UC San Diego
UC San Diego Electronic Theses and Dissertations

Title
Software Configuration Learning and Recommendation

Permalink
https://escholarship.org/uc/item/7fp0g525

Author
Zhang, Jiaqi

Publication Date
2014

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA, SAN DIEGO

Software Configuration Learning and Recommendation

A dissertation submitted in partial satisfaction of the
requirements for the degree of Doctor of Philosopy

in

Computer Science

by

Jiaqi Zhang

Committee in charge:

Professor Yuanyuan Zhou, Chair
Professor Steve Bin Jiang
Professor Jason Mars
Professor Lawrence Saul
Professor Stefan Savage
Professor Geoffrey Voelker

2014

Copyright

Jiaqi Zhang, 2014

All rights reserved.

The Dissertation of Jiaqi Zhang is approved and is acceptable in quality and form for publication on microfilm and electronically:

Chair

University of California, San Diego

2014

DEDICATION

To Liancheng Zhang and Weihong Liu, my parents and my foundation.

To all my friends who have supported me throughout the process.

TABLE OF CONTENTS

Signature Page iii

Dedication iv

Table of Contents v

List of Figures vii

List of Tables ix

Acknowledgements xi

Vita xiii

Abstract of the Dissertation xiv

Chapter 1 Introduction 1
1.1 Configuration Correctness 1
1.2 Configuration Performance 6
1.3 Contributions 7

Chapter 2 Findings and Design Principles 10
2.1 Characteristics of Configuration Parameters 10
2.2 Challenges in Applying Data Mining 12

Chapter 3 Misconfiguration Detection with EnCore 17
3.1 System Architecture 17
3.1.1 Data Collector 17
3.1.2 Data Assembler 18
3.1.3 Rule Generator 19
3.1.4 Anomaly Detector 20
3.1.5 Where and how to use the proposed tool 21
3.2 Data Assembler 21
3.2.1 Parsing Configuration Files 22
3.2.2 Inferring Configuration Entry Types 23
3.2.3 Environment Information Integration 25
3.3 Rules Inference 27
3.3.1 Template-Guided Inference Based on Types 28
3.3.2 Rule Filtering 32
3.3.3 Customization 33
3.4 Anomaly Detector 38
3.5 Related Work 41

Chapter 4 Accelerating Automatic Performance Tuning with iOpt 43
4.1 Performance Impact of Software Coniguration Settings 43
4.2 Helping Automatic Performance Tuning 46
4.2.1 Existing Approaches to Help Performance Tuning 46
4.3 Observations 48
4.4 iOpt Design 52
4.4.1 iOpt Usage 52
4.4.2 Architecture Design 54

Chapter 5 Experimental Measurements 62
5.1 Misconfiguration Detection Effectiveness 62
5.1.1 Injected Misconfigurations 63
5.1.2 Real Misconfigurations Problems 66
5.1.3 Detecting New Misconfigurations 67
5.2 Type Inference Accuracy 68
5.3 Correlation Rule Inference 70
5.4 Finding Performance Parameters with iOpt 73
5.4.1 iOpt Case Study 74

Bibliography 77

LIST OF FIGURES

Figure 1.1. **Examples of misconfigurations from real world.** Note that while the first problem can be directly told from the missing value, it is not applicable to the later three. 3

Figure 3.1. **The architecture of *EnCore* .** Both of the data assembler and the rule inference can be customized by users optionally. 18

Figure 3.2. **Use of EnCore.** 21

Figure 3.3. **Data assembling.** In addition to the configuration files, the assembler also takes the system environment data and users' input for type inference and augmenting the configuration entries. 23

Figure 3.4. **Examples of templates and the concrete rules generated from them.** 29

Figure 3.5. **Workflow of rule inference.** 32

Figure 3.6. **Grammar of template specification.** 34

Figure 3.7. **Format of the customization file.** 36

Figure 4.1. **Configurations impact on Hadoop performance.** With the changes of configuration settings, Hadoop has varied performance. 44

Figure 4.2. **Configurations impact on Hadoop performance.** Performance variation with the change of different single configuration parameters in PostgreSQL. 45

Figure 4.3. **Using iOpt in performance tuning.** The gray boxes are the elements of iOpt. The upper side of the figure needs to be done in the software developer side, while the lower part is in the software user site. 53

Figure 4.4. **iOpt configuration parameter analysis architecture.** 55

Figure 4.5. **iOpt configuration parameter analysis architecture.** 57

Figure 4.6. **iOpt configuration parameter recognizing.** 59

Figure 4.7. **iOpt value constraints recognization** 61

Figure 5.1. **MapReduce memory usage inferred from iOpt results.** 75

Figure 5.2. **Performance tuning results with iOpt memory hierarchy reports.** 76

LIST OF TABLES

Table 2.1. **Number (percentage) of configuration parameters that are associated with environment and correlations.** Apache includes the entries of two main modules: core and mpm; PHP includes the core entries; MySQL's entries are randomly sampled. 11

Table 2.2. **The number of attributes generated using data mining methods in the studied software.** 15

Table 2.3. **Time cost (in seconds) and size of frequent item set with different number of attributes.** 16

Table 3.1. **Predefined type and inference methods.** Due to space limit, we show here simplified regular expressions used in syntactic matching. More complicated expressions are used for other values, e.g., IPv6 address, which is not shown in the table. 24

Table 3.2. **Default augmented environment information.** 26

Table 3.3. **Predefined templates with description.** The templates are created based on the common correlations of configuration settings. The operators carry different meanings for different types, and can be overridden by users' customization. 31

Table 3.4. **Data structures for accessing environment information.** *Only available when collecting data from running instances; **The hardware specifications are not available for newly instantiated virtual machine images such as those from Amazon EC2. 39

Table 4.1. **Number (percentage) of configuration parameters that are related to performance.** "Total" is the number of all the configuration entries for each application. "Performance" is the number of configuration entries related to performance tuning. 49

Table 4.2. **The number of configuration parameters related to performance in each category.** 51

Table 5.1. **The number of injected misconfigurations detected by EnCore in the injection experiment.** 63

Table 5.2. **Detection of real world misconfigurations.** 65

Table 5.3. **Categories of new detected misconfigurations.** "FilePath" means the path setting is missing or set wrongly. "Permission" means the permission setting is wrong. "ValueCompare" shows the misoncifugraions violating value comparison rules. 67

Table 5.4. **Data type detection results.** 69

Table 5.5. **Detected correlation rules with the filters.** 71

Table 5.6. **Effectiveness of the entropy filter.** 72

Table 5.7. **Performance related parameters found with iOpt**. 73

Table 5.8. **The constraints and dependencies inference and their accuracy.** 74

ACKNOWLEDGEMENTS

I would like to thank Professor Yuanyuan Zhou for her support as the chair of my committee, and more importantly, as the advisor of both my research work and research spirit during the 5 years of my graduate study. Without her guidance through every single detail in the projects, there will not be this dissertation or any publications described here.

I would like to thank the members of my committee for their extreme care, and the efforts they put to make this happen.

I would like to thank all my fellow doctoral students in the Opera group, who gave me tremendous help in both my research and life.

I would like to thank Lakshmi and Xiaolan, my mentors during the interns. They have helped me in every way they can with their warm heart.

I would like to thank everyone in Sysnet-both the faculties and the students-they made every magic in the sysnet to build a big family where everyone cares each other.

Chapter 1, in part, is a reprint of the material as it appears in 19th International Conference on Architectural Support for Programming Languages and Operating Systems. Jiaqi Zhang, Lakshminarayanan Renganarayana, Xiaolan Zhang, Niyu Ge, Vasanth Bala, Tianyin Xu, and Yuanyuan Zhou. The dissertation/thesis author was the primary investigator and author of this paper.

Chapter 2, in part, is a reprint of the material as it appears in 19th International Conference on Architectural Support for Programming Languages and Operating Systems. Jiaqi Zhang, Lakshminarayanan Renganarayana, Xiaolan Zhang, Niyu Ge, Vasanth Bala, Tianyin Xu, and Yuanyuan Zhou. The dissertation/thesis author was the primary investigator and author of this paper.

Chapter 3, in full, is a reprint of the material as it appears in 19th International Conference on Architectural Support for Programming Languages and Operating Sys-

tems. Jiaqi Zhang, Lakshminarayanan Renganarayana, Xiaolan Zhang, Niyu Ge, Vasanth Bala, Tianyin Xu, and Yuanyuan Zhou. The dissertation/thesis author was the primary investigator and author of this paper.

Chapter 5, in part, is a reprint of the material as it appears in 19th International Conference on Architectural Support for Programming Languages and Operating Systems. Jiaqi Zhang, Lakshminarayanan Renganarayana, Xiaolan Zhang, Niyu Ge, Vasanth Bala, Tianyin Xu, and Yuanyuan Zhou. The dissertation/thesis author was the primary investigator and author of this paper.

VITA

2006	Bachelor of Engineering, Tsinghua University, Beijing, China
2009	Master of Engineering, Tsinghua University, Beijing, China
2009–2014	Research Assistant, University of California, San Diego
2014	Doctor of Philosophy, University of California, San Diego

PUBLICATIONS

"EnCore: Exploiting System Environment and Correlation Information for Misconfiguration Detection" in Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14), Salt Lake City, UT. March 2014.

"ATDetector: Improving the Accuracy of a Commercial Data Race Detector by Identifying Address Transfer" in Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture 2011 (MICRO'11), December 2011. Port Algre, Brazil. December 2011

" Do Not Blame Users for Misconfigurations" in Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13), Nemacolin Woodlands Resort, Pennsylvania, November 2013

" When Good Services Go Wild: Reassembling Web Services for Unintended Purposes" in Proceedings of the 7th USENIX workshop on Hot Topics in Security (HotSec'12), Beelevue, WA, 2012

"Ad Hoc Synchronization Considered Harmful" In the proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI'10), Vancouver, BC, CA, October 2010

FIELDS OF STUDY

Major Field: Computer Science and Engineering

ABSTRACT OF THE DISSERTATION

Software Configuration Learning and Recommendation

by

Jiaqi Zhang

Doctor of Philosophy in Computer Science

University of California, San Diego, 2014

Professor Yuanyuan Zhou, Chair

With software systems becoming more and more complex and configurable, failures due to misconfigurations are becoming a prominent problem. In addition to the severe consequences such failures can cause, the diagnosis process can be difficult and costly. Besides, configurations also heavily affect the system performance. The selection of optimal configuration settings to achieve high performance is thus desired but usually takes a long time. This paper presents the efforts to help with the system correctness and performance problem by configuration analysis.

To tackle the correctness problem and automatically detect software Misconfig-

urations, we take into account two important factors that are unexploited before: the interaction between the executing environment and the configuration file, and the rich correlations between configuration entries.

We leverage the fact that with the emerging cloud virtual machines, more system data than just the configuration files are accessible. With the training data enriched with whole system information, our tool learns multiple aspects of the configuration files from the whole system stack, and thus is able to deal with a much broader range of configuration errors. At the same time, our tool provides a highly customizable interface that helps fully utilize users' domain knowledge, making the learning phase adaptive and effective. Results show that EnCore is effective in detecting both injected errors and known real world problems. In addition, it finds 37 new misconfigurations in 25 existing Amazon EC2 public images, as well as 24 new configuration problems in 22 images in a commercial private cloud. These previously undiscovered errors can cause problems in various aspects such as service unavailability and security issues.

While correctness standard is usually consistent across different platforms and thus can be learnt with a large data set, different systems usually need different configuration settings to achieve high performance according to their specific characteristics. Therefore only learning the configuration values is not enough. Previous works try to automatically select the optimal configuration settings by trying out the whole space. However it usually takes a long time. We significantly reduce the computation time by analyzing the correlation and constraints of performance settings from source code.

Chapter 1

Introduction

To adapt to different execution environment, and meet different requirements, software (especially complicated ones) usualy requires the users to configure it before running. Some examples of such configurations could be the directory to store the data files in a database system, the number of tasks the system is allowed to run in parallel in a webserver, and the maximum size of the file that could be uploaded by the users.

However, as software systems become more flexible and feature-rich, their configurations have become highly complicated. For example, MySQL, Apache httpd server, and Hadoop have over 300, 200, and 200 configuration entries respectively [2, 7, 4]. As a result, correctly and optimally configuring software systems to achieve high performance or even correctness, has become a highly complex task, and mannual configurations often introduce errors and sub-optimal settings [33, 58, 61]. Without a propriate helper, the complexity has severely affected the data center systems in terms of both reliability and performace.

1.1 Configuration Correctness

Due to the large amount of configurations the administrators need to deal with, the chance to make mistakes is very high. For example, a configuration error can be a non-existing directory set to be the storage path, or two values that break the intended

relations such as equality.

Studies show that configuration errors are both common and highly detrimental to business [62, 58, 40, 41]. For example, it is reported that misconfigurations are the second major cause of service-level failures at one of Google's main services, right after hardware failures [22]. Many recent system failures from Microsoft Azure, Amazon EC2, and Facebook, are also the consequence of misconfigurations, and affected millions of their users [49, 36].

The damages of misconfigurations come from two aspects: the lost business due to system unavailability, as well as the amount of cost spent to fix them. While the first one seems obvious [6, 30], the second one, althought not equally well recognized, also brings a signficant burden for a company. In one particular case of a large commercial corporation, misconfiguration has contributed to as many as 27% of the trouble tickets in the customer support database, wasting a large amount of development time [62]. The situation is exacerbated by the fact that many organizations impose security and performance policies for best practices. Configuration settings that are otherwise valid from a functional perspective may not conform to these policies, leading to security flaws or performance anomalies. Detecting such sub-optimal configurations is also highly desirable.

Most of the time people rely on the software logs to locate and fix the failures. However, it is reported that most of the errors caused by configuration settings do not directly pinpoint the corresponding configuration entries [58]. For example in Figure 1.1, while (b), (c), and (d) all have some loggings, but they do not show any hint of the configurations that are their root causes.

Some existing works try to address misconfiguration problem by helping the users to diagnose the failure [21, 19]. However, these approaches, as troubleshooting tools, need first have the failure happen, which means the damages are already done,

Related Config: *.htc/contentType* not defined
Root Cause: *.htc/contentType* should be “text/x-comp”

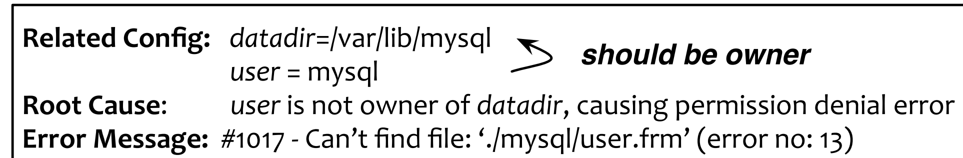
(a) Expected value not defined.

Related Config: *extension_dir*= “C:\Program Files\...\php\php_mysqli.dll”
Root Cause: *extension_dir* should be a directory, not regular file
Error Message: Call to undefined function mysql_connect()

(b) Wrong value causes modules not loaded.

Related Config: */selinux/enabled* = 1
Root Cause: SELinux prevents mysql from accessing files
Error Message: kernel: [4391754.436681] audit (1249498969.797:4): type= 1503 operation =” inode_permission" requested_mask="ra:: " denied_mask ="ra::" name="/var/log/mysql.err"

(c) SELinux disabled file accesses.



(d) Wrong file owner causes permission problem.

Figure 1.1. Examples of misconfigurations from real world. Note that while the first problem can be directly told from the missing value, it is not applicable to the later three.

and thus is often avoided as much as possible.

A promising approach for taming the configuration problem (or the configuration hell as described in [61]) is to automatically check a set of configuration settings for potential errors before deployment, just as one would normally do with application source code. However, most configuration files used today lack the rich structural and semantic information available in programming languages, which enable sophisticated analysis for errors [58, 56].

To overcome the limitation, researchers have tried to attack the problem of misconfiguration detection by learning, for each configuration entry, the common values used in a large collection of configurations (i.e. the training set), and flagging those values that are different from the common ones as potential misconfigurations [59, 58, 47]. They pioneered the approach of learning from other existing configured systems, and have shown its effectiveness. They typically have a genetic database of the configuration files. To check a target file, a diffing is performed with the existing ones, and statistical models or huristics are used to filter out the noise and find the suspicious values.

While proven useful in some scenarios, the potential of these tools [59, 58, 47] is greatly limited due to the simplistic treatment of each configuration setting as a string literal in isolation. Configuration settings bridge applications to their operating environment. Therefore, the diagnosis and remedation of the misconfigurations requires reasoning across both sides.

