization can be made during the last generation before termination. However, such conditions may be too strict and cost too much time ($\geq$ 35 hours as shown in Table 4.11). Many systems nowadays need to be updated more frequently (e.g., daily or even a few times a day) under the continuous integration/continuous delivery processes. Hence, during the case studies, we also examined the following two termination conditions: (1) (T_1:) $|MDR| \leq 0.25$, and (2) (T_2:) $|MDR| \leq 0.50$, in the hope that the search process terminates earlier (a.k.a., saving the time for searching), while we are still able to locate the optimal solutions.

Table 4.19 shows the runtime statistics for ESM-MOGA under three different termination criteria: T_0, T_1, and T_2. All termination criteria stopped at the same number of generations for Saleor and Quokka. However, for Wagtail, the less strict termination criteria T_1 and T_2 are met after the third generation. Hence, we also extracted the top three configuration settings for Wagtail at the end of the third

generation for further comparison.

We performed a pairwise comparison using the dominant comparison function between the top three configuration settings under termination criteria T_1 and T_2 (a.k.a., stopped after the third generation) and the top three configuration settings under termination criterion T_0 (a.k.a., stopped after the seventh generation). The results show that two out of the three top configuration settings under T_0 dominate all top three configuration settings under T_1 and T_2. The other remaining top three configuration settings under T_0 show no dominance when comparing against one of the top configuration settings under T_1 and T_2. The system performance under the top three configuration settings under T_1 and T_2 also shows large improvement for all scenarios when comparing against the default setting.

Findings: We have evaluated the ESM-MOGA under three different termination criteria: (1) $MDR \geq 0.50$, (2) $MDR \geq 0.25$, and (3) $MDR \geq 0.10$. The ESM-MOGA can terminate successfully (a.k.a., finding the optimal solutions) under all three criteria for the three case study systems. And the less strict termination criteria T_1 and T_2 can obtain some configuration settings which are as good as some top configuration settings under termination criterion T_0.

Implications: There are various configuration parameters within ESM-MOGA. It requires further research to systemically tune ESM-MOGA configuration parameters in order to achieve the best performance (finding the optimal solutions within the shortest amount of time). Furthermore, it would be beneficial to compare ESM-MOGA against other hyper-parameter tuning techniques (e.g., Google Viser [52]).

4.7 Threats to Validity

In this section, we will discuss the threats to validity.

4.7.1 Construct Validity

To avoid measurement errors and noise, we repeated each experiment 30 times [50].

We leveraged techniques from Alghmadi et al. [29] to determine the duration of performance tests, when the performance behavior becomes repetitive. We have

found that after 40 minutes under the default configuration settings for the three systems, the performance behavior becomes repetitive. For consistency concerns, we took additional 10 minutes of the performance data for our performance tests for the warmed up period. We assumed the warmup period would be similar under other configuration settings. To verify this threshold, we randomly sampled five performance test from all performance testing runs in each case study. For each sampled test, we divided the performance data into intervals of ten minutes and compared the performance behavior among the adjacent intervals. Our analyses confirmed that the performance behavior also became repetitive after 40 minutes under these five sampled configuration settings.

Since the JIT logs do not contain timestamps, the only way to monitor the jitting progress for PyPy is to periodically take snapshots of the existing JIT logs. However, regularly taking snapshots of the JIT logs would bring huge performance overhead for a server-based system. Hence, to minimize the measurement impact, we did not take snapshots in the middle of the performance tests in our case study in Section 4.5. Instead, we estimated the jitting progress by judging whether the system performance behavior stabilizes (a.k.a., becoming repetitive).

