STATICALLY INFERRING PERFORMANCE PROPERTIES OF SOFTWARE CONFIGURATIONS

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCE
IN CANDIDACY FOR THE DEGREE OF
MASTER

DEPARTMENT OF COMPUTER SCIENCE

BY
CHI LI

CHICAGO, ILLINOIS
NOVEMBER 17 2019
Copyright © 2019 by Chi Li
All Rights Reserved
# TABLE OF CONTENTS

LIST OF FIGURES ................................................................. v
LIST OF TABLES ............................................................... vi
ACKNOWLEDGMENTS ............................................................ vii
ABSTRACT ........................................................................ viii

1 INTRODUCTION ................................................................. 1
  1.1 Motivation ................................................................. 1
  1.2 Contribution ............................................................ 3

2 PERFORMANCE IMPACT TAXONOMY ................................. 6
  2.1 Data Dependency ....................................................... 6
  2.2 IF-related Control Dependency .................................... 8
    2.2.1 Compared with a constant .................................... 8
    2.2.2 Compared with a getting-closer variable .................. 8
    2.2.3 Compared with a back-and-forth variable ................ 9
    2.2.4 Compared with an unrelated variable ..................... 9
  2.3 LOOP-related Control Dependency ................................ 10
    2.3.1 Affecting loop bound ........................................ 10
    2.3.2 Affecting loop stride ......................................... 10
    2.3.3 Affecting synchronization-loop ............................. 10
    2.3.4 Affecting infinite-loop ...................................... 11

3 PERFCONF AND PATTERN IDENTIFICATION ......................... 12
  3.1 Identifying PerfConfs and PerfOps ............................... 12
    3.1.1 What are configuration (derived) variables? ........... 12
    3.1.2 What are performance operations? ........................ 13
    3.1.3 What are performance-related configurations? .......... 14
  3.2 Identifying PerfConf patterns .................................... 15

4 ANALYSIS BEYOND PATTERNS ........................................... 17
  4.1 Input Analysis ........................................................ 17
  4.2 Slope Analysis ........................................................ 18
  4.3 Configuration Setting Range Analysis .......................... 19
  4.4 Configuration Relation Analysis ................................ 19
  4.5 Monotonicity Analysis .............................................. 20

5 EVALUATION ................................................................. 22
  5.1 User-facing PerfConf Identification ............................. 22
  5.2 Performance-pattern Identification ............................. 24
  5.3 Performance Property Inference .................................. 29
  5.4 Analysis Time ......................................................... 33
# LIST OF FIGURES

1.1 Configuration file hdfs-default.xml ................................. 2
1.2 Performance related code using the configuration .................. 2
1.3 *LearnConf* Overview .............................................. 4

4.1 *iosortmb* has constant slope .................................. 19
4.2 *timeout* has non-constant slope ................................ 19
4.3 *upperLimit* enables *lowerLimit* ................................. 20
4.4 *numRetries* and *sleepTime* work simultaneously ............ 20

5.1 Data Dependency: sortMB .......................................... 25
5.2 If - w/ Constant: maxFsObj ...................................... 26
5.3 If - w/ getting closer: buffer size ............................... 26
5.4 If - w/ back&forth: bufferSize .................................. 26
5.5 If - w/ unrelated: commitlogSize ................................ 26
5.6 Regular loop bound: lowerLimit .................................. 27
5.7 Regular loop stride: splitSize ................................... 27
5.8 Sync loop: blockingSize ......................................... 27
5.9 Infinite loop: msgInterval ...................................... 27
5.10 Linear relation ................................................... 29
5.11 Range effect .................................................... 29
5.12 Related configurations ........................................... 30
5.13 Non-monotonic .................................................... 31
5.14 *LearnConf* enhance ............................................. 31
LIST OF TABLES

2.1 Configurations’ Performance-Impact Taxonomy. To ease the discussion, all the formulas, figures, and explanations in the table focus on the Toy Examples, where C is the value of a configuration and Perf refers to the memory-consumption contribution of one instance of the toy-example code snippet. The Perf—Conf figure depicts how the Perf (the y-axis) might change with the configuration setting (the x-axis); the Perf—Time figure depicts how the Perf (the y-axis) might change over time (the x-axis) under different configuration settings, C1 (blue dots) and C2 (red crosses). ................................................................. 7