Figure 1.1 shows several real-world misconfigurations, neither of which can be detected with existing approaches. while the first problem can be directly told from the missing value, it is not applicable to the later three: (a)the value varies widely in the training set, (b) this entry does not appear in the software's configuration file, and (c) the values are very common in the training set that none are considered to be abnormal. The root causes of (b), (c) and (d) are hidden in either the environment factors (b and c),

or the correlation between two configuration entries (d).

In Figure 1.1(b), `extension_dir` specifies the location of the extension libraries. While it should be a directory, the user incorrectly supplied a regular file. Existing approaches that detect errors by relying on anomalies in the configuration values cannot detect this error because the value of `extension_dir` often varies across a set of samples and hence cannot be meaningfully considered as an anomaly [58]. However, by analyzing a set of sample values for this setting in the context of the executing system, one can detect that the value should be a directory and not a file. Figure 1.1(c) shows the example where the configuration entry is not for the software, but rather the whole system setting. Detecting it requires the knowledge of the environment instead of the software itself alone.

In the example shown in Figure 1.1(d), `datadir` points to the directory where MySQL stores the table data, and `user` specifies the system user id with which MySQL operates on the data. MySQL requires that the user `mysql` should own the `datadir`, while in this example this is not the case in this example. Existing methods that compare the values of these configuration settings across a set of samples cannot detect this error because detecting this error requires reasoning along two dimensions: (i) correlation between multiple configuration settings – correlating `user` and `datadir`, and (ii) validation of the configuration settings in the context of the system (environment) they are used – checking in the system if `datadir` is owned by user `mysql`.

In order to detect the misconfigurations that are not explicitly reflected in their text values, we need to broaden the scope of the analysis and look beyond single configuration settings. Based on our observations (described in Section 2.1) we propose to widen the analysis to include two important factors: (i) multiple configuration settings and (ii) the execution context or environment.

Cloud computing and related technologies such as virtual machine images have

made it possible to easily capture and analyze the execution environment of systems. This ability has enabled researchers to view systems as structured data [28]. Our approach embraces this view of *systems as data* and exploits it to extract the environment information relevant to configuration settings in a system.

1.2 Configuration Performance

Performance is critical to every software users. Especially for the online services, most of the time, it means the amount of profit they can make from the services [35, 12]. Therefore, maximizing performance is one of the major goals in software deployment, especially for server softwares, where the resources are always limited compared to the workloads.

The performance of such softwares heavily depends on appropriate configurations [16, 17, 23, 60]. For example, reports show that even when adjusting only one or two configuration parameters, the software performance can have a difference of 2 or 3 times [9].

Tuning the software performance is a non-trivial task that costs large amount of time, even for expert users [37, 24]. It is caused by three major factors: 1) The number of tunable configuration parameters of a server software is often huge [55]. For example, Mapreduce has over 100 parameters, while MySQL has over 300. 2) There are many possible values for each parameter, and 3) The parameters may have implicit dependencies on each other. For example, the `datadir` parameter in MySQL corresponds to an existing directory in the file system, and the value of `max_wal_senders` should be smaller than that of `max_connections` in PostgreSQL.

Automatic performance tuning is an ideal way to save the users from the long-lasting trial-and-error tuning cycles [17, 23, 24]. However, current automatic tuning tools usually assume that they already have the target set of parameters to tune as well as

their value range, and focus on selecting the optimal values from the given configuration parameters, and do not address the above-mentioned problems. This is largely because they treat the software as a black box [24], and thus the only way is to perform the trial cycles.

Treating the software as a black box significantly limits the capability of an automatic tuning tool in several ways: 1) it cannot perform the first step of selecting the relevant parameters from hundreds of configurable parameters, leaving the work to the users. Table 4.1 shows the number of parameters and the performance-related ones. Without the knowledge, the tuning phase may either waste large amount of time trying parameters that do not affect the performance, or still requires the experts to select the tuning candidates. 2) it does not adapt to the software evolution, where the meaning or even existence of the parameters may change and the old configuration rules may not apply. 3) The generated configurations may break the dependencies of the parameters, and may cause the malfunction of the software or wasting time on invalid configurations.

There are a few works that treat the software as a white box [37, 17, 16]. However, they all focus on just one application which they know all the details in their execution, and the methods may not be applicable to the others.

1.3 Contributions

The contribution of this dissertation deals with the configuration correctness and performance problems that are related to software configurations. The misconfiguration detection is a general four-step methodology that enables the utilization of system context information and learning from peers to automatically analyze configuration files, to generate rules that are otherwise hidden from the users, and to apply them to the checking of new configuration files. Specifically, this methodology involves the following contributions:

1) With a learning set enriched with environment information, we automatically infer configuration entry types, based on both predefined and user-specified guidelines. The type inference serves two purposes: inferring type related rules, and providing the base for correlation rule inference. As far as we know, this is the first work to utilize the system context to help accurately infer the configuration entry types.

2) We seamlessly integrate the environment information to the configuration files by extending the eligible configuration entries that carry context semantics. The eligibility of a configuration entry is decided by its inferred type, depending on which various possible extensions are appended to the original entry. For example, we add an extension of *permission* to entries that are of type *FilePath*. The benefits of the integration are twofold: 1) the correctness of a single configuration entry can be examined from multiple dimensions instead of only its value, and 2) the extended information can be further utilized in later correlation detection. As for as we know, this is the first work to integrate system information into the original configuration files for correctness checking.

3) We build a highly extensible data analysis framework to learn the correlation rules from the training set. In addition to a predefined set of templates for the learning process, it provides a simple and extensible grammar that allows users to freely specify any other learning templates. It effectively incorporates users' domain knowledge into the learning process to significantly improve learning efficiency as well as accuracy. The grammar allows users to select the entries efficiently, to specify the desired combination actions, and to choose the comparison methods. The configuration entries are specified based on their types, which carry rich semantic information. Compared to selecting the attributes one at a time, it is much more efficient and easier to reason. The results are further filtered with various metrics.

4) We provide a configuration file checker. It utilizes the type, correlation, and value information learnt from the previous steps, and apply them to the checking of

misconfigurations in the new systems. Similar to the learning step, it also takes the environment information in the target system into consideration, and compares them with the training set.

5)We evaluate the prototype of our work with three server applications: Apache httpd server, MySQL, and PHP, using images crawled from Amazon EC2 as the training set. The results show that it is effective in detecting both real world misconfigurations and injected ones. In addition, it detects 37 new misconfigurations in 25 EC2 public images, as well as 24 new misconfiguration in 22 images in an IT company's private cloud. These previously undiscovered problems could harm the systems in different ways.

6) We describe a framework that generates performance related configuration parameters and their corresponding constraints, by analyzing the software source code. These information is to be used by any existing automatic performance tuning tools to improve their effectiveness, or to users to have a better knowledge of how the parameters may impact the performance.

Chapter 1, in part, is a reprint of the material as it appears in 19th International Conference on Architectural Support for Programming Languages and Operating Systems. Jiaqi Zhang, Lakshminarayanan Renganarayana, Xiaolan Zhang, Niyu Ge, Vasanth Bala, Tianyin Xu, and Yuanyuan Zhou. The dissertation/thesis author was the primary investigator and author of this paper.

Chapter 2

Findings and Design Principles

In this section, we elaborate the findings that motivate and influence the design choices of *EnCore* and *iOpt* . Our design principles derive from observations made in two exercises: the study of configuration entries, and our experience in applying off-the-shelf data mining techniques for misconfiguration detection.

2.1 Characteristics of Configuration Parameters

To understand the use and importance of the environment information and correlations between configuration entries, we manually studied the configuration entries in 4 representative server applications – Apache, MySQL, PHP, and sshd. In this section, we describe our key observations on the correlation characteristics, which validate the prevalence of the phenomena, and motivates *EnCore* . Together with the examples in Figure 1.1, they show the benefits of exploiting the environment and correlation in misconfiguration detection.

Finding 1: Configuration entries are not isolated, but have relation to the execution environment.

In our study, many configuration entries connect the application functionalities to its execution environment, and the values of these configuration entries are associated

Table 2.1. Number (percentage) of configuration parameters that are associated with environment and correlations. Apache includes the entries of two main modules: core and mpm; PHP includes the core entries; MySQL's entries are randomly sampled.

Apps	Total Studied	Env-Related	Correlated
Apache	94	29 (31%)	42 (46%)
MySQL	113	19 (17%)	31 (27%)
PHP	53	16 (30%)	20 (38%)
sshd	57	12 (21%)	29 (51%)

with the properties of the environment. In other words, these configuration values should not be seen as arbitrary strings. Rather, they reflects the system properties, and have rich semantic information. *This is the key characteristic that distinguishes them from other program variables.*

Although neglected by most existing work, the environment information is critical for detecting a range of configuration errors. Taking the example in Figure 1.1(b), if a misconfiguration detector catches the fact that *extension_dir* should be a directory in the file system, it can easily identify the one in Figure 1.1(b) as an anomaly since it is not a directory.

Table 2.1 shows the statistics of configuration entries whose values refer to system environment objects in the studied applications. For sshd, we studied all its configuration settings. For Apache httpd server and PHP, we selected all the settings for the main module. We also randomly selected over a hundred entries from MySQL. We studied a total of 317 configuration entries. The second column shows the number of entries that are related to the environment, and the third column shows the number of entries that correlate with each other. From the data, it is obvious that a significant portion, more than 20% of the configuration entries, point to environment objects. With the abundant environment information we can extract from systems, there is great potential

for exploiting them in misconfiguration detection.

Finding 2: Many configuration entries are correlated.

From Figure 1.1(b), we have seen that the setting of one configuration entry depends on the setting of the other entries, or the environment objects. Thus, the correlation information is essential for correctly setting these configuration entries and for detecting errors. Though some types of correlations (e.g., the equality and inequality) can be observed in the textual values, many correlations are often indirect or even implicit. For example, in Figure 1.1(b), the correlation between `datadir` and `user` goes beyond the textual values and requires one to understand the semantics of the two entries.

As shown in Table 2.1, around one third to half of the configuration entries have correlation with each other. This indicates that the correlation among configuration entries should be an important component in a misconfiguration detection tool. Unfortunately, impaired by treating configuration values as textual strings, the state-of-the-art detection tools are limited and unable to leverage these correlation information.

2.2 Challenges in Applying Data Mining

Since our goal is to extract the configuration correlations, either with the executing environment or between different configuration entries, the straightforward idea is to apply data mining methods to learn the association rules, as proposed in [47] and shown to be effective in other works [63].

We started out by trying to use off-the-shelf data mining methods (association rule mining) on our configuration data set. Among the various methods, association rule detection is suitable for our purpose of Since our target is to learning the configuration correlations. instead of classifying the systems, the suitable methods for us is the asso-

ciation rule detection methods. It is used to discover the deterministic relations between different attributes in the given data. In our case, they are all the values configuration entries may carry, as well as the integrated environment data. Assuming I is the set of the items that appear in the data set, the formal definition of an association rule is an implication of the form $X => Y$, where $X \subset I$, $Y \subset I$, and $X \cap Y = \emptyset$. It has two threashold to restrict the generated rules: *confidence c%* means $c\%$ of the instance of the data (configuration files, in our case) that contain X also contain Y, and *support s* means the proportion of the configuration files that have $X \cup Y$. Increasing s and c helps filter the noise and produce more accurate results. Therefore, in our case, we want to use the association rule detection to help identify the possible rules that if some of the attributes are specified as certain values, other attributes should be specified in certain ways.

We tried standard association rule mining algorithms including Apriori [46] and FP-Growth [31] provided by two widely used data mining tool sets – Weka [15] and RapidMiner [11]. They are commercial-strength tools used by both researchers and the industry. Weka is widely used as a data mining research platform that integrates many existing algorithms from the preprocessing of the data (such as value format transformation and filters) to the learning methods. It takes a csv file as input and generates classification results or learnt rules depending on the algorithms selected by the users. For each algorithm, it has its limited input format requirements as well as the possible parameters that could be set by the users. RapidMiner is a commercial-strength data mining software that is claimed to be deployed in thousands of applications in over 40 countries. Like Weka, it provides various different algorithms from the very front end that filters the data to the end where user selects the data mining algorithms. It provides a more elegant user interface that helps the users design and save the analysis flow, and preview the intermidiate results.

In this section, we describe our experience and findings of applying these data

mining methods on the configuration data and motivate the design choices of *EnCore*. In the discussion, we mainly use the results from FP-Growth as Apriori does not scale to large data sets [34, 48]).

Finding 3: The state-of-the-art mining algorithms are not scalable to handle the scale of configuration settings.

Configuration files (especially after augmented with system information) usually contain a large number of settings, which are turned into *attributes* in data mining algorithms. Table 2.2 shows the number of configuration settings (in terms of columns) in the studied software. "Original" is the number of attributes originated from the configuration files. "Augmented" is the number after environment infomation integration. "Binomial" is the number after conversion from nominal data to binomial. While the number of unique configuration entries are limited, the mining algorithms treat each occurrence of an entry as a different attribute. Further, the addition of environment information as additional attributes (described in Section 3.2) increases the total number of attributes. At the same time, Apriori and FP-Growth suffer from the boolean discretization problem [53] – before the mining process, the nominal attributes need to be discretized into boolean values, which causes a dramatic growth in the number of attributes.

Table 2.3 shows the execution time and size of the intermediate frequent item set for Apache, MySQL and PHP, where the number of settings range from 100 to 200+. With more than 200 attributes, some experiments are terminated with Out Of Memory (OOM) exception. The entries are randomly selected. The number of attributes refer to the scale involved in the mining after adding enviornment attributes and discretization. The experiments were carried on a server with 8 processors and 16GB of memory. Note that when the number of settings increase, both the execution time and the size of fre-

Table 2.2. The number of attributes generated using data mining methods in the studied software.

	Apache	MySQL	PHP
Original	5773	175	1672
Augmented	9853	555	1942
Binominal	12921	859	2374

quent item set increase exponentially, making it impossible to reason about the results or even finish the experiments.

Feature selection (and reduction) is a common technique used to attack the scalability problems of data mining methods. There are two approaches: (i) scheme-independent selection and (ii) wrapper methods that involve the machine learning methods themselves to decide the useful attributes. Neither of these two schemes can directly help reduce the number of attributes in our case. The first method requires the users to select attributes based on domain knowledge [18]. Although both Weka and RapidMiner provide interface for such selection, it is not possible for the users to choose from hundreds to thousands of configuration attributes without any clue on what might be a better combination. The second method, data mining techniques developed to wrap themselves in the attribute eliminating process [51, 25], are mostly useful for classification purposes and are not suited for the correlation learning.

In summary, off-the-shelf data mining techniques do not scale well to the large number of attributes in the configuration data enriched with environment information. We attack this scalability challenge in *EnCore* by using a type-based and template-guided approach to learning. As described in Section 3.3.1, *EnCore* effectively addresses the scalability problem by restricting the types of the involved attributes and limiting the rule types; it also expresses the various correlations by providing different learning templates.

Table 2.3. Time cost (in seconds) and size of frequent item set with different number of attributes.

entries	Apache			MySQL			PHP		
	attrs	time(s)	freq.	attrs	time(s)	freq.	attrs	time(s)	freq.
100	219	0.15	6K	217	0.13	13.9K	150	0.52	6K
150	436	1.6	173K	286	62	3.8M	235	3.8	542K
175	503	170	14M	315	358	10M	279	106	4.9M
200+	554	OOM	-	343+	OOM	-	336+	OOM	-

Finding 4: Frequent-item-sets style relations are not expressive enough to describe domain-specific correlations.

In the data mining community, correlations among different objects are usually described using frequent item sets or linear regression models. While they are sufficient in describing co-occurrence (things likely appearing together) and linear relationship of objects, they cannot capture the complex relations between configurations. For example, a “directory” configuration entry could be concatenated with another “file name” entry to form a “file path” entry; the file specified by the “file path” entry can be owned by the user specified by a “user name” entry. Clearly, these complex domain-specific correlations cannot be expressed by frequent item sets. This motivates the need for new techniques to learn the correlations among configuration settings. As described in Section 3.3.1, *EnCore* addresses this challenge via rule templates that capture the complex correlation patterns (not actual correlations, but patterns of correlations).