Our approach, ESM-MOGA, is a tailed version of the Multi-Objective Genetic Algorithm, which uses effect size measures to compare the results of different test runs. MOGA is an efficient search-based technique, which automatically explores

the solution space. It has been used widely in various software engineering research areas (e.g., test case generation [28, 46, 78], software architecture [56], and bug prediction [35]) and is shown to be highly effective. Although our case study results show that the configurations derived from the ESM-MOGA approach yield much better performance than the default configuration, the ESM-MOGA might not be the most efficient approach to locate the optimal JIT configuration setting(s). Furthermore, the search time that it takes to find the optimal solutions using ESM-MOGA varies depending on the systems and their associated workload. One of the future areas of research is to evaluate the effectiveness of various hyper-configuration tuning techniques in the context of tuning JIT configuration parameters.

4.7.2 Internal Validity

We kept all the other factors (e.g., the versions of the systems, the deployment infrastructure, and the workload) the same, while varying the JIT configuration settings for each performance tests.

In this paper, we assumed the systems which undergo the JIT tuning process, can handle the exercised workload. In other words, the systems are not in a bottleneck state when we tune their JIT configurations. It's a common practice that the top priority for bottlenecked systems would be performance diagnosis and migration actions instead of tuning their JIT configuration settings.

We used the WRS test and the values from CD to implement our dominance functions in the ESM-MOGA. WRS is a statistical test which compares the distributions of two datasets. CD is an effect size measure, which indicates the strength of the difference between two datasets. Both the WRS test and the CD are non-parametric tests, which do not hold any assumptions regarding the underlying distributions of the data. The two techniques have been used together in previous works [49, 48] to evaluate the system performance between two alternatives.

4.7.3 External Validity

In this paper, we have conducted a case study on the performance impact of the JIT configuration settings from PyPy. The experimented PyPy version is PyPy 5.7.1, which corresponds to Python version *2.7.13*. The empirical findings in the exploratory studies may not be generalizable to other Python versions (e.g, Python 3), other Python implementations (e.g., Jython), or other programming languages (e.g., Java or C#).

Our case study results have shown that the optimal JIT configuration settings vary from systems to systems. Our search-based configuration tuning framework can be used to automatically search for configuration settings, which are much better than the default. Our automated tuning technique can also be used to tune the JIT performance of other programming languages, whose parameters are integer

types. We plan to extend our approach in the future to accommodate other types of configuration parameters (e.g., string, float and boolean types).

As each experiment requires starting the Python-based applications with a different set of configuration parameters, it is not yet practical to apply ESM-MOGA techniques into the continuous integration and continuous delivery process. One of the future areas of research is to look into techniques like transfer learning [62] to infer the optimal configurations for the newer releases of the same systems or even configuration parameters other systems.

4.8 Conclusion

The JIT compilation is introduced to improve the runtime performance of software systems. During the system execution, various regions of the systems are compiled into the binary executable format, so that they can be executed more efficiently. In this paper, we have performed an empirical study on the performance impact of PyPy's JIT configuration settings. In particular, we have compared the performance differences between the default and some other configuration settings. We have shown that systems running under the default configuration setting does not necessarily yield the best performance. In addition, we have shown that there is no strong connection between the JIT coverage and the system performance and the optimal JIT configuration settings are system dependent. To cope with such

findings, we have developed a search-based approach, called ESM-MOGA, which automatically tunes the JIT configuration settings for a given system. Case studies have shown that systems running under the resulting configuration settings are significantly faster than the default configuration setting.

5 Conclusions and Future work

Python, as one of the most popular programming language, has been widely used by different domains, like science computing, web service, and data process, due to its dynamic features, enriched library, and large community. Unfortunately, no many empirical studies in the software engineering research field have been devoted to Python-based applications. In this thesis, we have conducted an empirical study on the evolution and the performance of PyPy, a Python-based compiler project.

In the first part of this thesis, we studied the evolution of PyPy and compared our findings with the work of Lin et al. [69], who studied the fine-grained source code changes on ten Python application across five domains (Web, Data processing, Science computing, NLP, Media) and reported 11 interesting findings from four dimensions. We first implemented a public available Python source code diffing tool PyDiff and applied PyDiff on all PyPy's historical commits. The result of the replication study shows that 6 out of 9 findings in the original study doesn't hold in the PyPy project. The evolution pattern like change type frequency and dynamic

change actions show significant differences between PyPy and the originally studied projects.