5.1 User-facing PerfConfs reported by LearnConf and those used by other sources. The k-th sources for different applications are different and hence are not summed together; F+: false positives; F-: false negatives. .......................................................... 22

5.2 PerfConf-PerfOp pattern distribution (Pk indicates the k-th pattern listed in Table 2.1). ................................................................. 24

5.3 PerfConf-PerfOp pattern identification accuracy. .......................................................... 25

5.4 All PerfConf performance properties. UF: User-facing; LR: linear relation; RE: range effect; RP: having related PerfConfs; MO: monotonic. .......................................................... 29

5.5 All user-facing PerfConfs in HBase. S1-3: Source 1-3; LR: linear relation; RE: range effect; RP: having related PerfConfs; MO: monotonic; #: false positive of LearnConf. .......................................................... 30

5.6 Analysis Time ................................................................. 34
ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my advisor, Prof. Shan Lu. Without her support and guidance, this thesis would not have been possible. I would also like to thank Prof. Henry (Hank) Hoffmann for his great support and constructive advice. Special thanks to Prof. Ravi Chugh for generously sparing their time to serve on my defense committee. I would also like to thank Shu Wang for his help on the project and all my PhD fellows and friends for their help in every aspects of my life.
ABSTRACT

Modern software systems often have a huge number of configurations whose performance properties are poorly documented. Unfortunately, obtaining a good understanding of these performance properties is a prerequisite for performance tuning. This paper explores a new approach to discovering performance properties of system configurations: static program analysis. We present a comprehensive taxonomy of how a configuration might affect performance through program dependencies. Guided by this taxonomy, we design LearnConf, a static analysis tool that identifies which configurations affect what type of performance and how. Our evaluation considers hundreds of configurations in four widely used distributed systems, demonstrating that LearnConf accurately and efficiently identifies configurations’ performance properties, and improves performance tuning while avoiding problems such as out-of-memory errors and timeouts.
CHAPTER 1
INTRODUCTION

1.1 Motivation

Software configuration plays a critical role in performance tuning, with system throughput and memory consumption varying widely under different configuration settings even for the same workload [30]. In practice, configuration is often the only mechanism that end-users have to manage performance across workloads, platforms, and usage goals [3]. Unfortunately, appropriately setting configurations to achieve specific goals is difficult, as the configuration space is huge—with 100s or 1000s of parameters, each of which takes a wide range of values—and the relationship between configuration settings and the resulting performance is often unclear without trial and error. Empirical studies find that performance issues contribute to 50% of configuration-related patches in open-source cloud systems and 30% of configuration-related forum questions [30]. In cloud systems, such mis-configurations have caused severe performance problems and outages, costing hundreds of millions of dollars [11, 16]. Clearly users and administrators need new tools to help understand how to configure these complicated software systems.

Figure 1.1 and Figure 1.2 illustrates the challenge of understanding performance-related configurations through an example from HDFS, a widely-used distributed file system. The documentation shown in Figure 1.1 describes a configuration parameter, \( \texttt{dfs.namenode.max.objects} \), as limiting the number of objects in the file system, with 0 indicating no limit. From the documentation, it is easy to understand the functional difference between a 0 a non-0 setting, but the performance difference is unclear and undocumented. It is only by examining this configuration’s usage in the code (Figure 1.2) that we can see the performance difference: if the setting is non-0, then a lock is acquired. This code executes for every user write request, and thus lock acquisition can greatly affect user latency. If the user/administrator changes the configuration from non-0 to 0, the write latency will
<name>dfs.namenode.max.objects</name>
<description>The maximum number of files, directories and blocks. A value of zero indicates no limit to the number of objects that dfs supports. </description>

Figure 1.1: Configuration file hdfs-default.xml

```java
1 if (maxFsObjects != 0) {
2   lock();
3   if (totalNodes() > maxFsObjects) {...}
4   unlock();
5 }
```

Figure 1.2: Performance related code using the configuration

suddenly drop, which will be difficult to interpret without a deep dive into the code.

As the example shows, configurations are associated with a rich set of performance properties. To properly tune the system, a user must understand all configuration’s properties including — but not limited to — the performance metric (e.g., memory or latency) affected; the type of user requests affected; the range of acceptable values for the configuration; and the range of affects on the actual performance. It is clearly unrealistic to expect users to trace the configuration parameters through the code to understand all these performance properties.