Chapter 2, in part, is a reprint of the material as it appears in 19th International Conference on Architectural Support for Programming Languages and Operating Systems. Jiaqi Zhang, Lakshminarayanan Renganarayana, Xiaolan Zhang, Niyu Ge, Vasanth Bala, Tianyin Xu, and Yuanyuan Zhou. The dissertation/thesis author was the primary investigator and author of this paper.

Chapter 3

Misconfiguration Detection with En-Core

3.1 System Architecture

Based on the above observations, we propose a framework and tool, called *En-Core* , which incorporates both system environment information and the correlations among multiple configuration entries to effectively and efficiently infer the best-practice rules from the the configured systems. These inferred rules are further applied by a configuration checker to detect configuration anomalies.

As depicted in Figure 3.1, *EnCore* has four major steps: data collecting, data assembling, rule inferencing, and anomaly detection. This section briefly describes each component.

3.1.1 Data Collector

The data collector gathers the necessary information from the training set (a set of configured systems). Its output is the raw data including all files relevant for analysis, as well as additional environment information in text format. Since the input is a set of systems, the collector works as a crawler that copies data from each of the them. It can be either a static file system reader that directly reads data from a virtual machine image,

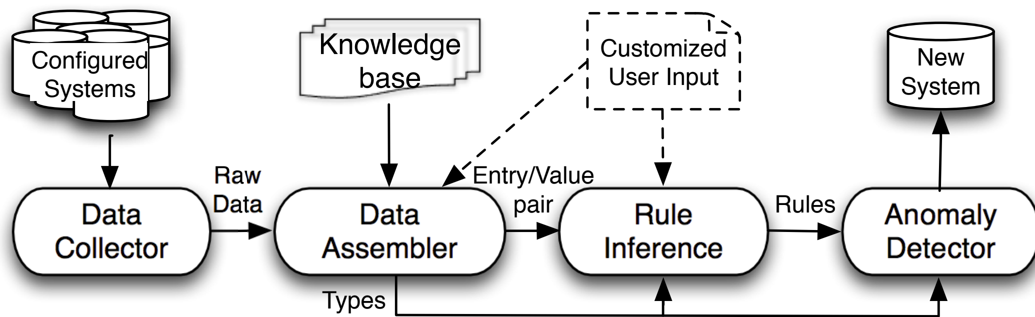


Figure 3.1. The architecture of *EnCore* . Both of the data assembler and the rule inference can be customized by users optionally.

or as in our case, a crawler that automatically copies data back from the target system.

Since we assume the users of *EnCore* are mainly administrators in an enterprise environment, or can utilize publicly available images such as EC2 like us, privacy is not of concern. But if needed, techniques such as FTN [57] can be used to alleviate the issue.

The data collector provides an interface for the users to specify any additional files to be included or commands to be executed during the crawling. By extending the crawler, it is easy to include any other environment factors such as environment variables and other files of interests.

3.1.2 Data Assembler

The data assembler first parses the collector's output and converts them (both configuration files and environment data) to uniform key-value pairs. It then infers the type of each configuration entry, which forms the foundation of the *EnCore* analysis framework, as all subsequent analyses incorporate the type information. Each configuration entry is then augmented with the additional environment information collected from the system. The assembler relies on a set of heuristics to infer the predefined types, but it also accepts an optional user input file that specifies heuristics to infer new types.

The details are described in Section 3.2.

After data assembly, the environment data and the original configuration entries are integrated together, and treated equally in the following components. Therefore we use “attribute” to refer to both of them in the following sections.

One critical step in data assembling is to extend the configurations with the environment information. , which is one of our key ideas. It happens after inferring the type of each configuration entry, which provides references for the environment data integration and is the base of the analysis framework.

3.1.3 Rule Generator

EnCore learns the best-practice configuration rules deployed in the training set by employing a variant of supervised learning. In order to address the challenges discussed in Section 2.2 and cater the learning process for the characteristics of configuration files, *EnCore* utilizes the rule templates, either predefined or user specified, to guide the learning process.

The templates specify the possible relationships (e.g. association rules or file ownership) among configuration entry types, not entry values. Thus, a small set of templates can cover a wide range of concrete rules. Section 5 shows that a total of 79 concrete rules are generated from the 11 predefined templates in 3 applications. They are aimed to capture the common possible correlations among the augmented configuration entries, such as the association rules or the file ownership relations, and are tailored to the concrete rules with specific configurations by learning from the training set. The semantic meaning of an example template is 'whether an entry of type *UserName* is always the owner of another entry of *FilePath*', 'whether an attribute of type *file path* is always a subdirectory of the other attribute of type *file path*'.

The template concept is similar to that of [29], but is more formalized with con-

cise grammar (as shown in Figure 3.6) to facilitate the users if they want to provide their own templates.

In our system, a template consists of four parts:

- The slots to be filled with concrete configuration names
- The type of each slot
- The combination of different slots
- The relationship between combinations of slots

An instance of the template is a concrete rule that has the slots filled with the entry names. For example, the following template

[A<FilePath>]=>[B<UserName>] *T1*

specifies that 'an entry of type *UserName* is the owner of another entry of type *FilePath*', where the types are inferred in the previous step. Based on the given templates and input key-value pairs, the rule generator iterates over all the possible combinations of attributes and selects concrete rules that best fit the templates. A concrete rule has the placeholders (e.g. A and B in *T1*) filled with concrete attribute names.

This step also prunes possible false rules with multiple filters. The output of the inferences are the rules that can be utilized by the anomaly detector. More details are in Section 3.3.

3.1.4 Anomaly Detector

The anomaly detector detects the rule violations in the configurations of the target systems. Its input is the rules, the type information, and the target systems. It outputs warnings whenever it finds an anomaly, such as a violation of a correlation rule, a wrong type, or a suspicious value. The results are ranked based on the type of the violation

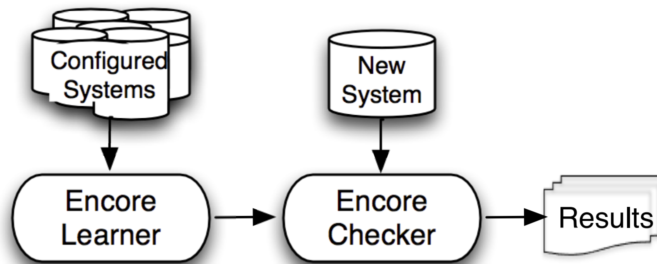


Figure 3.2. Use of EnCore.

and the underlying independent statistical model. Since the checking and the learning are cleanly separated, the learned rules can be reused to check different systems. More details are in Section 3.4.

3.1.5 Where and how to use the proposed tool

EnCore can be used either after a new system is configured, or after a failure happens. As shown in Figure 3.2, the user inputs the training set to *EnCore* together with the system to be checked. *EnCore* reports warnings that pinpoint to the potential problematic configuration entries. The usage scenario is similar to that of other misconfiguration detection tools [58, 59], except that the user can also optionally provide *EnCore* with additional types and templates that are specific to the user's environment. to consider, depending on how thoroughly the user wants *EnCore* to check the system.

3.2 Data Assembler

The data assembler takes the raw system files including the target configuration files, as well as the system environment information such as metadata of files in the file system, system configurations, and hardware specifications. It parses these files, infers the types of configuration entries, and augments each configuration entry with the environment information according to its type. The output is a set of well formed

key-value pairs. Figure 3.3 summarizes the process.

3.2.1 Parsing Configuration Files

The data assembler first converts the configuration files from application-specific format to uniform key-value pairs. We build the parser on top of Augeas [39], a general configuration file parser supporting various software configuration formats.

It takes the text configuration file as input, and outputs a tree structure, where each node represents a configuration entry, with the configured value being its value. Each level of the tree structure in the key corresponds to a section in the configuration file. If there are multiple occurances of one entry, they are assigned with a serialized number. By recursively iterating the tree structure, it is trivial to get a set of key-value pair, where the key is the XPath leading to the node. For example, in Apache, the value of a *Directory* entry in the third *IfModule* section is named */IfModule[3]/Directory/arg*. Entries that carry multiple values are automatically separated into multiple nodes.

The use of Augeas largely eliminates the requirements for the users to parse the configuration files. However, in case the configurations of certain software are not covered, Augeas provides an extensible interface to import other parsers, enabling users to easily import their own configuration parser into *EnCore* . Some software vendors also provide tools for the parsing process [13]), and can also be utilized by *EnCore* as long as the output format complies to *EnCore* 's requirements.

After parsing the configuration files, the assembler stores and organizes all the data in a *.csv* file. Each column represents a structured configuration entry generated by the parser, and each row represents the values of all the entries in a system.

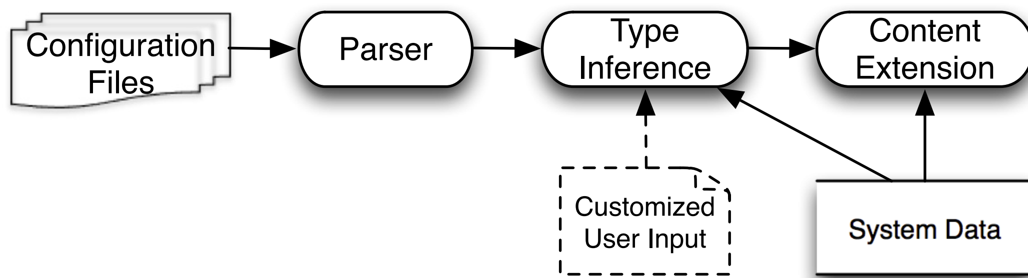


Figure 3.3. Data assembling. In addition to the configuration files, the assembler also takes the system environment data and users' input for type inference and augmenting the configuration entries.

3.2.2 Inferring Configuration Entry Types

EnCore relies on the type information of each configuration entry for further environment data integration and correlation detection. Similar to previous work on type inferring [44], *EnCore* needs to have the domain knowledge of what the types are and how to determine them. Unlike inferring the types from source code as in [44], where the semantics of system and library calls are required, *EnCore* needs the knowledge of data format of each type.

Inferring certain data types might involve heavy-weight checking, and can be computationally expensive. For example, to determine a `FilePath` type, we have to check whether the file value exists in the file system or not. To mitigate the cost, the type inference employed in *EnCore* is a novel two-step process that leverages both syntactic patterns of data values as well as the environment information of the system. The first step performs *syntactic matching*, making a crude *guess* at the type of the entry using a predefined syntactic pattern. For example, any string that contains a slash is a potential `FilePath` type. This step is followed by a heavy-weight *semantic verification* that validates the type by checking the corresponding external resources (such as the file system). For example, if an entry value is classified as a `FilePath` type by the first

Table 3.1. Predefined type and inference methods. Due to space limit, we show here simplified regular expressions used in syntactic matching. More complicated expressions are used for other values, e.g., IPv6 address, which is not shown in the table.

Types	Syntactic	Semantic
FilePath	/.+(/.+)*	File System
UserName	[a-zA-Z][a-zA-Z0-9_]*	/etc/passwd
GroupName	[a-zA-Z][a-zA-Z0-9_]*	/etc/group
IPAdress	[\d]{1,3}(.[\d]{1,3}){3}	N/A
PortNumber	[\d]+	/etc/services
FileName	[\w _-]+.[\w _-]+	File System
Number	[0-9]+[.0-9]*	N/A
URL	[a-z]+://.*	N/A
PartialFilePath	/?.+/(.+)*	File System
MIME Types	[\w/-.]+	IANA [5]
Charset	[\w]+	IANA
Language	[a-zA-Z]{2}	ISO 639-1
Size	[\d]+[KMGT]	N/A
Boolean	Values Set	N/A
String	N/A	N/A

step, the verification searches the full file system meta-data to validate the existence of the path in the file system. The first step prunes away most of the improbable types , making the inference efficient; the second step guarantees the inference accuracy. The combination of the syntactic matching and heavy-weight semantic verification proves to be both effective and accurate (see Section 5.2). Table 3.1 shows the details of each step and default types *EnCore* infers.

Table 3.1 shows the concrete types *EnCore* infers and how they are determined. The first step is to use the simple heuristics such as regular expressions to determine the possible type, and then the type is validated using the source listed in the 'Verification'

column. For example, a type of FilePath is first recognized by matching the regular expression. If matched, the system further consults the file system to decide whether it is a real file path in the system. A second example is GroupName. After matching the regular expression, the /etc/group file is checked to determine whether it is really a valid group name in the system. Note that the validation is optional as some of the types are self-evident (e.g. Boolean), and sometimes system information is not available for validation.

EnCore is able to infer most configuration entry types in the taxonomy of [44]. Certain types such as `TimeInterval` and `Count`, however, are not easily distinguishable, due to the identical syntactic patterns and lack of verification methods. Since our purpose of type inference is to detect type violations and to provide a better foundation for correlation detection instead of precise type analysis, we consider this limitation acceptable – missing this information means possible lower rule detection efficiency as it affects the effectiveness of type-based attribute selection (see Section 3.3), but does not affect the effectiveness. As described in Section 3.3, we use other ways to accelerate the detection process.

3.2.3 Environment Information Integration

One of the essential ideas of *EnCore* is to enrich the original data with additional environment information, and take them into consideration in the analysis. *EnCore* selects types inferred in Section 3.2.2 that carry system semantics, and augments them by attaching new attributes that represent the properties of each type. For example, an entry type of `FilePath` can be augmented with a new attribute that tells whether it is a directory or a regular file. We call these attributes *augmented attributes*. Certain environment data that is independent of the configuration entries are also collected, such as the system hardware specification.

Table 3.2. Default augmented environment information.

(a) Augmented attributes for eligible types by default. For each configuration entry type, *EnCore* augments it with attributes that reflect system context. Each augmented type is shown with an example entry, the information source in the system (we call it reference) used to compute the augmented values, and the values of the example augmented attributes. Each augmented attribute is assigned with a type.

AttrType/Example/Ref.	Augmented Attributes	Type	Value
FilePath datadir=/usr/data File System	datadir.owner	UserName	mysql
	datadir.group	GroupName	mysql
	datadir.type	Enum	dir
	datadir.permission	Permission	664
	datadir.contents	String	dirDes
	datadir.hasDir	Boolean	True
	datadir.hasSymLink	Boolean	True
IPAddress AllowFrom=10.0.1.1 RFC 1918, RFC 4193	AllowFrom.Local	Boolean	True
	AllowFrom.IPv6	Boolean	False
	AllowFrom.AnyAddr	Boolean	False
UserName user=mysql /etc/group	user.isRootGroup	Boolean	False
	user.isAdmin	Boolean	False
	user.isGroup	GroupName	mysql

(b) Augmented attributes for environment info by default. The attributes are gathered from corresponding files or commands. *EnCore* can be easily customized to consider more data.

Env Type	Augmented Attributes
Sys Config	Sys.IPAddress, Sys.HostName, Sys.FSType, Sys.Users
OS Related	OS.DistName, OS.Version, OS.SEStatus
Hw Spec	CPU.Threads, CPU.Freq, MemSize, HDD.AvailSpace

Table 3.2a shows the types that *EnCore* augments by default. For example, for each entry of type `FilePath`, we attach seven attributes: the owner and group information, whether it's a directory or file, the permission associated with the object, and if

it is a directory, the contents of the directory, whether it contains sub directories, and whether it contains symbolic links. This information is retrieved from the file system meta-data collected by the collector. Different types have different attributes, whose values are determined using different sources of information (see Table 3.2a.)

As shown in the examples in Table 3.2a, the augmented attributes are directly appended to the original entry names with a dot separator. Table 3.2b shows the environment information that is independent of the configuration files. These attributes are appended to the existing csv file as additional columns, and are treated equally as other attributes in the rule inference process.

As a general data analysis framework, *EnCore* provides the opportunities for the users to use their domain knowledge to infer new types, or to enhance the existing type inference methods. The new types information is placed into a file that is read into *EnCore* before the inferencing step starts. To specify a new type, the user provides 1) the name of the new type, 2) the hints to infer the new type, and optionally, 3) the verification methods. Environment information is also provided to the users to facilitate the inference methods. The users can also override the inference or validation methods of the existing types. Section 3.3.3 details how *EnCore* can be customized to infer new types and to include new attributes.

3.3 Rules Inference

With the extended configuration data as the training set, *EnCore* infers rules using a template-based method, either the predefined templates or those defined by the users. The inferred rules are then written to a file with detailed description of the attributes involved and the relation type, so that they can be used to perform the checking against the target systems.

3.3.1 Template-Guided Inference Based on Types

To overcome the challenges faced by directly applying data mining techniques (Section 2.2), the rule inference in *EnCore* adopts a template-guided approach. A template aims at capturing the common correlations among the configuration entries and system information, such as the association rules or the file ownership relations in Figure 1.1(b). The templates are tailored to the concrete rules with specific configurations by learning from the training set.