PyPy mainly gains its performance via its JIT compiler. However, the tuning of the JIT compiler configuration is usually ignored by the PyPy users and lead to suboptimal performance. In the second part of the thesis, we performed an exploratory study on PyPy's JIT configuration parameters. Our study shows the configuration of the JIT compiler can make a big difference to the performance of applications running on the top of the compiler. Based on our findings, we proposed our search-based configuration tuning approach, (ESM-MOGA) to tuning the JIT configuration for the applications running on the top. Case studies on three open source systems show that systems running under the resulting configuration settings significantly out-perform (5% - 60% improvement in average peak performance) the default configuration setting.

In the future, we plan to extend the scale of our replication study in source code changes. In particular, we want to understand whether similar findings can be found in Python applications from other domains. Also, we want to see if this research can be applied to applications that are written by other dynamic program languages (e.g. Java). For our work of JIT configuration tuning, we plan to further expand the ESM-MOGA to accommodate more objectives (e.g., memory and network efficiency) during its search process. We also would like to apply

the ESM-MOGA on other Python-based applications or frameworks (e.g., machine learning based frameworks like TensorFlow). In addition, we plan to explore the use of data mining or experimental design techniques to further reduce the number of performance tests conducted during the search process. Finally, we would like to evaluate the effectiveness of various hyper-parameter tuning techniques in the context of tuning JIT configurations.

Bibliography

[1] Apache JMeter. http://jmeter.apache.org/, visited 2015-10-23

[2] Chrome V8 JavaScript engine. https://opensource.google.com/projects/v8. Last accessed 2/19/2019

[3] Distributed Evolutionary Algorithms in Python (DEAP). https://github.com/DEAP/deap. Last accessed 10/06/2017

[4] Django web framework. https://www.djangoproject.com/. Last accessed 02/19/2019

[5] Evaluation tools in Python (DEAP). http://deap.readthedocs.io/en/master/api/tools.html?highlight=dominance. Last accessed 10/06/2017

[6] Flask web framework. http://flask.pocoo.org/. Last accessed 02/19/2019

[7] IBM Java 8 JIT and AOT command-line options. https://www.ibm.com/support/knowledgecenter/SSYKE2_8.0.0/com.ibm.java.aix.80.doc/diag/appendixes/cmdline/commands_jit.html. Last accessed 10/06/2017

[8] Java Microbenchmark Harness (JMH). http://openjdk.java.net/projects/code-tools/jmh/. Last accessed 10/06/2017

[9] Numpy: scientific computing with Python. http://www.numpy.org/. Last accessed 02/19/2019

[10] Oracle Java 8 Advanced JIT Compiler Options. https://docs.oracle.com/javase/8/docs/technotes/tools/windows/java.html#BABDDFII. Last accessed 10/06/2017

[11] Pandas: python data analysis library. https://pandas.pydata.org/. Last accessed 02/19/2019

[12] Performance monitoring tools for Linux. https://github.com/sysstat/sysstat, visited 2015-10-23

[13] Pydiff. https://bitbucket.org/YangguangLi/pydiffer/src/master/

[14] PyPy speed center. http://speed.pypy.org/. Last accessed 10/04/2017

[15] Quokka CMS (Content Management System) - Python, Flask and MongoDB. http://quokkaproject.org/. Last accessed 10/06/2017

[16] Replication package. https://github.com/seasun525/PyPyJITTuner

[17] Saleor - An e-commerce storefront for Python and Django. http://getsaleor.com/. Last accessed 10/06/2017

[18] scikit-learn: machine learning in Python. https://scikit-learn.org/stable/. Last accessed 02/19/2019

[19] TechEmpower Web Framework Benchmarks. https://www.techempower.com/benchmarks/. Last accessed 10/04/2017

[20] Tensorflow: machine learning framework. https://www.tensorflow.org/. Last accessed 02/19/2019