Much prior work identifies configuration issues that affect functional correctness. Some approaches identify statistically abnormal settings [29, 31, 42, 43] by comparing many users’ settings for the same configuration parameter. Some work uses program analysis [38, 8] to identify the desired data type or value range of a configuration to avoid exceptions and to identify configurations that have dependencies with software failure sites [26, 37, 4, 44]. These techniques are inspiring but cannot be applied to understanding configurations’ performance properties as the performance impact typically has no relationship with the types of functional behaviors — e.g., exception throwing or fail-stop errors — explored by these prior works.

Some previous work applies machine learning [45, 41] and control theoretic techniques [30, 23, 17, 18] to automatically find performance-appropriate configuration settings. Learning
and control approaches both rely on intensive profiling and training to build models, requiring access to profiling inputs at design time. Thus, the relationships they discover between performance and configuration parameters are only valid if the runtime behavior is within some known factor of the design time behavior. If the workload varies considerably or the users set a configuration to some extreme values not exercised during profiling, the models these systems rely on will be insufficient to deliver the required performance [13, 9]. Furthermore, such offline profiling or training requires significant time to collect the necessary measurements [41], and the time grows exponentially with the number of configurations to be modeled.

Overall, there is a need for novel techniques that automatically determine some performance properties of configurations independent of inputs or statistical profiles. Such an approach would complement existing software documentation and profiling techniques to help both users and automated tools configure software systems for performance.

1.2 Contribution

This paper proposes static analysis techniques for understanding configuration parameter’s relationships to observed performance. Static analysis can play an important role in this process because—ultimately—it is the program logic that determines how configuration affects performance.

Our key insight is that any configuration that affects performance dynamically must have a data- or control-flow dependency with certain time- or space-intensive operations, which we refer to as Performance Operations or PerfOps. Consequently, performance impacts must be fully or partially reflected by static program structures and data/control dependency relationships, and thus identifiable through static analysis.

Following this insight, we design a taxonomy of static program dependency structures. This taxonomy comprehensively summarizes how a configuration setting affects: (1) the performance impact from one dynamic instance of a PerfOp; (2) whether or not a PerfOp
executes at run time; or (3) the frequency or the number of times a PerfOp executes at run time. The taxonomy is sufficiently detailed so that once we know a configuration can affect a specific PerfOp following one of the patterns, we immediately know many detailed performance properties. These patterns are presented in Sec. 2.

Guided by our taxonomy, we design LearnConf, a static analysis tool that automatically identifies configurations that have performance impacts, referred to as PerfConfs, and infers the detailed properties of a PerfConf. LearnConf works in three steps, as shown in Figure 1.3. First, LearnConf uses static data and control flow analysis to automatically identify configurations (PerfConfs) that affect PerfOps. Second, given a pair of a PerfConf and the PerfOp it affects, LearnConf analyzes the code on the PerfConf-PerfOp dependency chain to categorize it according to our taxonomy. Third, LearnConf conducts further pattern-specific analysis to determine a PerfConf’s detailed performance properties. These additional details include a variety of information useful for performance tuning including: whether the relationship between configuration and performance is linear or monotonic, whether the configuration interferes with other configurations, and whether the configuration affects user requests or systems services.

We evaluate LearnConf by applying it to four widely used open-source systems, HDFS, HBase, Cassandra, and MapReduce. LearnConf static analysis automatically identifies 69 PerfConfs that affect user-facing job performance. We carefully compare this result with both software documentation and configurations manually picked by prior work for performance tuning [21, 19, 7, 34, 5, 35, 6, 45, 27, 33, 1]. We find that LearnConf has a low false negative
rate of 15% (correctly identifying 60 out of 71 true user-facing PerfConfs) and a low false positive rate of 13% (only 9 out of the identified PerfConfs have no performance impact). In comparison, we find that among the configurations manually identified by prior work, 15% of them actually have no performance impact. Furthermore, prior approaches fail to identify 17 true PerfConfs that can lead to out-of-memory or more than 10% latency changes.