Unlike Lint-like checkers [3, 1] which require hand-written rules, *EnCore* only needs a guidance on the type of rules that the users are interested in. It infers concrete rules automatically from the training set. For example, the user can specify a template for *EnCore* to learn the association rules among three boolean configuration entries; then *EnCore* checks whether rules of this type exist in the training set, as well as the involving configuration entries.

Figure 3.4 shows three example templates, their meanings, and the concrete rules inferred from them in different softwares. For instance, the template in Figure 3.4(a) defines the possible ownership relationship between two entries. Learning from the system images with MySQL, *EnCore* infers the concrete rule that describes the ownership relation between entries *DataDir* and *User*, which is used to identify the misconfiguration described in Figure 1.1(b).

Why use templates? The use of templates brings three benefits to address the challenges faced by direct data mining. 1) It effectively describes the possible types of correlations beyond frequent item set relation; 2) It avoids wasting the computation and resource on patterns that are not likely to exist in the configuration entries; and 3) It makes the learning process "*extensible*": the more templates are used, the larger coverage of possible rules is achieved. Note that the templates can be easily customized,

```
Template: [A<FilePath>] => [B<UserName>]
   Rule:  DataDir => user                                   //MySQL
```

(a) Rule Template: an entry B of type UserName should be the owner of an entry A of type FilePath

```
Template: [A<FilePath>]+[B<FilePath>] => <FilePath>
   Rule:  ServerRoot + LoadModule/arg2 => <FilePath>  //Apache
```

(b) Rule Template: the concatenation of two entries (A of type FilePath and B of type FilePath) forms a file path

```
Template: [A<Size>] < [B<Size>]
   Rule:  upload_max_filesize < post_max_size               //PHP
```

(c) Rule Template: an entry A of type Size should be smaller than an entry B of type Size

Figure 3.4. **Examples of templates and the concrete rules generated from them.**

as discussed in Section 3.3.3.

The template specification uses data type information to restrict the eligible configuration entries. For example, in Figure 3.4(a), *B* needs to be of type `UserName`, which means when the learning process tries to instantiate the template, it only fills in *B* with the attributes of type `UserName`.

The type information provides an intuitive and effective way of attribute selection, which is critical to solve the scalability problem in the learning phase. In addition, it is a natural base for associating environment information. For example, as described in Section 3.2, the file permission data are only meaningful to a configuration entry of type `FilePath`.

EnCore provides several predefined templates that make it readily usable. They are shown in Table 3.3 with their descriptions. Our evaluations are based on these templates.

Rule Inference Process. With the type-based templates, *EnCore* learns rules

from the training set. Figure 3.5 summarizes the workflow of the learning process. After parsing the customization file, the templates and operators are input to the analysis engine , which then performs the detection of the templates one-by-one. For each template, it first searches for the operator evaluation function that is suitable for the given types. The search is done in the order of customized ones, predefined ones, and the default one that is used for String. If the operator is not found in the customization file, the predefined operators are searched. If still not found, the operators that are used for String is selected.

Table 3.3. Predefined templates with description. The templates are created based on the common correlations of configuration settings. The operators carry different meanings for different types, and can be overridden by users' customization.

Template	Description
[A<AnyTypeA>] == [B<AnyTypeA>]	An entry should be equal to another entry of same type
[A<AnyTypeA>] = [B<AnyTypeA>]	One instance of an entry should equal to at least one instance of another entry of same type
[A<ExtBoolean>] → [B<Boolean>]	An extended boolean indicates a boolean entry whose extended attribute has boolean value
[A<IPAddress>] < [B<IPAddress>]	An entry of IPAddress is a subnet of another entry
[A<FilePath>]+[B<FileName>]=><FilePath>	Concatenation of a file path entry with a partial file path entry forms a full file path
[A<String>] < [B<String>]	An entry is substring of another entry
[A<UserName>]<[B<GroupName>]	The user name belongs to the group name
[A<FilePath>] ! = [B<UserName>]	The file path is not accessible by the user specified in the entry
[A<FilePath>] => [B<UserName>]	The entry of UserName is the owner of the file path specified in the entry A
[A<Number>] < [B<Number>]	The number in one entry is less then that of the other entry
[A<Size>] < [B<Size>]	The size in one entry is smaller then that of the other entry

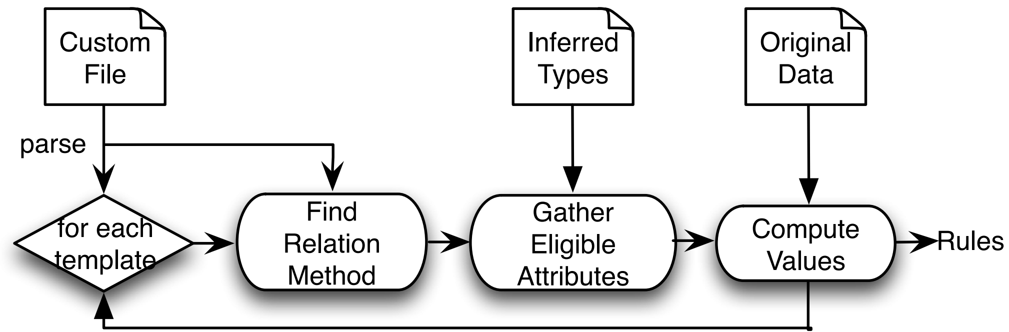


Figure 3.5. Workflow of rule inference.

In *EnCore* , each correlation is associated with a validation method that determines whether the correlation holds or not. The validation methods are identified by the relation operators(such as $<$ and ==) together with the types of the participating attribute placeholders (such as `A` and `B` in Figure 3.4; we discuss the correlation interface in detail in Section 3.3.3 when introducing how to customize correlations).

For each template, *EnCore* tries to instantiate the template by replacing the placeholders with eligible attributes that match the data type specified in the template. *EnCore* iterates over every possible instance of the template and checks whether it is valid using the validation method. If the correlation is valid, *EnCore* regards it as a rule candidate, which would further go into the filtering(Section 3.3.2). Note that this process is highly parallelizable because there is zero state sharing between each instance computation. As a result, we implement *EnCore* as a multi-process program to achieve high performance.

3.3.2 Rule Filtering

As accurately as the templates may be defined, like any data-driven method, there are false positives in the resulting rules. *EnCore* use three metrics to filter them. The first two metrics are *support* and *confidence*, which are typical metrics used in association rule detection. *Support* means how many times the frequent item set (in our

context, the attributes involved in the rules) occurrs in the data set, and *confidence* means the percentage that the rule is valid.

Configuration entries have a unique characteristic: values of certain attributes are stable in many systems. For example, the warning level in PHP has consistently been set to 10 in our training set. If an entry seldomly changes, the values it carries are not interesting, and the rules involving this entry are likely to be noise. To take this into account, we use a third metric: *entropy* [50]. It measures the diversity of the dataset: its value increases when more diverse values are seen for a given entry. The value of entropy is defined as

$$H = -\sum_{i=1}^{n} p_i \ln p_i, \qquad p_i = N_i/N,$$

where n is the number of different values, p_i is the probability of taking the ith value, N is the times this entry appears in the training set, and N_i is the times it is assigned with ith value. We set a threshold value for each of the three metrics. The threshold for the entropy is denoted by H_t. When there are two values, each having a probablility of of 90% and 10% respectively, it is set to $H_t = 0.325$. For an attribute to be included, it needs to have $H > H_t$, meaning it is more diverse than the minimum threshold. For a rule to be included, all the involved attributes need to be included.

Some other works use other metrics. For example, [47] uses Google's search results to measure how strong two entries are related. It makes sense when the entries are from different components. However, it is often difficult if they are from the same software, as they are likely to appear together on the same page.

3.3.3 Customization

EnCore is a fully customizable framework. Users can extend it with different levels of customization using different interfaces. Specifically, users can 1) specify new templates to infer new types of rules; 2) define new types in addition to the default types;

```
Template := Expression '--' Confidence

ExtendedSlot := Slot | Slot '::' EnvVar | NULL

Slot := '[' SlotName Type ']'

SlotName := CapitalLetter | DEF.Letter

Type := '<' DefinedTypes '>'

DefinedTypes := 'Types.AnyType'|
                'Types.FilePath' |
                'Types.PartialFilePath' |
                'Types.Number' |
                'Types.UserName' |
                'Types.GroupName' |
                 .... |
                 CustomTypes

Operator := ' == ' | ' != ' | ' ~= ' | ... |
            CustomOperators

EnvVar := 'Env_FS_Type' |
          'Env_FS_Owner' |
          'Env_FS_Permission' | ...

Expression := Generating | Relation | Concat

Generating := Concat '=>' Type

Relation := Concat Operator Concat '|' Type':'Type

Concat := ExtendedSlot |
          ExtendedSlot '+' Concat

Confidence:= [1-9][0-9]*'%'
```

Figure 3.6. Grammar of template specification.

and 3) tag new environment information for the environment data integration.

Template definition is the core of the customizable rule generation. It composes of two major parts:the *slots* (the capitalized letter and the type in square brackets in Figure: 3.4) and the *relations* (the plus and arrow symbols in Figure: 3.4). A slot is a place holder to be filled in the rule detection process, and has two parts: the name and the type.

The name will be filled with the concrete entry name after a rule is found; while the type is usually specified with a concrete type name by the users, and is used by the rule detector to select the eligible attributes.

There are also two types of relations. The first relation defines how different attributes can be aggregated. It can be the simple concatenation of strings, the algorithmatic addition, or any other aggregation methods that users want to specify. It is a generalization of the 'logical and' operation in association rules. The second type is a comparison operator, which defines how values are compared such as 'greater than' or 'equal to'. It can be viewed as a generalization to the definition of the 'implication' in association rule.

As shown in the example templates in Figure 3.4, a *name-type* pair in a square bracket forms a complete slot. The name slots are filled with place holders names such as 'A'. If the place holders have the same name, the entries to be filled in the rule are the same. Users can also specify the name slot with a concrete entry name by appending the 'DEF.' prefix. The names in the angle brackets specify the type of the slot, starting with a 'Types.' prefix. It also allows users to use 'Types.AnyType' when all the types should be considered.

The comparison relation operator divides a template into two parts. The left part is always a combination of slots. Depending on what can appear on the right side, there are two kinds of relation templates: *generating template*, and *comparison template*.

```
$$TypeDeclaration
      <NewType>
$$TypeInference
       <NewType> (value): { return True }
$$TypeValidation
       <NewType> (value):{ return True }
$$TypeAugmentDeclaration
        <NewType>.Group
$$TypeAugment
        <NewType>.Group (value):{ return '' }
$$TypeOperator
        <NewType>:<NewType> Operator '<' (v1,v2): { return True }
$$Template
        [A<Types.NewType>] < [B<Types.NewType>] -- 90%
```

Figure 3.7. Format of the customization file.

Generating template has a type specified on the right side, meaning the aggregation of the attributes on the left side could produce some values of another type. The comparison template has another combination of attributes on the right, meaning the left combination relates to the right combination by comparison. Figure 3.4(a) specifies a template where the concatenation of an attribute of 'FilePath' and another one of 'PartialFilePath' may generate a value of type 'FilePath'. The '+' operator simply means string concatenation. Figure 3.4(b) is a template of a typical boolean value association rule where two boolean values may indicate another one. The '+' operator here means 'logical and'. The semantics of all the operators can be reloaded depending on the types that they operate on. *EnCore* provides a set of predefined operators with certain types.

Figure 3.6 shows the template grammar. Note that besides the expression for the relation, users are also allowed to specify the confidence for the rules, which is used to filter those that are unlikely to be true.

Customization File Format

EnCore uses a single file to gather all the user-customizable input. The file has seven sections, each specifying a customizable aspect of the system. Figure 3.7 shows the format of the file and some sample input. Note that *EnCore* provides predefined types, operators, and templates as described in Table 3.1 and Section 3.3.1. *Therefore the customization file is optional.*

As illustrated in Figure 3.7, each section's name is determined, and is prefixed with '$$' symbol. New type information is specified in the top 3 sections including its name declaration, how to infer them, and how to verify them. Users need to implement the hint and verification methods, which are written in Python currently. The methods are given the values to be inferred, and a bool value is returned to indicate whether the value is this type. While the verification method is optional, the hint function is required for each new type. In the case that an attribute can be inferred with multiple types, the priority is given to customized types over predefined ones, and to the order they appear in the customization file. The following two sections define the extra attribute extensions besides predefined ones in Table 3.2. The extensions are first declared, and followed by the methods to compute their values. The types to extend can be both predefined types or customized types declared in the first section.

The type operator section defines the operators used in the templates, including both aggregation and comparison ones.To declare an operator, users need to provide the types involved in the operators, as well as the operator symbol, which will be used in the templates. The system feeds the operator functions with values to be analyzed, and the system information which is under analysis. For the aggregation operators, the returned value should be a pair that contains both the value and its type.

The template section lets the users to specify the kinds of rules that they want to

try in the detection. There can be multiple templates, each in one line. The grammar of the template follows Figure 3.6. These templates will be attempted one-by-one after the predefined ones.

Environment Information for Customization

EnCore provides the users with the environment information already gathered by the framework in the crawling phase, in the form of global variables. They are accessible in any place in the customization file where users' program is needed. Table 3.4 shows the data structures accessible by the users, as well as the sources where the system gets the information. They are organized in either array lists or maps. For example, with the FS.FileList, the user can get all the file names (including their paths) in the system, and with FS.FileMetaMap, the file path is used as a key, and the meta information of the file can be retrieved. The contents of the data structures are refreshed with the data in the new system when processing different systems. They are also available when performing the checking of new image.

3.4 Anomaly Detector

With the learned rules, *EnCore* looks for potential anomalies in the target systems. It goes through the same data assembling process for the new system as in the learning phase, including parsing, type inferencing, and environment information integration. Then, it checks the following aspects of the target configuration and produces a ranked list of errors.

1. Entry Name Violation. Previous studies show that misspelling is an important source of misconfigurations [44]. Since we have the knowledge of all the already examined configuration entries, if the new system includes entries not seen before, it is likely a misspelled one. Note if the configuration file has nested sections, it is only prioritized if

Table 3.4. Data structures for accessing environment information. *Only available when collecting data from running instances; **The hardware specifications are not available for newly instantiated virtual machine images such as those from Amazon EC2.

Category	Data Structure	Sources
Files Data	FS.FileList	File System
	FS.FileMetaMap	
Account Info	Acct.UserList	/etc/passwd /etc/group
	Acct.GroupList	
	Acct.UserGroupMap	
Service	Service.Ports	/etc/services
	Service.PortServMap	
Env Variables*	Env.VarValueMap	env
Security	Sec.SELinux	/selinux/*
Hardware**	HW.Cores	/proc/*
	HW.Memory	
	HW.DiskSize	

the inner-most entry name is suspicious. The unseen combination of section nesting is given the lowest priority in ranking.

2. Correlation Violation. *EnCore* checks if the target system follows the correlation rules learned from the training set. Since each rule is an expression, the detector of *EnCore* evaluates the expression with the values from the target configuration and reports warnings when violation is found. The rule is ignored if the involved entries are absent in the target configuration file.

3. Data Type Violation. For each entry to be checked, the checker reads its type information inferred from the training set, and gets the corresponding syntactic matching function and semantic verification function. The two functions are used to match and verify the target configuration value. A type violation is reported if the verification or matching fails. Another possible way is to also infer the entry types of the new configuration file. However, since the detector has only one instance, it may produce inaccurate type inference results.

4. Suspicious Values. The detector compares the values of configuration entries from the new system with the values in the training set. It reports a warning if the new value is different from all the previous ones. When multiple entries have unseen values, we adopt the *Inverse Change Frequency* method [59] that gives higher ranks to entries with less diverse values in the training set. Note that the statistical method used here is orthogonal to how rules are learned in *EnCore* : other methods (e.g., those in PeerPressure) can also be adopted here. We chose this simple design and found it satisfy our need. In fact, *with more environment information integrated, more aspects of the configuration entries can be checked against suspicious values.* For example, the detection of the error in Figure 1.1(a) is directly attributed to the extended attribute of *extension_dir.type* – all the values in the training set have type *directory*, but the value in the target system has

type *regular file*. This warning is ranked much higher than other possible suspicious values since its value set in the training set has a cardinality of only 1.