[21] The Python Profilers. https://docs.python.org/2/library/profile.html. Last accessed 10/28/2018

[22] Tracing a Program As It Runs. https://pymotw.com/2/sys/tracing.html. Last accessed 10/28/2018

[23] vmprof - a statistical program profiler. http://vmprof.com/. Last accessed 10/06/2017

[24] Wagtail CMS: Django Content Management System. https://wagtail.io/. Last accessed 10/06/2017

[25] What is Load Balancing? https://www.nginx.com/resources/glossary/load-balancing/. Last accessed 10/06/2017

[26] What is python used for at Google? https://www.quora.com/What-is-python-used-for-at-Google. Last accessed 10/06/2017

[27] Wilcoxon rank-sum test. https://en.wikipedia.org/wiki/Mann%E2%80%93Whitney_U_test

[28] Abdessalem, R.B., Panichella, A., Nejati, S., Briand, L.C., Stifter, T.: Testing autonomous cars for feature interaction failures using many-objective search. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE) (2018)

[29] Alghmadi, H.M., Syer, M.D., Shang, W., Hassan, A.E.: An automated approach for recommending when to stop performance tests. In: 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 279–289 (2016)

[30] Barrett, E., Bolz-Tereick, C.F., Killick, R., Mount, S., Tratt, L.: Virtual machine warmup blows hot and cold. Proceedings of the ACM Programming Language **1**(OOPSLA), 52:1–52:27 (2017). DOI 10.1145/3133876. URL http://doi.acm.org/10.1145/3133876

[31] Bolz, C.F., Cuni, A., Fijalkowski, M., Rigo, A.: Tracing the meta-level: Pypy's tracing jit compiler. In: Proceedings of the 4th Workshop on the Implementa-

tion, Compilation, Optimization of Object-Oriented Languages and Programming Systems (ICOOOLPS), pp. 18–25 (2009)

[32] Bondi, A.B.: Automating the analysis of load test results to assess the scalability and stability of a component. In: Proceedings of the 2007 Computer Measurement Group Conference (CMG), pp. 133–146 (2007)

[33] Brecht, T., Arjomandi, E., Li, C., Pham, H.: Controlling garbage collection and heap growth to reduce the execution time of java applications. ACM Trans. Program. Lang. Syst. **28**(5), 908–941 (2006). DOI 10.1145/1152649.1152652. URL http://doi.acm.org/10.1145/1152649.1152652

[34] Candan, K.S., Li, W.S., Luo, Q., Hsiung, W.P., Agrawal, D.: Enabling dynamic content caching for database-driven web sites. In: Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data (SIGMOD), pp. 532–543 (2001)

[35] Canfora, G., Lucia, A.D., Penta, M.D., Oliveto, R., Panichella, A., Panichella, S.: Multi-objective cross-project defect prediction. In: Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation (ICST) (2013)

[36] Chawathe, S.S., Rajaraman, A., Garcia-Molina, H., Widom, J.: Change detection in hierarchically structured information. In: ACM SIGMOD Record, vol. 25, pp. 493–504. ACM (1996)

[37] Clark, M.: How the BBC builds websites that scale. http://www.creativebloq.com/features/how-the-bbc-builds-websites-that-scale (2017). Last accessed 10/06/2017

[38] Cramer, T., Friedman, R., Miller, T., Seberger, D., Wilson, R., Wolczko, M.: Compiling java just in time. IEEE Micro **17**(3), 36–43 (1997)

[39] Deb, K., Agrawal, S., Pratap, A., Meyarivan, T.: A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: Nsga-ii. In: International Conference on Parallel Problem Solving From Nature, pp. 849–858. Springer (2000)

[40] Duan, S., Thummala, V., Babu, S.: Tuning Database Configuration Parameters with iTuned. Proceedings of the VLDB Endowment **2**(1), 1246–1257 (2009). DOI 10.14778/1687627.1687767. URL http://dx.doi.org/10.14778/1687627.1687767