We also conduct in-depth case studies for 20 PerfConfs, which demonstrate that LearnConf can indeed statically predict the dynamic performance properties of system configurations, accurately predicting the type of performance impact and quantitative relationship between the PerfConf’s value and the corresponding Performance metric. Finally, our experiments show that LearnConf improves profiling-based techniques for tuning PerfConfs by avoiding problems caused by incorrectly trained models.
CHAPTER 2
PERFORMANCE IMPACT TAXONOMY

Our taxonomy of program-dependence relationships between a PerfConf and a performance-intensive operation (PerfOp)\(^1\) includes three high-level categories:

1. A data dependence between the PerfConf and a PerfOp parameter (Section 2.1);

2. An if-related control dependence where the PerfConf helps determine whether the PerfOp is executed (Section 2.2);

3. A loop-related control dependence where the PerfConf controls the number or frequency of PerfOp executions (Section 2.3).

Table 2.1 provides one toy example for each of the detailed patterns that belong to one of these three categories. Formulas and figure illustrations about how the performance impact of the toy-example code snippet might change under different configuration setting \(C\) are also shown in the table, and will be elaborated below.

2.1 Data Dependency

Here the configuration’s value (or a derivative) is passed as a parameter to the PerfOp and hence affects every dynamic instance of the PerfOp.

In cases when the corresponding PerfOp-parameter value has a linear relationship with the PerfOp’s performance contribution (e.g., for \texttt{sleep}, \texttt{malloc}, etc.), the PerfConf also has a simple, often linear, relationship with the corresponding performance metric, as shown in Table 2.1 Figure 1.3.

\(^1\) We define a PerfOp as an instruction or an API call that is particularly time-intensive (e.g., \texttt{sleep}, \texttt{lock}, I/O) or memory-intensive (e.g., \texttt{new}, \texttt{malloc}). We detail what APIs are treated as PerfOps by \texttt{LearnConf} in Section 3.1.2.
Table 2.1: Configurations’ Performance-Impact Taxonomy. To ease the discussion, all the formulas, figures, and explanations in the table focus on the Toy Examples, where \( C \) is the value of a configuration and Perf refers to the memory-consumption contribution of one instance of the toy-example code snippet. The \( \text{Perf—Conf} \) figure depicts how the Perf (the y-axis) might change with the configuration setting (the x-axis); the \( \text{Perf—Time} \) figure depicts how the Perf (the y-axis) might change over time (the x-axis) under different configuration settings, C1 (blue dots) and C2 (red crosses).

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Toy Example</th>
<th>Formula</th>
<th>Perf—Conf</th>
<th>Perf—Time</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Dependency — PerfConf affects the amount of performance contribution from one PerfOp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Data</td>
<td>malloc(( C ))</td>
<td>Perf = ( C )</td>
<td>( C ) affects the impact of one malloc() linearly (1.3)</td>
<td></td>
</tr>
<tr>
<td>Control Dependency (IF) — PerfConf affects whether a PerfOp is executed</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>with Constant</td>
<td>if (( V &lt; C )) malloc(a); else malloc(b);</td>
<td>Perf = ( a ) ( \text{if} \ (\text{if} \ \text{if}) ) ( b )</td>
<td>( C ) affects whether execute malloc(a) or malloc(b) statically (2.3)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>with getting closer var</td>
<td>if (( V &lt; C )) { ( V++ ); malloc(a); } else malloc(b);</td>
<td>( a ) ( \text{if} \ (\text{if} \ \text{if}) ) ( b )</td>
<td>( C ) affects when to switch from malloc(a) to malloc(b) (3.4)</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>with back&amp; forth var.</td>
<td>if (( V &lt; C )) { ( V++ ); malloc(a); } else { ( V-- ); malloc(b); }</td>
<td>( a ) ( \text{if} \ (\text{if} \ \text{if}) ) ( b )</td>
<td>( C ) affects how quickly switch from malloc(a) to malloc(b) (4.4)</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>with unrelated var.</td>
<td>if (( V &lt; C )) malloc(a); else malloc(b);</td>
<td>Perf = ( a )</td>
<td>( c_1 ) ( \text{if} \ (\text{if} \ \text{if}) ) ( c_2 )</td>
<td>( C ) affects the probability of malloc(a) executes over malloc(b) (5.4)</td>
</tr>
<tr>
<td>Control Dependency (LOOP) — PerfConf affects the frequency/#-of-times a PerfOp executes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Regular loop bound</td>
<td>for(( i &lt; C ); i++ { ( \text{if} \ (\text{if} \ \text{if}) ) ( \text{malloc(a);} ) }</td>
<td>Perf = ( a ) ( \text{if} \ (\text{if} \ \text{if}) ) ( C )</td>
<td>( C ) affects the number of malloc(a) via loop bound (6.3)</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Regular loop stride</td>
<td>for(( i &lt; N ); i++ { ( \text{if} \ (\text{if} \ \text{if}) ) ( \text{malloc(a);} ) }</td>
<td>Perf = ( a ) ( \text{if} \ (\text{if} \ \text{if}) ) ( N / C )</td>
<td>( C ) affects the number of malloc(a) via loop stride (7.3)</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Sync loop</td>
<td>while (( i &lt; C )) { ( \text{wait(); } )</td>
<td>Perf = ( a )</td>
<td>( c_1 ) ( \text{if} \ (\text{if} \ \text{if}) ) ( c_2 )</td>
<td>( C ) affects the frequency of malloc(a) via synchronization loop (8.4)</td>
</tr>
<tr>
<td>9</td>
<td>Infinite loop</td>
<td>malloc(a); sleep(( C )); //in infinite loop</td>
<td>Perf = ( a )</td>
<td>( c_1 ) ( \text{if} \ (\text{if} \ \text{if}) ) ( c_2 )</td>
<td>( C ) affects the frequency of malloc(a) via a fixed interval (9.4)</td>
</tr>
</tbody>
</table>
2.2 IF-related Control Dependency