3.5 Related Work

Misconfiguration troubleshooting and detection. There are generally two approaches to detect and troubleshooting misconfigurations: one is to learn from the peers' behavior and settings, and apply them to the target systems, and this is usually a black-box approach; the other is to analyze the code of the application to infer the reasons that lead to the incorrect behavior, i.e. a white-box approach. Both white-box and black-box approaches are used for the misconfiguration troubleshooting and detections.

PeerPressure [58], Strider [59], and [47] all utilize existing systems for configuration values comparison. Strider uses classified working/failing configurations, as well as heuristic based on changing frequency to shrink the suspicious entries set. PeerPressure advances it by deploying a statistic model and thus eliminates the need of manual labeling. [47] uses both frequency and googling results as ranking heuristics, and tries to discover the substring dependency between the entries across different components. FTN [57] focuses on addressing integrity and privacy issues when seeking for peer contents. AutoBash [54] and [20] learn from the peers' steps of problem solving, and apply them to the target systems. CODE [63] learns the configuration access pattern for anomaly detection. Our work utilizes the environment information and various entry correlations that were unexploited before, and is able to help discover a much broader types of misconfigurations.

ConfAID [21] and X-Ray [19] rerun the program in the given environment and use dynamic information flow tracking to find the possible root causes in the configuration file. The applications need to be rerun in the given environment, and with the failure point, they help track back to the place where the error begins to propagate, thus decide

which configuration entry may have caused the problem. [43] precomputes the mapping between source code lines and the corresponding configuration options using program analysis, and when the software crashes with source code information, users could use them as a hint to search for the database and find the root cause in the configuration file. *EnCore* both help and complement troubleshooting works, as detecting the errors is the first step to guard against misconfigurations.

Configuration Entry Analysis. [44] analyzes source code to infer configuration entry types.*EnCore* does not need the source code, and also leverages the information for more efficientn misconfiguration detection.

Configuration Testing. ConfErr [38] and KLEE [26] generate both realistic and high-coverage test cases to test the softwares' robustness against misconfiguration. Both testing and detection reduce users time to deal with configuration error. Besides, *EnCore* benefits the testing work by exploiting new opportunities for error injection such as environment errors, and the correlation rule violations, and help with more targeted injection.

Chapter 3, in full, is a reprint of the material as it appears in 19th International Conference on Architectural Support for Programming Languages and Operating Systems. Jiaqi Zhang, Lakshminarayanan Renganarayana, Xiaolan Zhang, Niyu Ge, Vasanth Bala, Tianyin Xu, and Yuanyuan Zhou. The dissertation/thesis author was the primary investigator and author of this paper.

Chapter 4

Accelerating Automatic Performance Tuning with iOpt

4.1 Performance Impact of Software Coniguration Settings

Different from *EnCore* , which detects correctness issues in systems, *iOpt* focuses on the configuration tuning for the performance optimizations. Studies show that configurations impact heavily on how the system performs in a certain system. For example, tuning two configuration settings in Hadoop could result in two times performance difference [16].

Figure 4.1 shows our experimental results of Hadoop MapReduce, with 6 different set of configuration settings. The testing environment consists of 1 master node and 10 slave nodes. he job is a 10GB terasort which we split into 20 map tasks and 20 reduce tasks by default. We also set the block size on hdfs to 512M. Each node is IBM RC2 virtual machine with 6 vCPUs and 32GB memory.

We configured 6 different settings. The first set is the default configurations from MapReduce. With all default hadoop configuration, the job finishes around 392s. The total number of records in the job is 107374128, and the number of spilled records is about 3 times of that. In the second set, we adjust two parameters and set to "mapred.child.java.

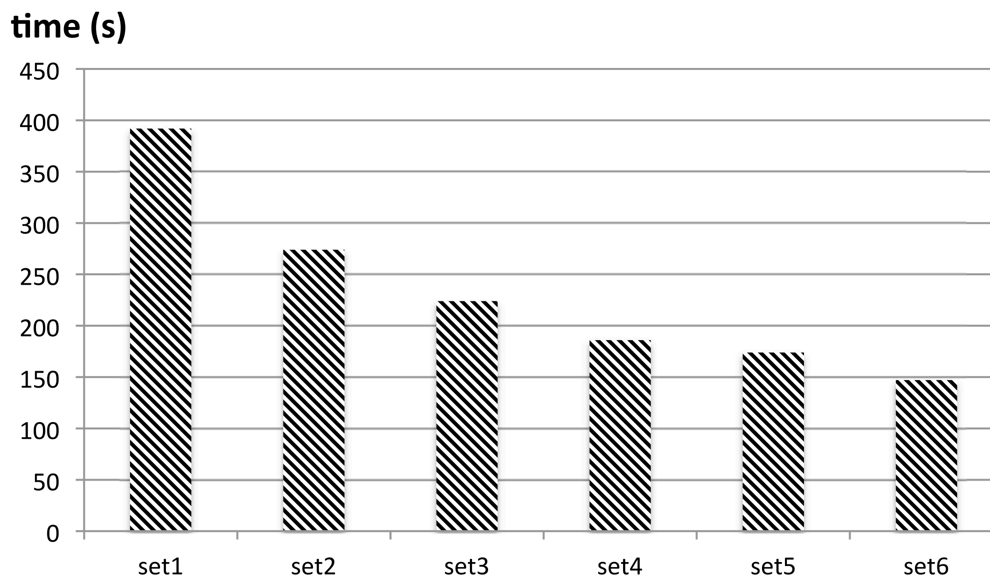
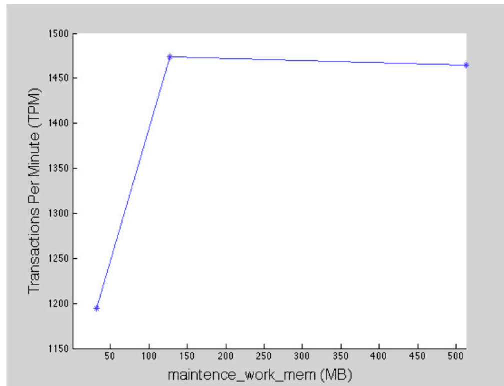
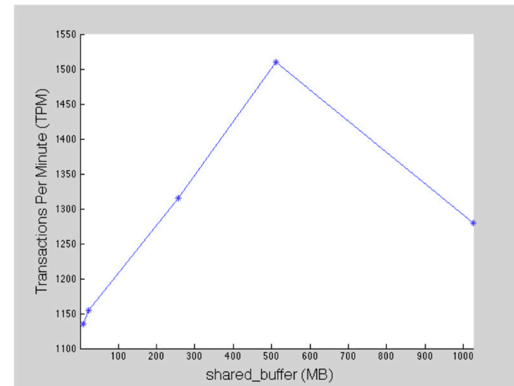


Figure 4.1. Configurations impact on Hadoop performance. With the changes of configuration settings, Hadoop has varied performance.

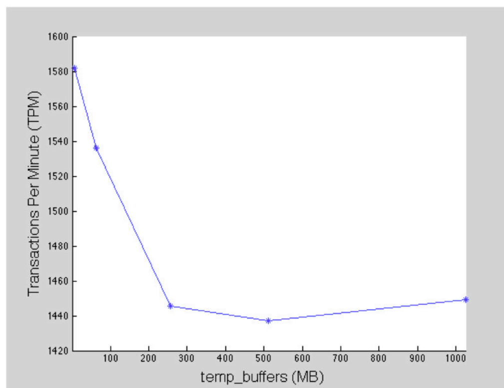
opts = 16G io.sort.mb = 2046M", the execution time now reduces to about 274s. When further setting "io.sort.record.percent = 0.15", the execution time is now around 224s, the number of spilled records now is just 2x of the total record number as we avoid spilling in the map phase. In the forth set, we set "mapred.inmem.merge.threshold = 0" and "mapred.job.reduce.input.buffer.percent = 1.0", the execution time reduces to around 186s as we now also avoid spilling in the reduce phase. When setting "io.buffer.size" to 128k, we further reduce the execution time to 174s. (the default io buffer size is just 4k, however, it is a cluster parameter rather than a map-reduce one). Finally, if we increase the map slots to 4 and reduce slots to 2 (default is 2 map slots and 1 reduce slots) per slave node to fully leverage the 6 vCPUs, and split the job into 40 map tasks and 20 reduce tasks, we can reduce the execution time to about 147s. The above resulting setting may not be the absolute optimal, but it should be fairly close to that. For really large jobs, further experiments show that a couple of other parameters



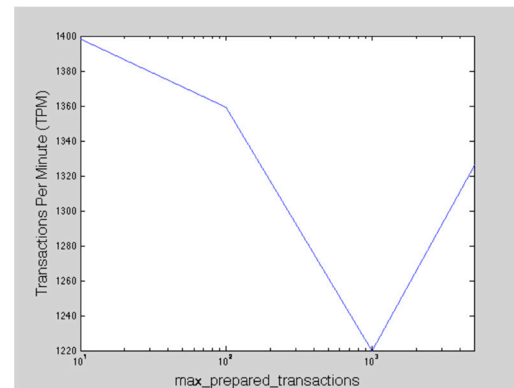
(a) performance with maintence_work_mem



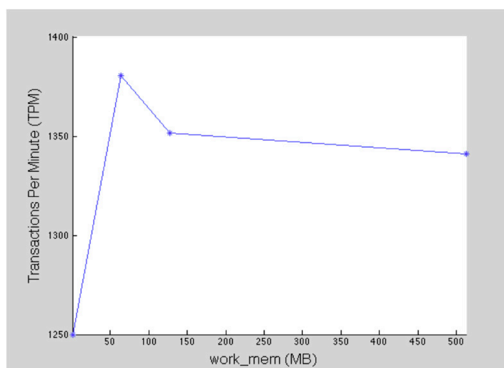
(b) performance with shared_buffer



(c) performance with temp_buffers



(d) performance with max_prepared_transactions



(e) performance with work_mem

Figure 4.2. Configurations impact on Hadoop performance. Performance variation with the change of different single configuration parameters in PostgreSQL.

are also important, such as ”mapred.map.tasks” and ”mapred.reduce.tasks” (the number of map/reduce tasks) and ”mapred.reduce.slowstart.completed.maps” (how long should reduce tasks wait after map tasks start).

Figure 4.2 shows the performance change when changing different performance-related configuration parameters. The experiment is carried on a single node machine. It shows that for PostgreSQL, even with only single paramter change, the performance can be improved by 10% to 36%.

4.2 Helping Automatic Performance Tuning

Automatic performance tuning tries to involve as less human efforts as possible in the tuning process, saving the administrators from the manual burdens. It usually goes through a fail-and-try cycle: adjust a paramter, run the workload, measure the performance. The process continues until an optimal setting is found [42, 27].

However, this tuning process itself takes a long time to complete. This is usually because there are often a large number of tunable parameters for each application (as described in the previous chapters), and the set of legal values of each parameter is large as well [52, 17, 23]. For example, the configurations that specify the memory usage in Hadoop usually takes the input as a percentage (from 0 to 1.0), thus it could take many rounds of parameter adjustments and workload running profiling to reach to an optimal configuration.

4.2.1 Existing Approaches to Help Performance Tuning

A traditional way of finding the optimal configuration parameter settings is to use a recursive search algorithm. When performing different experiments, the algorithms try to search for the next best experiment to conduct, in order to reach a local optimal solution. To accelerate this process, the searching can use many heuristic search algo-

rithms, such as Simulated annealing, genetic algorithms, and Tabu search, to name a few. [24] is one of them, and developed a smart climbing-hill algorithm using the ideas of importance sampling and Latin Hypercube sampling, to reduce the amount of needed experiments.

iTuned [52] targets on finding an optimal setting of configurations for high performance. It tries to accelerate performance tuning by deciding which experiment to conduct. From an existing set of experiments, it constructs a response surface, and predicts the upcoming experiment with adaptive sampling, instead of randomly selecting the next experiment.

SARD [23] tries to rank the database configuration parameters based on their impact on performance with a statistical approach. It first needs to conduct a set of experiments, where each experiment has the configurations set to different values. With the result, it analyzes how each parameter impact the overal performance. In order to reduce the number of experiments needed, it only considers two extreme values for each parameter, and thus utilizes a P&B design methodology to estimate their effects. However, experiments in the above sections show that the optimal performance is usually not at the parameters' extreme values.

Some work assumes the specific knowledge of a particular system, and provides more specific guide in their tuning. STMM [17] focuses on automatically tuning the memory arrangement of the DB2 data base management system, to achieve an overal better performance. Since it is built into a database system itself, in each tuning cycle, it directly monitors the system memory distribution, and determines if the performance can be improved by database memory redistribution with a cost-benefit analysis of each memory pool used in the system. Since it is a database built-in feature, it assumes all the knowledge of the target system, and does not apply to others. With the same spirit, Oracle 11g introduced the SQL Performance Analyzer to help the admistrators analyze

the impact of the system parameter changes [37]. Starfish [16] concentrates on the performance tuning of Hadoop MapReduce. It assumes the knowledge of MapReduce architecture and performs the tuning according to each work phase.

Different from the above approaches of conducting experiments and selecting the optimal configuration settings, MassConf [60] utilizes the crowdsourcing method to find a better configration for the target system. It asks the vendor to collect the information from the existing users - including their configuration goals, system environment information, and their configuration selections. When a new user wants to deploy the system, based on the new user's environment and goal, MassConf gives a ranked list of the paramters to tune.

4.3 Observations

In order to help reduce the amount of time spent on the automatic performance tuning, we propose iOpt, to help reduce the amount of experiments the automatic tuning approaches need to perform in order to reach to an optimal settings.

In order to get the insight of the configurations that could impact the application performance, we studied the configurations of seven popular server applications, including MapReduce, STORM, PostgreSQL, MySQL, HBase, HDFS, and Hadoop core libraries.

Finding 5: While the number of all configuration entries are large, the performance related configuration entries are few.

Table 4.1 shows the number of performance related configuration entries for each application. All the configuration entries are examined manually. We did our best to find the semantic of each entry from both the application manual and the whole Internet, to decide whether they relate to the software performance.

Table 4.1. Number (percentage) of configuration parameters that are related to performance. "Total" is the number of all the configuration entries for each application. "Performance" is the number of configuration entries related to performance tuning.

Apps	Total	Performance Related
MapReduce	126	22 (17.5%)
STORM	80	11 (13.8%)
PostgreSQL	115	15 (13%)
MySQL	252	31 (12.3%)
HBase	157	10 (6.4%)
HDFS	74	8 (10.8%)
Hadoop-Core	73	3 (4.1%)

Compared to the total number of configuration settings each application provides, they only occupy around 10%. Among the studied applications, MapReduce has a higher rate of performance-related configurations. This is due to the nature that all the purpose of MapReduce is to improve the performance. There are many other configuration entries serving the purposes of connecting the software to the environment, such as the `datadir` entry shown in Figure 1.1, which is used to specify the directory to store the database files. That means, if we blindly choose all the configuration entries as candidates, the actual time used for the tuning could be at least 10 times longer than it should have.

Finding 6: Most performance related configuration entries control three behaviors of the application: computing resource usage, memory resource usage, and I/O usage.

By further observing the configurations in the target applications, we find that most of the parameters usually fall to 3 categories: memory usage, computing resource (CPU) usage, and I/O behavior. Table 4.2 shows the number of performance-related con-

figuration paramters in each category.The first column is the total number of performance-related parameters. The rest columns show the number of the parameters in each category. Note that the numbers in the three categories may not add up to the total number for some applications, because a few parameters may not fit into any of the three categories.

Memory resource usage usually means how much memory space can the functionality take. It can indicate the amount of memory to allocate to certain usage, for example, **io.sort.mb** in MapReduce indicates the percentage of memory to use for sorting files; it may also specify the caching capability, for example, **query_cache_size** in MySQL puts a boundary on how much memory can be used to cache the query, so that it doesn't need to be re-evaluated next time. Usually a higher value of the paramter of this kind means higher memory usage, and often result in higher performance.

Computing resource usage mostly means how many concurrent tasks the target functionality can be run at the same time. For example, **thread_concurrency** in MySQL specifies how many concurrent threads can be run in the system; **max_connections** in PostgreSQL specifies the maximum number of connections can be accepted at the same time. Since higher number of concurrent executing tasks ususally indicates higher throughput, if the resource limitation allows, these parameters are usually set higher to achieve better performance.

Table 4.2. The number of configuration parameters related to performance in each category.