[41] Eaton, K.: How One Second Could Cost Amazon $1.6 Billion In Sales. https://www.fastcompany.com/1825005/how-one-second-could-cost-amazon-16-billion-sales. Last accessed 10/06/2017

[42] Falleri, J.R., Morandat, F., Blanc, X., Martinez, M., Monperrus, M.: Fine-grained and accurate source code differencing. In: Proceedings of the 29th ACM/IEEE international conference on Automated software engineering, pp. 313–324. ACM (2014)

[43] Fluri, B., Gall, H.C.: Classifying change types for qualifying change couplings. In: Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on, pp. 35–45. IEEE (2006)

[44] Fluri, B., Giger, E., Gall, H.C.: Discovering patterns of change types. In: Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, pp. 463–466. IEEE Computer Society (2008)

[45] Fluri, B., Wuersch, M., PInzger, M., Gall, H.: Change distilling: Tree differencing for fine-grained source code change extraction. IEEE Transactions on software engineering **33**(11) (2007)

[46] Fraser, G., Arcuri, A.: Evosuite: Automatic test suite generation for object-oriented software. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE) (2011)

[47] Gal, A., Eich, B., Shaver, M., Anderson, D., Mandelin, D., Haghighat, M.R., Kaplan, B., Hoare, G., Zbarsky, B., Orendorff, J., Ruderman, J., Smith, E.W.,

Reitmaier, R., Bebenita, M., Chang, M., Franz, M.: Trace-based just-in-time type specialization for dynamic languages. In: Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pp. 465–478 (2009)

[48] Gao, R., Jiang, Z.M.J.: An exploratory study on assessing the impact of environment variations on the results of load tests. In: Proceedings of the 14th International Conference on Mining Software Repositories (MSR) (2017)

[49] Gao, R., Jiang, Z.M.J., Barna, C., Litoiu, M.: A framework to evaluate the effectiveness of different load testing analysis techniques. In: 2016 IEEE International Conference on Software Testing, Verification and Validation (ICST) (2016)

[50] Georges, A., Buytaert, D., Eeckhout, L.: Statistically Rigorous Java Performance Evaluation. In: Proceedings of the 22nd International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA) (2007)

[51] Gewirtz, D.: Which programming languages are most popular (and what does that even mean)? http://www.zdnet.com/article/which-programming-languages-are-most-popular-and-what-does-that-even-mean/ (2017). Last accessed 10/06/2017

[52] Golovin, D., Solnik, B., Moitra, S., Kochanski, G., Karro, J., Sculley, D.: Google vizier: A service for black-box optimization. In: Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD) (2017)

[53] Gong, L., Pradel, M., Sen, K.: Jitprof: Pinpointing jit-unfriendly javascript code. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE), pp. 357–368 (2015)

[54] Grigorik, I.: Optimizing Encoding and Transfer Size of Text-Based Assets. https://developers.google.com/web/fundamentals/performance/optimizing-content-efficiency/optimize-encoding-and-transfer. Last accessed 10/06/2017

[55] Hashemi, M.: 10 Myths of Enterprise Python. https://www.paypal-engineering.com/2014/12/10/10-myths-of-enterprise-python/ (2014). Last accessed 10/06/2017

[56] Henard, C., Papadakis, M., Harman, M., Traon, Y.L.: Combining multi-objective search and constraint solving for configuring large software product lines. In: Proceedings of the 37th International Conference on Software Engineering (ICSE) (2015)

[57] Hopkins, W.G.: A new view of statistics. [Online accessed 2017-10-14] http://www.sportsci.org/resource/stats/index.html (2016)

[58] Hoskins, D.S., Colbourn, C.J., Montgomery, D.C.: Software performance testing using covering arrays: Efficient screening designs with categorical factors. In: Proceedings of the 5th International Workshop on Software and Performance (WOSP)

[59] Hoste, K., Georges, A., Eeckhout, L.: Automated just-in-time compiler tuning. In: Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO), pp. 62–72 (2010)