In this category, a variable $C$ derived from a PerfConf is used in an if-condition predicate, whose evaluation picks from code paths containing different PerfOps. Consequently, the PerfConf setting affects which PerfOp is executed. Suppose the PerfOps on two code paths have performance contributions $a$ and $b$, then the execution of one instance of the if-statement is a piece-wise function, as shown in Eq. 2.2 and Figure 2.3 in Table 2.1. If the if-statement is executed multiple times, the aggregated performance impact also depends on how the comparison with $C$ changes over time. We identify four such patterns of change.

2.2.1 Compared with a constant

The PerfConf-derived variable $C$ is compared with a constant variable $V$ in the predicate, with neither $C$ nor $V$ ever changing values after their initial assignment. Therefore, the PerfConf has a range effect on performance because the corresponding if-else code is only executed for a particular range of PerfConf settings. In part of the range, the PerfConf has one effect on performance, but when the PerfConf is moved across the range boundary it suddenly has a very different effect. In Table 2.1 Figure 2.1, we illustrate the simple and also common scenario where the $C$’s value is exactly PerfConf and the performance output (e.g., memory consumption) of the if/else branch is constant. Once the configuration is set, the program always takes one branch and has the same performance impact (Figure 2.4).

In this pattern, the PerfConf statically directly determines whether or not certain PerfOps will be executed.

2.2.2 Compared with a getting-closer variable

The PerfConf-derived variable $C$ is compared with a variable $V$ whose value changes, moving towards $C$, in one branch of the if-else structure and does not change in the other branch. Looking at one dynamic instance of this if-else code structure, its performance output de-
pends on $V$’s current value in Table 2.1 Figure 2.3. Looking at a long execution including multiple instances of this if-else structure, $V$ approaches $C$ and eventually causes the if-else predicate’s result to flip and never flip back—the performance output of every dynamic if-else instance eventually switches from one set of PerfOps to the other set, as illustrated in Figure 3.4 in Table 2.1.

In this pattern, such a switch only happens once, and the PerfConf affects when the switch occurs.

### 2.2.3 Compared with a back-and-forth variable

The PerfConf-derived variable $C$ is compared with $V$ whose value changes in different directions in the two branches of the if-else structure. Thus, when the conditional is executed multiple times, $V$’s value goes up and down, causing the predicate’s outcome, and hence the performance output, of each if-else instance to switch back and forth (Figure 4.4 in Table 2.1).

In this pattern, the PerfConf’s value controls how quickly the switch occurs.

### 2.2.4 Compared with an unrelated variable

The PerfConf-derived variable $C$ is compared with a variable $V$ whose value is not conditioned on $C$. Typically, $V$’s is related to workload, system status, or some other external property.