Application Name	Performance Related	Memory Usage	Computing Resource Usage	I/O Related
MapReduce	22	13	5	4
STORM	11	6	5	0
PostgreSQL	15	10	3	2
MySQL	31	26	4	1
HBase	10	4	4	1
HDFS	8	1	2	4
Hadoop-Core	3	2	0	0

Another important performance category is the I/O behavior, this is because compared to computation, I/O usualy takes a lot of time to complete. Some configuration paramters allow the users to specify the behaviors in order to reach an optimal performance. For example, **fsync** in PostgreSQL shows whether to enforce the synchronized write operation for every write on disk; **dfs.datanode.readahead.bytes** in HDFS specifies the behavior of pre-mapping the unread disk data to system memory, thus increasing the disk read performance.

Many applications use the configurations from all the three categories to provide the users the opportunities to fine tune it according to their systems and workloads. A few applications may also use other configurations to control some adhoc performance-related functionalities, such as HDFS and Hadoop-Core as shown in Table 4.2. In this work, we only focus on the three major categories.

4.4 iOpt Design

The goal of iOpt is to help automatic performance tuning reduce the necessary number of experiments. It's approach contains two major parts: 1)**select the performance related configuration entries**, and 2) **find the constraints of the entries**. The first part filters out the parameters that are not to be considered in the experiments, and the second part filters the impossible values to be used in the experiments. As described in the above sections, they could significantly reduce the number of experiments needed for the tuning, and also make sure the generated experiments are valid, i.e. the configurations used in the experiments are eligible and meaningful.

4.4.1 iOpt Usage

iOpt helps any automatic performance tuning method save time. Figure 4.3 shows the position of iOpt in the process of software performance tuning. The gray

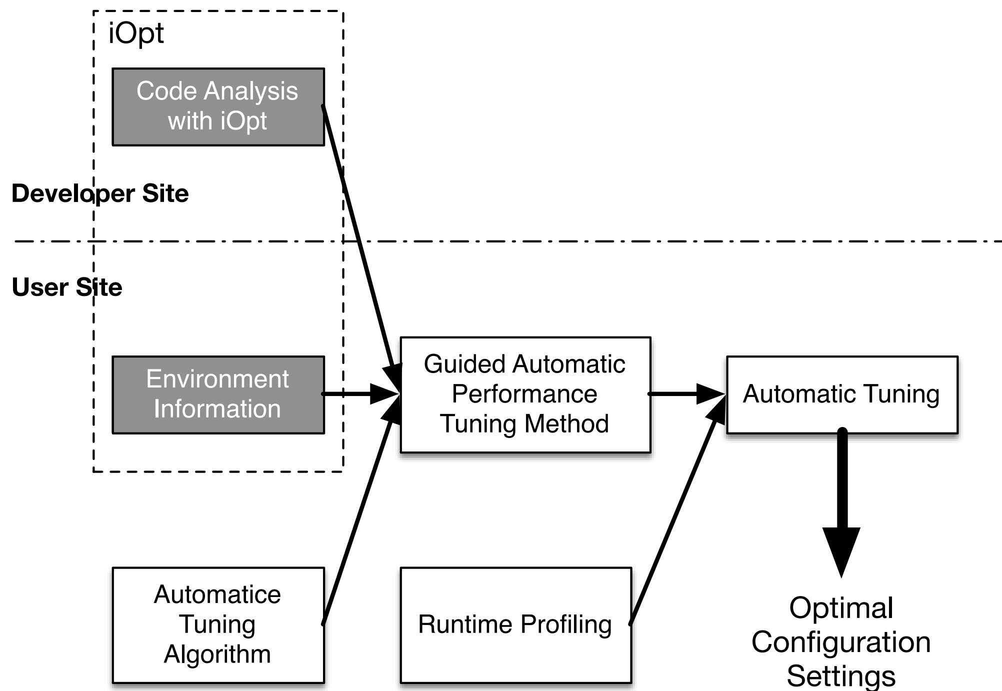


Figure 4.3. Using iOpt in performance tuning. The gray boxes are the elements of iOpt. The upper side of the figure needs to be done in the software developer side, while the lower part is in the software user site.

boxes show the two components of iOpt: the software static code analysis to find all the performance-related configurations as well as the constraints, and the environment information integration to utilize the analyzed data and merge with the running system.

With the iOpt output as well as any automatic tuning algorithm such as those found in [52, 17, 23, 42, 27], a guided automatic performance tuning method is formed, where the original algorithms would have a automatically generated limited set of tunable parameters (so as their value set). With the generated tuning method, the user uses it in the same way as the original one: applying it to the runtime profiling, and finally generate the optimal configuration settings.

Note that iOpt itself is not targeted to be a stand-alone automatic performance tuning tool. It doesn't include the profiling and testing framework, nor it provides the

evaluation or selection algorithm. Its goal is to reduce the input size of the existing performance tuning methods.

Although iOpt is designed to be working with other performance tuning tools, it's output could offer a significant help to the users who tune the software by themselves: it reveals the performance related parameters, their constraints and relations, as well as their expected values. A user can utilize these information to help them in the process of tuning.

4.4.2 Architecture Design

Figure 4.3 shows two components of iOpt: the environment information integration and the configuration entry analysis. The environment information part is largely the same as the one described in *EnCore*, and will not be duplicated here. In this section, we mainly focus on how to analyze the configuration entries to achieve the goal of configuration parameter selection and constraints analysis. For the C/C++ analysis, iOpt largely reuses the methods of SPEX, as described in [55].

In order to achieve the goal of suggesting the parameters, iOpt needs to understand the semantic meanings of the configuration entries to some extend. Unlike in *EnCore* where we study from the configurations of other systems, in iOpt we start the analysis from the source code. This is mainly because while correctness is usually assured by all the systems in the same way, but different systems may require different settings to achieve an optimal performance according to the whole system settings, therefore it's not reliable to consult the other systems' setting to optimally configure the target system.

Note however, the source code analysis does not mean the users should have the source code at hand. Instead, the analysis could be done on the software developer side: whenever there is a major release that has modifications to the configuration parameters, the analysis could be run again (automatically), so that it does not bring much burden

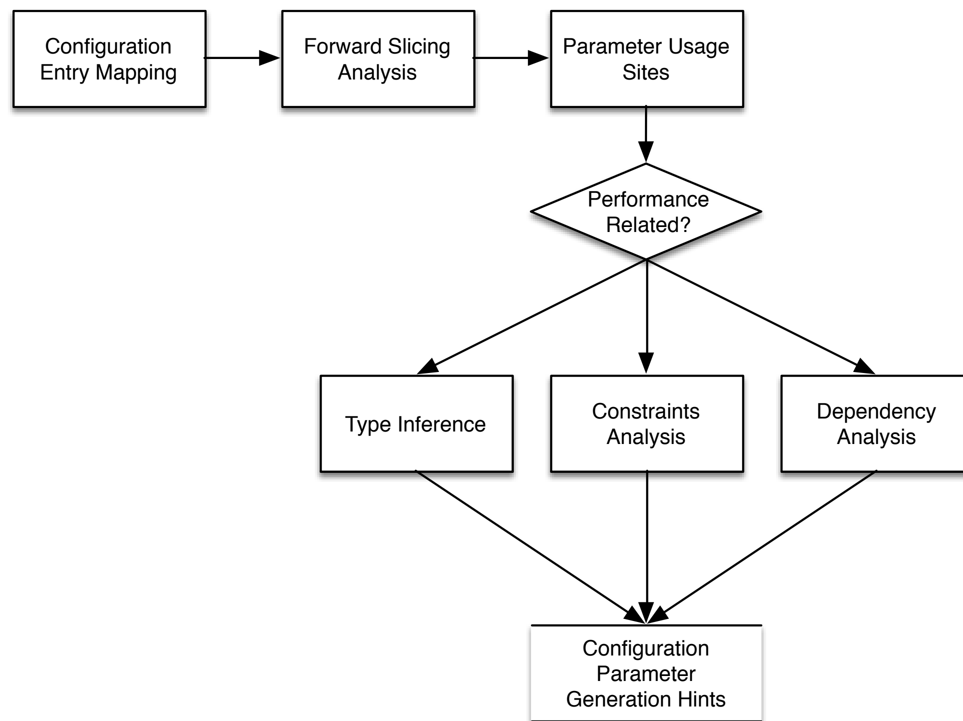


Figure 4.4. iOpt configuration parameter analysis architecture.

to the developers This is because although the final settings could be different between different target systems, the information of configuration entries are the same for one software.

Figure 4.4 describes the whole architecture of iOpt. iOpt firstly needs to find all the configuration entry reading sites in the source code, in order to start the analysis. It then performs a forward slicing analysis for each of the found configuration reading site. With the hints on how the read value is used, iOpt first decides whether the target configuration is related to performance. Further, it tries to analyze their types, their value constraints, and the dependency between different configuration entries. Finally, with all the information gathered in the previous steps, iOpt generates the configuration parameter hints to provide to the performance tuning methods.

Configuration Entry Mapping

To begin with the analysis, iOpt needs to first identify the starting points - the source code locations where the configuration entries are read. There are two choices for us to locate the configuration reading sites:

- annotating all the configuration entry names
- annotating the configuration reading classes/methods.

The first approach is the easiest one for the iOpt implementation. However, there are three problems with this approach: 1) there are too many configuration entries, as shown in Table 4.1, thus bringing a huge burden to the developers; 2) the configuration entries set needs to be validated with each software update in order to keep the correctness; and 3) some of the configuration names are seperated into several parts, and combined dynamically in the software - the analysis cannot find the appearance of the configuration entry even if it knows the configuration entry names.

Given the considerations, we take the second approach: to annotate the configuration reading methods or classes. The insight is that the developers usually develope a set of reusable configuration reading structures in order to use for the hundreds of configuration entries, and a well-structured piece of code could provide a clean interface to manage them more easily.

Figure 4.5 shows three configuration reading sites in both Hadoop (written in java) and PostgreSQL (written in C++). For the mapping functions in Java programs, especially Hadoop related projects, a standard way is to use a well-defined set of APIs, as described in [45]. As shown in Figure 4.5, they usually use the `getXXX()` functions such as `getInt(), getFloat(), getBoolean()` etc. As a fact, in total we only need to annotate 6 functions for the Hadoop applications.

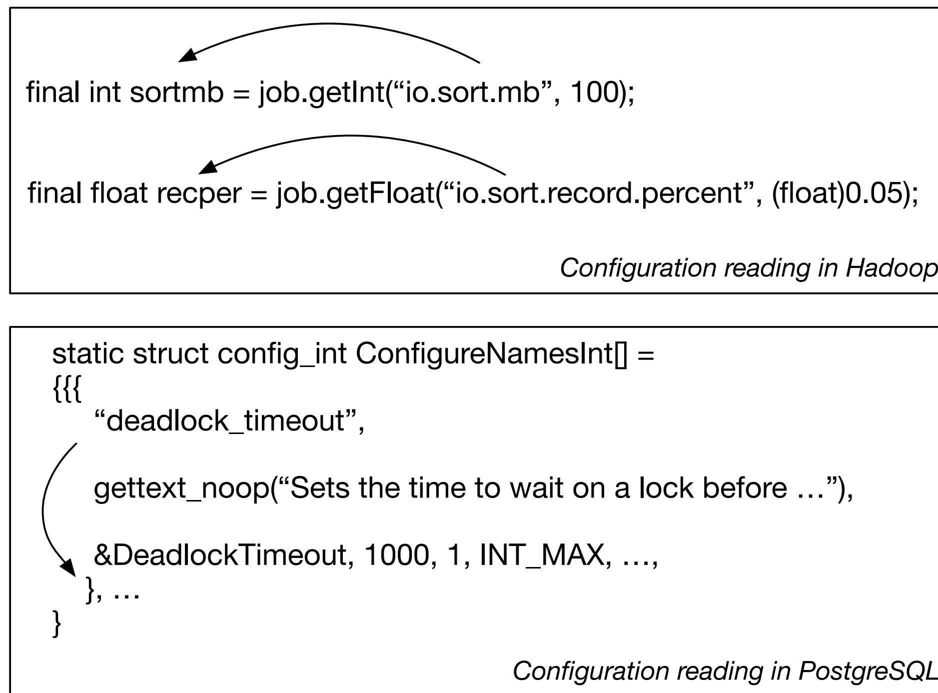


Figure 4.5. iOpt configuration parameter analysis architecture.

However, C/C++ programs may use other ways to conduct the task of configuration entry reading. We studied the mapping of 20 widely-used software and found that the majority of them manifests through 3 programming abstractions: *structure, branch,* and *container* [55]. Figure 4.5 shows an example of the *structure-based* mapping in PostgreSQL: the "deadlock_timeout" configuration is mapped to `DeadlockTimout` global variable. iOpt can find the reading site if the structure and the corresponding field are annotated. The other two types are further described in our paper [55], and is not duplicated here.

Recognizing Performance Related Parameters

After iOpt finds all the configuration reading sites of a given parameter, it checks whether the target parameter is related to the software performance. This checking utilizes our observation in Table 4.2, that the performance related configurations are

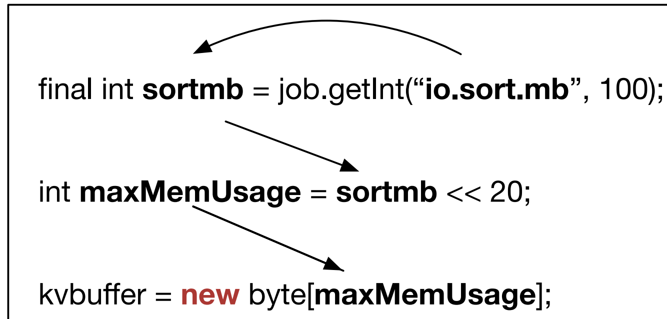
usually used for the purpose of three categories: computing resource, memory resource, or I/O behaviors.

In order to find the usage of the parameters, iOpt conducts an inter-procedure data flow analysis on each of them. Figure 4.6 shows three examples of how iOpt recognizes the parameters of the three performance related categories. Note that all the source code in the examples are greatly simpliefied, and they may not necessarily reside in the same function. However, it does not affect the inter-procedure analysis.

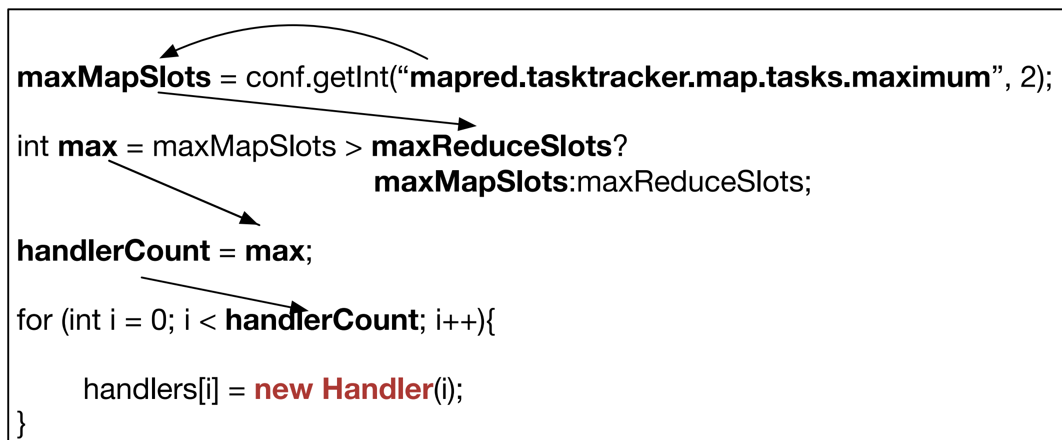
For each category, iOpt uses the corresponding hints to find the related parameters. For the memory usage related parameters, iOpt outputs a configuration entry when if it is used as an argument of a memory related system calls. The system calls in Java are usually the `new` keyword with the array identifier. Sometimes it could also appear in the jvm arguments to specify the size of the total memory the application could use. For C/C++, they are `malloc, calloc, alloc,` or `new`. Figure 4.6(a) shows how "io.sort.mb" is recognized as a memory related parameter.

The computing resource related parameters are recognized by the hints of threading. Figure 4.6(b) shows how "mapred.tasktracker.map.tasks.maximum" is recognized as a performance paramter: the number of the created threads has control dependency on the parameter. In Java the thread creation is done either with a class that extends `Thread` or a class that implements `Runnable`. In the example shown in the figure, the `Handler` can be threaded, and thus the related parameter is recognized. In C/C++, the thread creation is usually done with `pthread_create`, or `fork` if it relates to process creation. Note that for Java applications, we don't recognize process creations as it is done from the console when starting the jvm.