[60] Insights, G.P.: Remove Render-Blocking JavaScript. https://developers.google.com/speed/docs/insights/BlockingJS. Last accessed 10/06/2017

[61] Jamshidi, P., Siegmund, N., Velez, M., Kästner, C., Patel, A., Agarwal, Y.: Transfer learning for performance modeling of configurable systems: An exploratory analysis. In: Proceedings of the International Conference on Automated Software Engineering (ASE) (2017)

[62] Jamshidi, P., Siegmund, N., Velez, M., Kästner, C., Patel, A., Agarwal, Y.: Transfer learning for performance modeling of configurable systems: an exploratory analysis. In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE) (2017)

[63] Jantz, M.R., Kulkarni, P.A.: Exploring single and multilevel jit compilation policy for modern machines 1. ACM Transactions on Architecture and Code Optimization (TACO) **10**(4), 22:1–22:29 (2013)

[64] Jiang, Z.M.J., Hassan, A.E.: A survey on load testing of large-scale software systems. IEEE Transactions on Software Engineering (TSE) (2015)

[65] Kampenes, V.B., Dybå, T., Hannay, J.E., Sjøberg, D.I.K.: Systematic review: A systematic review of effect size in software engineering experiments. Information Software Technology **49**(11-12), 1073–1086 (2007)

[66] Komorn, R.: Python in production engineering. https://code.facebook.com/posts/1040181199381023/python-in-production-engineering/ (2016). Last accessed 10/06/2017

[67] Lengauer, P., Mössenböck, H.: The Taming of the Shrew: Increasing Performance by Automatic Parameter Tuning for Java Garbage Collectors. In: Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering (ICPE), pp. 111–122 (2014)

[68] Libič, P., Bulej, L., Horky, V., Tůma, P.: On the limits of modeling generational garbage collector performance. In: Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering (ICPE) (2014)

[69] Lin, W., Chen, Z., Ma, W., Chen, L., Xu, L., Xu, B.: An empirical study on the characteristics of python fine-grained source code change types. In: 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 188–199. IEEE (2016)

[70] Lion, D., Chiu, A., Sun, H., Zhuang, X., Grcevski, N., Yuan, D.: Don't get caught in the cold, warm-up your jvm: Understand and eliminate jvm warm-up overhead in data-parallel systems. In: Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI), pp. 383–400 (2016)

[71] Malloy, B.A., Power, J.F.: An empirical analysis of the transition from python 2 to python 3. Empirical Software Engineering pp. 1–28 (2018)

[72] Martí, L., García, J., Berlanga, A., Molina, J.M.: An approach to stopping criteria for multi-objective optimization evolutionary algorithms: The mgbm criterion. In: Evolutionary Computation, 2009. CEC'09. IEEE Congress on, pp. 1263–1270. IEEE (2009)

[73] Oaks, S.: Java Performance: The Definitive Guide, 1st edn. O'Reilly Media, Inc. (2014)

[74] Osogami, T., Kato, S.: Optimizing System Configurations Quickly by Guessing at the Performance. In: Proceedings of the 2007 ACM SIGMETRICS In-

ternational Conference on Measurement and Modeling of Computer Systems (SIGMETRICS) (2007)

[75] Romano, D., Pinzger, M.: Analyzing the evolution of web services using fine-grained changes. In: 2012 IEEE 19th International Conference on Web Services, pp. 392–399. IEEE (2012)

[76] Romano, J., Kromrey, J.D., Coraggio, J., Skowronek, J.: Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen'sd for evaluating group differences on the NSSE and other surveys? In: Annual meeting of the Florida Association of Institutional Research (2006)

[77] Rossum, G.: Python reference manual (1995)

[78] Shamshiri, S., Rojas, J.M., Fraser, G., McMinn, P.: Random or genetic algorithm search for object-oriented test suite generation? In: Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation (GECCO) (2015)