In this pattern, the PerfConf’s setting affects the long-term probability of one set of PerfOps being executed over the other, and hence, the expected long-term performance outcome of the if-else code, as shown in Fig. 5.4 in Table 2.1.

**Note** In theory, there could be other patterns of this IF-related control category. For example, the value of $C$ could also change over the time, or the value of $V$ might change in both branches of the if-else structure but in the same direction. These cases are rare in practice.
and can be easily converted to some of the patterns discussed above.

2.3 LOOP-related Control Dependency

In this category, a variable $C$ derived from the PerfConf is used inside a loop to control the bound, stride, or other features that hence affects the number or frequency of PerfOp execution.

2.3.1 Affecting loop bound

$C$ is used in the bound expression of a PerfOp-containing loop. Figure 6.1 from Table 2.1 depicts a simple scenario, where the performance output of the loop-enclosed PerfOp is roughly constant, the loop stride is constant, and the bound is linear in $C$. In this case, the relationship between the PerfConf and the corresponding performance metric is linear (Formula 6.2 and Figure 6.3). This simple case is common in practice, as we will see in Section 5.

2.3.2 Affecting loop stride

$C$ is used to set the loop stride, as shown in Figure 7.1 of Table 2.1. Here, a higher PerfConf value reduces the executions of a corresponding PerfOp as shown in Figure 7.3 in the table.

2.3.3 Affecting synchronization-loop

Sometimes, $C$ is the bound of a synchronization loop (e.g., a spin lock in Figure 8.1 of Table 2.1) with a PerfOp executed once the loop exits. In this case, the loop’s index variable does not change inside the loop, but can be set by another thread. The bound variable, $C$, helps decide how long the thread needs to wait until it can execute the PerfOp after the loop. The PerfConf setting here decides how frequently a performance burst occurs, as illustrated in Figure 8.4 in the table.
2.3.4 Affecting infinite-loop

As illustrated in the toy example in Figure 9.1 of Table 2.1, some system threads contain infinite loops that execute until system shut down. In this case, $C$ cannot affect when the loop exits but may affect the execution time of an operation inside the loop (e.g., through data-dependency), and hence affect the frequency of all other PerfOps inside the loop. The Perf-Conf setting here decides the frequency of PerfOps that are executed periodically throughout system lifetime, as shown in Figure 9.4 in the table.
CHAPTER 3
PERFCONF AND PATTERN IDENTIFICATION

As shown in Figure 1.3, LearnConf analyzes Java byte code and outputs a list of performance-related configurations and their performance properties. To achieve that, LearnConf first identifies configuration variables (Section 3.1.1) and PerfOps (Section 3.1.2) in the target system. LearnConf then analyzes the dependency between every PerfConf-PerfOp pair to categorize it into one of the 9 patterns in our taxonomy (Section 3.2). Finally, LearnConf performs pattern-specific analysis to determine detailed performance properties (Section 4).

3.1 Identifying PerfConfs and PerfOps

3.1.1 What are configuration (derived) variables?

LearnConf first identifies all the invocations of configuration-loading APIs (e.g., all the getInt, getFloat, and other APIs inside Hadoop’s Configuration class) in the software, and tags the return values of these API calls as initial members of the configuration-variable set. LearnConf then keeps adding into this set with variables that have data dependency with any variable already in the set, until reaching a fix point. LearnConf does not track control dependency at this step, except for one case: if the assignment of a boolean variable $V_b$ depends on $C$, we consider $V_b$ to be derived from $C$, because $V_b$ may then be used for an if/loop predicate, which will be equivalent to $C$ being directly involved in the predicate.

Identifying variables that have data dependency on a variable $C$ is straightforward when $C$ is a stack variable. LearnConf simply conducts path-sensitive dependency analysis inside the function holding $C$; when $C$ is the parameter or the return value of a function, LearnConf conducts similar path-sensitive analysis inside the callee or caller function.

Things get harder when $C$ is a heap variable. In this case, in theory, the program analysis cannot be limited to one function — we may need to use expensive alias analysis to first
identify all references of $C$ and then analyze program dependency accordingly, which would be difficult to scale.