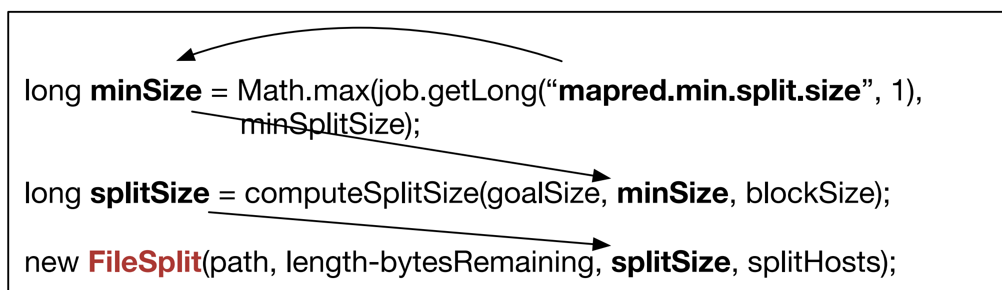
Similar to the previous two mentioned categories, the I/O related parameters are recognized via their respective system calls. The example shown in Figure 4.6(c) shows the recognization of "mapred.min.split.size" by tracing its usage to the class `FileSplit`,



(a) Recognizing memory related parameters



(b) Recognizing computing related parameters



(c) Recognizing I/O related parameters

Figure 4.6. iOpt configuration parameter recognizing.

which conducts I/O operations with the classes of `DataOutput` and `DataInput`. In C/C++, we recognize the system calls of `read/write/send/recv`.

Type Inference

it performs the type inference, in order to make the tuning phase aware of what kinds of values to set to a specific parameter. Note that iOpt only focuses on the performance related parameters, which makes the type inference simpler than the one described in [45]. This is because those parameters are usually only members and boolean values.

As mentioned in the above section, Hadoop related Java applications usually use a set of defined functions to read the parameter, and iOpt can infer the type of the parameter mostly from the function names, such as `getBoolean` (bool type), `getInt` (integer type), and `getFloat` (float type). For C/C++ programs, it can also be inferred from the target variables that store the read values.

Constrains Analysis

Many configuration parameters have their own constraints on tha values they could take. For example, a number describing the percentage needs to be a floating number between 0 to 1. Figure 4.7 shows an example of the value constraint in MapReduce. iOpt first identifies the reading sites of the parameter ”io.sort.record.percent”, and does a forward slicing. Along the analysis, it finds that there are value comparisons that has control dependency to an exception site. Therefore, we can conclude that the target parameter should fall into the range specified in the branch statement.

Note that some other constraints may not be included in the source code. Instead, they are implicitely limited by the executing environment. For example, the memory usage is limited by the system virtual memory, and the number of threads are usually

```
final float recper = job.getFloat("io.sort.record.percent", (float)0.05);

if (recper > 1.0 || recper < 0.01 ){

        throw new IOException();
}
```

Figure 4.7. iOpt value constraints recognization

bounded by the number of processing units in the system. iOpt considers this by recognizing the relevant parameters and correlate them to the environment information.

Chapter 5

Experimental Measurements

We test the effectiveness of *EnCore* with the rules learnt from public images crawled from Amazone EC2. In the evaluation we only use the predefined types and templates. The experiments are carried out with 3 software including Apache httpd server, MySQL, and PHP. For each software, we have different amount of training set according to the availability of the software on the EC2 public images: 127 images for Apache, 187 images for MySQL, and 123 images for PHP.

The rules *EnCore* infers from the training set include two parts: the type rules and correlation rules. The rule-based checking is in addition to the value comparison checking as in existing works, which also takes advantage of *EnCore*'s integration of environment information.

5.1 Misconfiguration Detection Effectiveness

To test *EnCore*'s effectiveness against misconfigurations, we perform three experiments. In the first experiment, random errors are injected to correctly configured systems, and *EnCore* is used to detect them. In the second experiment, we apply *EnCore* to check against known real-world misconfiguration problems. In the last experiment, we directly use the anomaly detector to the public images from EC2 and the virtual machine images in a commercial company's private cloud, to look for possible new configuration

Table 5.1. The number of injected misconfigurations detected by EnCore in the injection experiment.

App	Total	Baseline	Baseline+Env	EnCore
Apache	15	4	9	14
MySQL	15	5	14	15
PHP	15	9	12	15

errors.

5.1.1 Injected Misconfigurations

To inject misconfigurations, for each software, we randomly pick an image that is not in the training set, and inject 15 random errors to the configuration file with ConfErr [38]. ConfErr is a misconfiguration injection tool to test and quantify the resilience of software systems to human-induced configuration errors. It uses human error models rooted in psychology and linguistics to generate realistic configuration mistakes; it then injects these mistakes and measures their effects, producing a resilience profile of the system under test.

After the injection, we use *EnCore* to check against the tinted systems. The results are compared with two other approaches: 1)the Non-Correlation and Environment-Unaware (NCEU) one, which is used by existing works and does not consider environment information nor correlation, and 2) Non-Correlation but Environment-Aware (NCEA), which is enhanced by the type-based environment information integration provided by *EnCore*, so that not only configuration values, but the environment information is also taken into consideration. As shown in Table 5.1, there is a significantly larger coverage of *EnCore*. In the table, “Baseline” is the non-correlation and environment-unawared approach, adopted in most existing work. “Baseline+Env” is non-correlation

but environment-aware, i.e., only using the type-based environment infomation integrated by EnCore. It further shows that with the environment information integration and type detection, the existing value comparison approaches could also gain huge benefits, and discover more problems.

Among the 26 injected errors that are detected by *EnCore* but not NCEU, we find 9 of them are due to the lack of correlation detections, and the rest 17 of them are because of the lack of environment information(20) or the type detection(1). There is one error not detected in Apache, where the value is a pure string that specifies a format, which various in the training set and at the same time not having any hint from other sources. In this experiment, we assume all the approaches are able to detect unseen directive entries, as it is a trivial extension to any configuration error detection work.

However, note that the misconfiguration injection itself is environment-unaware, and only injects in the configuration files but not the other related places in the system. For example, it does not try to change the permission or the ownership of a file specified in the configuration. This limitation of the injection tools also limits the significance of the benefits brought by *EnCore* in the experiment.

Table 5.2. Detection of real world misconfigurations.

ID	Software	Problem Description	Info	Rank
1	Apache	Website not granted desired protection because DocumentRoot does not have related Directory	Corr	1(5)
2	PHP	Does not connect to database due to extension_dir pointing to a file instead of the directory	Env	1(1)
3	MySQL	File creation error due to datadir's wrong owner	Env + Corr	1(1)
4	MySQL	Data writing error due to undesired protection from AppArmor	Env	1(2)
5	PHP	Modules not loaded because extension_dir is set to a wrong location	Env	1(1)
6	Apache	Website unavailability because directory contains symbolic links when FollowSymLinks is off	Env + Corr	1(3)
7	Apache	Website visitors are unable to upload files due to the wrong permission set to Apache user.	Env + Corr	1(1)
8	MySQL	Out of memory error due to too large table size allowed in configuration	Env + Corr	-
9	MySQL	Logging is not performed even with relevant entry set correctly due to wrong permission	Env + Corr	1(1)
10	PHP	Failure when uploading large file due to the wrong setting of file size limit	Corr	2(2)

5.1.2 Real Misconfigurations Problems

To test how the system reacts to real world misconfigurations, we used the problem set described in [62]. he problems are manually reproduced in a new testing image. Table 5.2 shows the problem description as well as the detection results. The “Info” column describes the information needed to detect the misconfigurations: “Corr” refers to correlation, and “Env” refers to environment infomation. The “Rank” column shows the rank of the real misconfiguration in the warning report, with the total number of warnings shown in “()”. “-” means the misconfiguration is missed by *EnCore*. Note that the majority of them require both environment and correlation information. Due to space limit, we skipped the problems with similar root causes, and only show 8 problems of different kinds. In addition, we show two real cases caused by the configuration errors we found in EC2 image as described in Section 5.1.3.

EnCore does not detect problem #8 in Table 5.2. The root cause of the problem is that the *max_heap_table_size* is set to be equal to the system memory size, but the system cannot allocate all the space to MySQL. In fact, detecting this problem is a typical usage of environment information and correlation. *EnCore* misses it only due to the lack of hardware information in the training set. The hardware data is avoided intentionally when we are crawling images. Because they can be chosen arbitrarily when instantiating the images on EC2, and could not reflect the real usage scenario when the images are created and deployed. It is trivial to extend the data with this information in real usage.

When checking Apache httpd server, some other warnings are generated mostly from configuration entry name checking. This is because Apache allows embedded direcives in arbitrary levels of sections, and if the combination of the sections and directives are not seen in the training set, a warning is reported. Problem #10 has the root cause of the problem ranked No.2 due to another misconfiguration in the file, which violates a

Table 5.3. Categories of new detected misconfigurations. "FilePath" means the path setting is missing or set wrongly. "Permission" means the permission setting is wrong. "ValueCompare" shows the misoncifugraions violating value comparison rules.

Source	FilePath	Permission	ValueCompare	Total
EC2	3	10	24	37
PrivateCloud	10	3	11	24

rule with a higher confidence.

5.1.3 Detecting New Misconfigurations

In addition to the images in the training set, we collected 120 images from EC2 for testing, and use *EnCore* to directly check them against the generated rules. It is to our surprise to find a total of 37 configuration errors in 25 images. This was not expected because the public images are mostly used as templates to produce other images, and considered to be correct. We also check 300 images in the commercial private cloud of an IT company, and find 24 problems in 22 images. Table5.3 shows their categories. All of them are manually verified to be vulnerable configurations that either affect the security of the systems, or cause unexpected behaviors.

Three of them has file path specified in the configuration files while the corresponding file should but does not exist. For example, one image has MySQL's configuration file to include another file which does not exist, causing MySQL unable to start. Another example is in Apache, the aliased directory does not exist, causing the website visiter unable to view certain contents.

We find that many of these misconfigurations cannot be detected by the software itself, and the related rules are not enforced in the source code. But they can cause undesired behavior or security problems after deployed. These configuration errors have caused real problems that are already encountered by the other software users [10, 8].

For example, the log file in MySQL is not supposed to be accessible by other users because it may contain sensitive data [8]. However, even with the detrimental excessive privilege, MySQL executes normally with the security issue that is difficult to be discovered. In total, we find 4 images having this security problem.

Another example is in PHP, the size of the uploaded file is limited by two entries: *post_max_size* and *upload_max_filesize*, and the first one has higher priority. Thus for the later one to take effect, it needs to be smaller than the first one, otherwise the upload of a file larger than *post_max_size* fails even if the file size is smaller than *upload_max_filesize*. It can be noticed that the ratio of problematic images in the commercial private cloud is lower than that of EC2. This is expected because they have been deployed in real usage for quite some time, and should have most problems discovered already.

It is interesting to notice that none of these misonfigurations can be detected without the use of environment information and correlation detection. For example, the permission violation is detected by the rule found from the 8th template in Table 3.3, where file path permission is correlated with user names. In this particular case, it checks the accessiblity of the Linux system user 'nobody'. The experiments illustrate the effectiveness and neccessarity to integrate these two important factors when dealing with misconfiguration issues.

5.2 Type Inference Accuracy

Table 5.4 shows the type inference results as described in Section 3.2.2.The "Entries" column is the total number of configuration entries. The "NonTrivial" column refers to the types with semantic meanings that are not regarded as "string" or "number." "FalseTypes" / "Undetected" show the number of entries with wrongly detected or undetected types. For the entries that do not match any hint or do not pass verification, we assign them with the type String or Number(trivial), which may include the configura-

Table 5.4. Data type detection results.

Apps	Entries	NonTrivial	FalseTypes	Undetected
Apache	371	207	14	20
MySQL	131	86	3	11
PHP	249	164	13	8

tions that specify string format, a regular expression, or a count number. For the rest of them, we infer them according to Table 3.1.

To verify the results, we manually check the semantics of all the entries and compare them with the inference output. We have both false positives and false negatives. A major reason of the false inference is the use of regular expression and wildcards in the configuration files to specify the file names or paths. Another reason is that the specified file names or paths may not necessarily exist in the system, and they are used just as a optional hints to the software. For example, in Apache users could specify the index file to serve a directory. But the file may actually not exist, and the software handles its absence automatically. Another example is the log files of MySQL that will be created when the software is running. For PHP, the wrong detection mostly come from the integer values mistakenly determined as Boolean, when all the training images are using the values of 0 or 1. Note however, these happen because we are using the data set of template images from Amazon EC2, which means they are usually the clean systems to start with. The results will be improved with the snapshots of working systems as they already have all the needed files in place, and more customized configuration values.

It is obvious that the configuration entries in our study do not cover all the possible configuration options in the software, because they need to be present in our training set. However, less obviously, we may have additional configuration entries recorded in addition to those specified in the manual. This is because certain configuration entries

have multiple segments. For example, the `LoadModule` directive in Apache constitutes two parts: the module name, as well as the module file. The parser separates them to form two individual entries. It makes sense to our *EnCore* as the two parts have different meanings and thus different types.

Also note that the number of configuration entries here is different from the number of columns specified in Table 2.2, The reason is that Table 2.2 represents the number of the tree nodes in the data set, and different instances of the same configuration entries are treated as different columns because we need to provide the original data to the rule generator. However, it makes sense to aggregate the types inferred from different instances of the same configuration entry together.

5.3 Correlation Rule Inference

Table 5.5 shows the number of rules generated by applying the predefined templates (described in Table 3.3) to each software. For each software, we measure the number of rules found for the template, as well as the number of false positives. We use the confidence of 90%, support number of 10% of the total number of images in the training set, and entropy of 0.325 (described in Section 3.3.2). The threshold is selected according to the nature of our training set. EC2 images are often used as general 'template' images for the users to customize them to their own needs. Therefore many of the images' configuration entries are the default values. In this case, the entropy is usually small. For the same reason, the possibility of the occurences of certain configuration entries, especially those that are not in the sample configuration files provided by the software, is also small. Therefore we use a small support number for the purpose of filtering. A significant source of false rules is the undetected types, which causes the comparison of unrelated entries. For example, in Apache, both *MinSpareServers* and *Timeout* are inferred as Numeric type. Thus they are compared although being not

Table 5.5. Detected correlation rules with the filters.

Apps	Detected Rules	False Positives
Apache	42	9
MySQL	29	4
PHP	31	10

correlated. Since *MinSpareServers* are mostly smaller than *Timeout* numerically, it is reported as a rule.

The detected correlations in Table 5.5 is the results after applying the filters. As described in Section 3.3.2, besides the standard filters of confidence and support number, we also use entropy. To evaluate the effectiveness of entropy filter in our data set, Table 5.6 shows the number of false rules filtered in each software. "Original" shows the number of rules after applying the confidence and support filters. FP (False Positive) Reduced is the number of false rules filtered by entropy filter. FN (False Negative) Introduced is the number of true rules that are filtered. The number of original detected rules are the results after only applying the confidence and support filters.

In our experiments, entropy is mostly effective against the false positives in the correlations related to numeric rules, as well as binomial association rules. The reason is that some of these values are directly derived from the sample configuration files and not changed by the image developers. For example, the *HostnameLookups* directive in Apache is always set to *Off*; and the *min_server_severity* directive in PHP is always set to *10*. The high number of originally reported rules in PHP are mostly contributed by the comparison of numeric entries. Since many of these settings are not changed much, the filter could effectively rule them out.

However, for the same reason, entropy also inevitable filters some true correlations. For example, while *net_buffer_length* should be smaller than *max_allowed_packet*

Table 5.6. Effectiveness of the entropy filter.

Apps	Original	FP Reduced	FN Introduced
Apache	113	71	7
MySQL	52	23	1
PHP	567	536	1

in MySQL, since all the values of *net_buffer_length* is *8K*, this rule is filterd by mistake. Although we think exposing more correlations is more critical than suppressing the false reports when detecting misconfigurations, given the false positive and false negative number introduced by the filter, we consider the trade-off worthwhile and beneficial to the end users.