[79] Shang, W., Jiang, Z.M., Adams, B., Hassan, A.E.: Mapreduce as a general framework to support research in mining software repositories (msr). In: Mining Software Repositories, 2009. MSR'09. 6th IEEE International Working Conference on, pp. 21–30. IEEE (2009)

[80] Siegmund, N., Grebhahn, A., Apel, S., Kästner, C.: Performance-influence models for highly configurable systems. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015. ACM (2015)

[81] Siegmund, N., Kolesnikov, S.S., Kästner, C., Apel, S., Batory, D., Rosenmüller, M., Saake, G.: Predicting Performance via Automated Feature-interaction Detection. In: Proceedings of the 34th International Conference on Software Engineering (ICSE) (2012)

[82] Singer, J., Brown, G., Watson, I., Cavazos, J.: Intelligent selection of application-specific garbage collectors. In: Proceedings of the 6th International Symposium on Memory Management, ISMM '07

[83] Singh, R., Bezemer, C.P., Shang, W., Hassan, A.E.: Optimizing the performance-related configurations of object-relational mapping frameworks using a multi-objective genetic algorithm. In: Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering (ICPE), pp. 309–320 (2016)

[84] Sopitkamol, M., Menascé, D.A.: A method for evaluating the impact of software configuration parameters on e-commerce sites. In: Proceedings of the

5th International Workshop on Software and Performance (WOSP), pp. 53–64 (2005)

[85] Thonangi, R., Thummala, V., Babu, S.: Finding Good Configurations in High-Dimensional Spaces: Doing More with Less. In: 2008 IEEE International Symposium on Modeling, Analysis and Simulation of Computers and Telecommunication Systems (2008)

[86] Thung, F., Lo, D., Jiang, L.: Automatic recovery of root causes from bug-fixing changes. In: 2013 20th Working Conference on Reverse Engineering (WCRE), pp. 92–101. IEEE (2013)

[87] Wang, K., Lin, X., Tang, W.: Predator - An experience guided configuration optimizer for Hadoop MapReduce. In: 4th IEEE International Conference on Cloud Computing Technology and Science Proceedings (2012)

[88] Wimmer, C., Brunthaler, S.: Zippy on truffle: A fast and simple implementation of python. In: Proceedings of the 2013 Companion Publication for Conference on Systems, Programming, & Applications: Software for Humanity (SPLASH) (2013)

[89] Würthinger, T., Wimmer, C., Humer, C., Wöß, A., Stadler, L., Seaton, C., Duboscq, G., Simon, D., Grimmer, M.: Practical partial evaluation for high-performance dynamic language runtimes. In: Proceedings of the 38th ACM

SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, pp. 662–676. ACM, New York, NY, USA (2017). DOI 10.1145/3062341.3062381. URL http://doi.acm.org/10.1145/3062341.3062381

[90] Würthinger, T., Wimmer, C., Wöß, A., Stadler, L., Duboscq, G., Humer, C., Richards, G., Simon, D., Wolczko, M.: One vm to rule them all. In: Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward!) (2013)

[91] Xi, B., Liu, Z., Raghavachari, M., Xia, C.H., Zhang, L.: A Smart Hill-climbing Algorithm for Application Server Configuration. In: Proceedings of the 13th International Conference on World Wide Web (WWW), pp. 287–296. ACM, New York, NY, USA (2004). DOI 10.1145/988672.988711. URL http://doi.acm.org/10.1145/988672.988711

[92] Xu, T., Jin, X., Huang, P., Zhou, Y., Lu, S., Jin, L., Pasupathy, S.: Early Detection of Configuration Errors to Reduce Failure Damage. In: Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI) (2016)

[93] Yilmaz, C., Porter, A., Krishna, A.S., Memon, A.M., Schmidt, D.C., Gokhale, A.S., Natarajan, B.: Reliable effects screening: A distributed continuous quality assurance process for monitoring performance degradation in evolving soft-

ware systems. IEEE Transactions on Software Engineering (TSE) **33**(2), 124–141 (2007). DOI 10.1109/TSE.2007.20