In LearnConf, we simplify the dependency analysis related to heap variables leveraging the common usage pattern of configuration variables — a configuration variable is usually used to compute the value of either a stack variable of a primitive type or a dedicated configuration-field of an object (e.g., LruBlockCache.maxSize, HRegion.flushSize, etc). Consequently, when the variable $C$ is a field $f$ of a heap object $obj$ of class $CL$, LearnConf considers the field $f$ of all objects of class $CL$ as a configuration variable, and does not conduct any alias analysis. In practice, we have found our design to well balance accuracy and analysis complexity, as will be demonstrated in the evaluation.

3.1.2 What are performance operations?

To check which configuration variables are used to affect performance, we first need to decide which code snippets should be considered as performance-intensive operations (PerfOps). Below, we discuss how LearnConf decides memory-expensive operations and time-expensive operations.

The main challenge here is that many, if not all, instructions in a program make positive contributions to the execution time and memory consumption. It is meaningless to consider all of them as PerfOps. Therefore, LearnConf identifies a set of operations that are likely to have large performance impacts at run time, using simple static analysis that scales to large system software. LearnConf does not aim to be free of false positives or false negatives, which is infeasible given the nature of this task.

Memory Operations Many operations can lead to a temporary memory-consumption increase. To identify operations that can potentially lead to long-lasting, large impacts, LearnConf focuses on two types of operations related to arrays. First, a heap or static array allocation instruction that allocates an array with a non-constant array length (e.g., new CLASS[V]). Second, a container-add operation that adds the reference of an array, whose
content comes from an I/O-library operation and hence has a likely non-constant size but untrackable allocation site, into a heap or static container. Furthermore, to avoid double counting, LearnConf analysis makes sure that the added array obj is only referenced by one container at a time, which is done by LearnConf using the default pointer-alias analysis [2] provided by WALA[32].

Latency operations We consider several types of operations as time-expensive, including (1) operations that explicitly cause a thread to pause, including Thread::sleep() and all lock-synchronization APIs (i.e., Object::wait(); (2) Java I/O-library operations; (3) operations that directly affect the parallelism level of the system, including new Thread(), new ThreadPoolExecutor(), etc.; (4) a configurable list of expensive operations in distributed systems, which currently only includes heartbeat() functions in LearnConf.

3.1.3 What are performance-related configurations?

After identifying configuration variables and PerfOps, LearnConf then analyzes whether any PerfOp has data or control dependency upon any configuration variable. If so, the corresponding configuration is identified as a performance-related configuration (PerfConf). LearnConf will then feed the configuration variable and the corresponding PerfOp into its pattern-identification component (Section 3.2).

This analysis is actually done in the same pass as LearnConf identifies configuration variables. When LearnConf identifies a function parameter as a configuration variable, it checks whether this function is a PerfOp. If so, the corresponding configuration is identified as a PerfConf. Similarly, when LearnConf identifies a variable that is part of a control-flow predicate, LearnConf further analyzes the corresponding if-else/loop body, including k levels of callee functions (Default k is 2), to see if there is any PerfOp enclosed. If so, the configuration is identified as a PerfConf.
3.2 Identifying PerfConf patterns

Data-dependency pattern Whenever a configuration variable appears as the parameter of a PerfOp, LearnConf identifies such a data-dependency pattern.

There is just one exception: if the performance operation is sleep or wait, LearnConf further checks whether the sleep or wait is inside an infinite loop (i.e., the outcome of the loop-exit condition is not changed inside the loop body). If so, LearnConf considers this as an Infinite Loop pattern.

Control-dependency IF patterns When a configuration variable $C$ is used in the predicate of an if-statement, LearnConf checks the variable $V$ that is compared with $C$. If $V$ is a constant, a “Compared with constant” pattern is caught. Otherwise, the value of $V$ must be changed somewhere in the program, outside or inside the if-else body, which LearnConf analyzes to make further categorization following the definition in Section 2: when $V$ is not changed in either the if-branch or the else-branch, this is a “Compared with unrelated variable” pattern; when $V$’s value is only changed in the if-body or the else-body, but not both, this is a “Compared with getting-close variable” pattern; otherwise, this is a “Compared with back-and-forth variable” pattern.