For the two parts that are involved in the comparison of numeric values, it is possible that one part is usually within a range that is much smaller than the other part. Thus although they may not have direct logical relation, our tool still reports them. Currently we allow these reports because even if the software logically allows the comparison order to be reversed, this anomaly is likely to have problem as it violates the rule found from most other peers. For example, although they have no semantic relation, we report the correlation of *thread_stack* < *max_binlog_size* in MySQL, as the former one is usually within hundreds of KB while the later is usually within hundreds of MB. We consider it beneficial as if the user mistakenly set the *thread_stack* value too large that is even larger than *max_binlog_size*, it might be harmful. Indeed the MySQL manual states that the default value of 192KB is large enough for normal operation. However, if this behavior is not desired by the users, a more complicated filtering rule based on clustering algorithms such as *k-means* [32] can be used. With the help of clustering, the numeric configuration entries can be first divided into several groups based on their values, and correlation rules are filtered if the involved entries fall into different groups.

Table 5.7. Performance related parameters found with iOpt.

application	Memory Related		Computing Related		I/O related	
	Original	Found	Original	Found	Original	Found
MapReduce	13	11	5	3	4	3
PostgreSQL	10	8	3	3	2	2

We leave the choice of additional filters to our future work.

5.4 Finding Performance Parameters with iOpt

The Java detector in iOpt is implemented with IBM Wala Java program analysis framework [14], and the C/C++ version SPEX is implemented with LLVM. To test the effectiveness of iOpt in finding performance related parameters, we conduct an analysis of both MapReduce and PostgreSQL. Table: 5.7 shows the the results of performance related parameters detection. In total, 77% of performance related parameters in MapReduce and 86.7% in PostgreSQL are found.

One major reason for the false negative is that for some of the memory related parameters, they do not take control at the places where memory is created (with `alloc` etc.). For example, `maintainece_mem` in PostgreSQL is referred only when using the memory, and controls the upper limit of the memory usage in a given pool. Therefore there is not a general hint on whether it relates to the memory usage.

At the same time, iOpt analysis can suffer from false positives - meaning the parameter is reported as performance related but actually not. Due to the scalability issue with Wala, we tested 30 parameters in MapReduce, and found 7 false positives. In PostgreSQL we also found 3 false positives. Interestingly, they all come from memory related paramters. The major reason is that they participated in deciding how much memory should be allocated at certain memory allocation sites. However, these places

Table 5.8. The constraints and dependencies inference and their accuracy.

application	Data Range		Contrl Dependency		Value Relation	
	Found	Accuracy	Found	Accuracy	Found	Accuracy
Apache	42	94.6%	1	100%	9	81.8%
MySQL	213	99.1%	35	94.7%	10	71.4%
PostgreSQL	186	97.3%	44	91.7%	6	85.7%
OpenLDAP	20	73.1%	0	N/A	2	50%
VSFTP	84	100.0%	68	63.9%	1	100%
Squid	120	100%	14	77.8%	9	100%

actually do not consume the major part of the memory, and is used only to keep meta information instead of the large amount of memory used to store data to affect the performance. For example, while `max_connections` in PostgreSQL is recognized as computing resource related, it is also recognized as memory related, due to the fact that its value also participates in deciding how many semaphores to allocate with the memory allocation method.

To test the constraints analysis, we run Spex with 6 C/C++ applications with all the configuration parameters. It is obvious that there are significant amount of constraints specified in the software source code. The accuracy is valideted by manually examining all the inferred constraints. It shows an overall accuracy of over 90%. One of the major cause of inaccuracy is caused by the aliased pointers in configuration parameter parsing.

5.4.1 iOpt Case Study

While iOpt is targeted at providing a filtered input to the automatic performance tuning tools, it could help other users understand more about how the configuration parameters are used in the system, and thus providing an insight help on tuning the

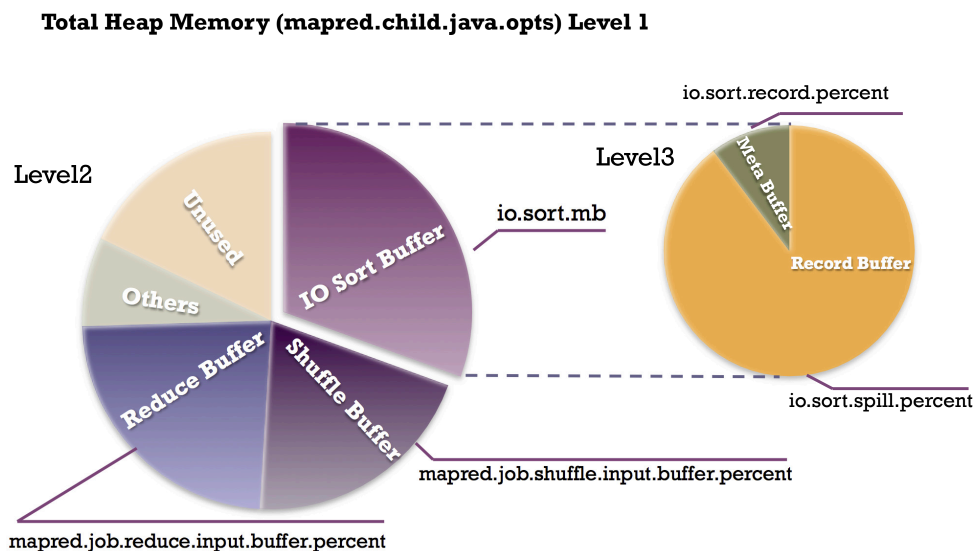


Figure 5.1. MapReduce memory usage inferred from iOpt results.

parameters more efficiently.

Figure 5.1 shows how the memory pools are arranged in MapReduce. The whole memory is mainly split by the IO sort buffer, the reduce buffer, and the shuffle buffer, and the IO sort buffer is further split by a meta buffer and record buffer. iOpt first locates that the parameter of `mapred.child.java.opts` controls the whole jvm memory usage with the parameter of `Xmx`, and thus we know it is the first-level memory - any memory allocated is part of it. Next, iOpt recognizes the three parameters `io.sort.mb`, mapred.job.reduce.input.buffer.percent, and mapred.job.shuffle.input.buffer.percent are memory related. What's more, from their allocation sites, we find they are all allocating memory from the whole memory pool. Therefore iOpt inform that these parameters are splitting the whole memory in the same second level. Thus the user can consider these parameters together - increasing one parameter while decreasing another.

Further, when analyzing the paramter of `io.sort.record.percent`, iOpt finds that this parameter is a percentage, and multiplies the memory denoted by `io.sort.mb`, thus it occupies a part of memory from IO sort buffer. In the same way, it also reports

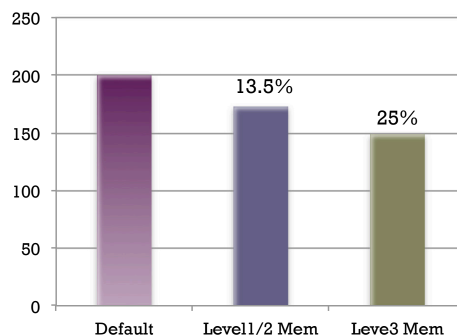


Figure 5.2. Performance tuning results with iOpt memory hierarchy reports.

`io.sort.spill percent` also occupies IO sort buffer. Therefore, these two parameters are considered together at the 3rd level. Figure 5.2 shows the performance improvement with the parameter adjustment on each memory level.

Chapter 5, in part, is a reprint of the material as it appears in 19th International Conference on Architectural Support for Programming Languages and Operating Systems. Jiaqi Zhang, Lakshminarayanan Renganarayana, Xiaolan Zhang, Niyu Ge, Vasanth Bala, Tianyin Xu, and Yuanyuan Zhou. The dissertation/thesis author was the primary investigator and author of this paper.

Bibliography

[1] Android lint. http://developer.android.com/tools/help/lint.html.

[2] Apache httpd server. http://httpd.apache.org.

[3] Google styleguide. https://code.google.com/p/google-styleguide/.

[4] Hadoop mapreduce. http://hadoop.apache.org/.

[5] Internet assigned numbers authority. http://www.iana.org.

[6] Misconfiguration brings down entire .se domain in sweden. http://www.circleid.com.

[7] Mysql database. http://www.mysql.com.

[8] Mysql log security. http://www.securityfocus.com /advisories/3803.

[9] Overview of starfish. http://www.cs.duke.edu/starfish/index.html.

[10] Php configuration error. http://stackoverflow.com/ questions/7754133.

[11] Rapidminer. http://www.rapid-i.com.

[12] Special report: Site performance equals profit. http://www.internetretailer.com/2013/11/04/sponsored-supplement-site-performance-equals-profit.

[13] The support authority: Get to know the visual configuration explorer. IBM Developer Networks.

[14] Wala: T.j. watson libraries for analysis. http://wala.sourceforge.net/wiki/index.php/ Main-Page.

[15] weka. http://www.cs.waikato.ac.nz/ml/weka.

[16] Starfish: A self-tuning system for big data analytics. In *5th Conference on Innovative Data Systems Research (CIDR '11)*, 2011.

[17] Sam Lightstone Yinxin Diao M Surendra Adam Storm, Christian Garcia-Arellano. Adaptive self-tuning memory in db2. In *International Conference on Very Large Data Bases*, 2006.

[18] Sarabjot Anand, David Bell, and John Hughes. The role of domain knowledge in data mining. In *1995 International Conference on Information and Knowledge Management*, 1995.

[19] Mona Attariyan, Michael Chow, and Jason Flinn. X-ray: Diagnosing performance misconfigurations in production software. In *OSDI*, 2012.

[20] Mona Attariyan and Jason Flinn. Using causality to diagnose configuration bugs. In *USENIX Annual Technical Conference*.

[21] Mona Attariyan and Jason Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *OSDI*, 2010.

[22] L. A. Barroso and U. Holzle. The data center as a computer: An introduction to the design of warehouse-scale machines.

[23] Mohamed Mokbel Biplob Dednath, David Lilja. Sard: A statistical approach for ranking database tuning parameters. In *Data Engineering Workshop on ICDEW*, 2008.

[24] Mukund Raghavachari Cathy H Xia Bowei Xi, Zhen Liu and Li Zhang. A smart hill-climing algorithm for application server configration. In *2004 International World Wide Web Conference*, 2004.

[25] P.S. Bradley and O.L. Mangasarian. Feature selection via concave minimiation and support vector machiens. In *5th International Conference on Machine Learning*, 1998.

[26] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *8th USENIX conference on Operating Systems Design and Implementation*, 2008.

[27] Jeffrey Hollingsworth Cristian Tapus, I-Hsin Chung. Active harmony: Towards automated performance tuning. In *2002 ACM/IEEE Conference on Super Computing*, 2002.

[28] Glenn Ammons Todd Mummert Bowen Alpern Vasanth Bala Darrell Reimer, Arun Thomas. Opening black boxes: using semantic information to combat virtual machine image sprawl. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, 2008.

[29] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *18th Symposium on Operating Systems Principles*, 2001.

[30] J. Gray. Dependability in the internet era., 2001. Keynote presentation at the 2nd HDCC Workshop.

[31] Jiawei Han, Jian Pei, and Yinwen Yin. Mining frequent pattern without candidate generation. In *SigMOD*, 2000.

[32] J.A. Hartigan and M.A. Wong. Algorithm as 136: A k-means clustering algorithm. *Journal of the Royal Statistical Society*, 1979.

[33] Mei Hong, Zhang Lu, and Yang Fuqing. A component-based software configuration management model and its supporting system. In *ICSE*, 2002.

[34] Roberto J. and Bayardo Jr. Efficiently mining long patterns from database. In *1998 ACM SigMOD International Conference on Management of Data*, 1998.

[35] Jayasinghe.D Malkowski.S Pengcheng Xiong Pu.C Kanemasa.Y Jack Li, Qingyang Wang and Kawaba.M. Prot-based experimental analysis of iaas cloud performance: Impact of software resource allocation. In *IEEE 9th International Conference on Services Computing (SCC'12)*, 2012.

[36] R. Johnson. More detail's on today's outage. http://www.facebook.com/note.php?noteid=431441338919.

[37] P.Belknap K. Yagoub. Oracle's sql performance analyzer. In *IEEE Data Engineering Bulletin*, 2008.

[38] Lorenzo Keller, Prasang Upadhyaya, and George Candea. Conferr: A tool for assessing resilience to human configuration errors. In *International Conference on Dependable Systems and Networks (DSN)*, 2008.

[39] David Lutterkort. Augeas - a configuration api. In *2008 Linux Symposium*, 2008.

[40] Kiran Nagaraja, Fabio Oliveria, Ricardo Bianchini, Richard P. Martin, and Thu D. Nguyen. Understanding and dealing with operator mistakes in internet services. In *6th USENIX conference on Operating Systems Design and Implementation*, 2004.

[41] D. Oppenheimer, A. Ganapathi, and D. Patterson. Why do internet services fail, and what can be done about it. In *USENIX Symposium on Internet Technologies and Systems*, 2003.

[42] Dejan Perkovic Peter Keleher, Jeffrey Hollingsworth. Exposing application alternatives. In *International Conference on Distributed Computer System*, 1999.

[43] Ariel Rabkin and Randy Katz. Precomputing possible configuration error diagnoses. In *26th IEEE/ACM International Conference on Automated Software Engineering*, 2011.

[44] Ariel Rabkin and Randy Katz. Static extraction of program configuration options. In *ICSE*, 2011.

[45] Ariel Rabkin and Randy Katz. Static extraction of program configuration options. In *International Conference on Software Engineering*, 2011.

[46] Agrawal Rakesh and Srikant Ramakrishnan. In *20th International Conference on Very Large Data Bases*, 1994.

[47] Vinod Ramachandran, Manish Gupta, Manish Sethi, and Soudip Roy Chowdhury. Determining configuration parameter dependencies via analysis of configuration data from multi-tiered enterprise applications. In *International Conference on Autonomy Computing*, 2009.

[48] Brin S, Motwani R, Ullman J, and Tsur S. Dynamic itemset counting and implication rules for market basket data. In *1997 ACM SigMOD Conference on Management of Data*, 1997.

[49] Amazon Web Service. Summary of the amazon ec2 and amazon rds service disruption in the us east region. http://aws.amazon.com/message/65648.

[50] Claude Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 1984.

[51] Koen Smets and Jilles Vreeken. Slim: Directly mining descriptive patterns. In *SIAM International Conference on Data Mining*, 2012.

[52] Shivnath Badu Songyun Duan, Vamsidhar Thummala. Tuning database configuration parameters with ituned. In *International Conference on Very Large Data Bases*, 2008.

[53] Ramakrishnan Srikant and Rakesh Agrawl. Mining quantative association rules in large relational tables. In *1996 ACM SigMOD Conference on Management of Data*, 1996.

[54] Y-Y Su, M Attariyan, and J Flinn. Autobash: Improving configuration management with operating system causality analysis. In *21st ACM Symposium on Operating Sytems Principles*, 2007.

[55] Peng Huang Jing Zheng Tianwei Sheng Ding Yuan Yuanyuan Zhou Tianyin Xu, Jiaqi Zhang and Shankar Pasupathy. Do not blame users for misconfigurations. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*, 2013.

[56] S Traugott and J Huddleston. Bootstrapping an infrastructure. In *Large Installation System Administration Conference*, 1999.

[57] Helen J. Wang, Yih-Chun Hu, Hun Yuan, Zheng Zhang, and Yi-Min Wang. Friends troubleshooting network: Towards privacy-preserving, automatic troubleshooting. In *IPTPS*, 2004.

[58] Helen J. Wang, John C. Platt, Yu Chen, Ruyun Zhang, and Yi-Min Wang. Automatic misconfiguration troubleshooting with peerpressure. In *OSDI*, 2004.

[59] Yi-Min Wang, Chad Verbowski, John Dunagan, Yu Chen, Helen Wang, Chun Yuan, and Zheng Zhang. Strider: A black-box, state-based approach to change and configuration management and support. In *17th Large Installation Systems Admistration Conference*, 2003.

[60] Ricardo Bianchini Wei Zheng and Thu D.Nguyen. Massconf: Automatic configuration tuning by leveraging user community information. In *2011 International Conference on Performance Engineering*, 2011.

[61] Matt Welsh. What i wish systems researchers would work on. http://http://mattwelsh.blogspot.com/2013/05.

[62] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N. Bairavasundaram, and Shankar Pasupathy. An empirical study on configuration errors in commercial and open source systems. In *SOSP*, 2012.

[63] Ding Yuan, Yinglian Xie, Rina Panigrahy, Junfeng Yang, Chad Verbowsky, and Arunvijay Kumar. Context-based online configuration-error detection. In *2011 USENIX Anuual Technical Conference*, 2011.