Control-dependency LOOP patterns When a configuration variable $C$ is used in a loop-exit condition predicate, where it is compared with a variable $V$, LearnConf conducts a set of checking about $C$ and $V$ to see which LOOP pattern this belongs to. (1) If neither $V$ nor $C$ is updated in the loop body, this is identified as a synchronization loop (“Synchronization loop bound” pattern); (2) If $V$ is updated in the loop body and yet $C$ is a loop invariant, LearnConf considers the corresponding PerfConf to affect the loop bound (“Regular loop bound pattern”); (3) If $C$ is updated in the loop body, LearnConf checks the difference between the values of $C$ before and after the update. If the difference depends on the value of PerfConf, LearnConf considers the PerfConf to affect how an index-variable changes its value across iterations (“Regular loop stride” pattern); otherwise, LearnConf considers the
PerfConf to affect the loop bound.
After the analysis above, LearnConf has discovered a list of PerfConfs. For each PerfConf, LearnConf has identified PerfOps that the PerfConf affects through a specific pattern. In this section, LearnConf infers additional performance properties that help users understand and tune PerfConfs.

4.1 Input Analysis

Many PerfConfs affect system performance under specific inputs/workloads. Knowing which user input/workload is affected by a PerfConf is critical for performance tuning.

LearnConf analysis starts from every user-request entry function $F_u$ and then identifies all functions $F_u$ that can be invoked directly or indirectly by $F_u$. A PerfConf-PerfOp pair is determined to affect user request $u$ if the PerfOp is inside a function in $F_u$. Otherwise, it is determined to affect background services.

Identifying user-request entry functions is straightforward, as they are typically well-defined RPC functions in a dedicated client-server interface class in distributed systems (e.g., in HDFS, the ClientProtocol interface class defines all user-request entry functions). To identify $F_u$, LearnConf initializes it with the entry function. It then extends the set until reaching a fixed point based on three rules: (1) if a function $f'$ is invoked by a function $f \in F_u$, $f'$ also belongs to $F_u$; (2) if a function $f'$ is an RPC function that can be invoked by an RPC-call in $f \in F_u$, $f'$ also belongs to $F_u$; (3) if a function $f \in F_u$ starts a new thread (e.g., through Thread::start), the entrance function of the corresponding child thread (e.g., a Thread::run function) also belongs to $F_u$.

Finally, we should note that the above analysis works well for PerfOps that affect execution latency, but is unsuitable for PerfOps that affect memory consumption or thread creation—these PerfOps’ impact tends to go beyond one thread and would be considered by
LearnConf to affect both user requests and background services.

4.2 Slope Analysis

For a PerfConf that has a roughly linear relationship with a performance metric (i.e., $P = \alpha \times \text{Conf} + \beta$), knowing the exact slope of this linear relationship (i.e., $\alpha$) is useful for performance tuning and satisfying performance constraints. Prior work typically obtains this information from extensive profiling [30, 9, 10].

There are four patterns from Table 2.1 which are most likely to produce a linear relationship: Data dependency, Regular loop bound, Regular loop stride (inverse linear), and Infinite loop (inverse linear). For these patterns, the parameter of a PerfOp,\(^1\) or the bound or index or frequency of a loop that encloses a PerfOp has a data dependency upon the PerfConf. Consequently, LearnConf simply extracts the data-dependence slice deriving such parameter or bound or index from the PerfConf and applies symbolic evaluation to that slice to obtain an expression $f(\text{conf})$, like $\text{sortmb}*1024*1024$ for the example in Figure 4.1 and $\text{timeout} * \text{Math.pow}(2, \text{attempts})$ for the example in Figure 4.2 (LearnConf considers binary operations, unary operations, and Java Math library functions in its evaluation). Given such an expression, LearnConf easily outputs the slope, which could be a constant like $1024 * 1024$ for configuration sortmb in Figure 4.1 or a symbolic expression with program variables like $2^{\text{attempts}}$ for configuration timeout in Figure 4.2.

Note that, it is possible that a PerfConf affects more than one PerfOp, and LearnConf handles this by further checking whether multiple PerfConf-PerfOp pairs are on the same path. We consider two cases. (1) If a PerfConf-PerfOp pair does not share its execution path with other PerfOps that also depend on this PerfConf, its slope analysis result can be directly reported. (2) If multiple PerfConf-PerfOp pairs (for the same PerfConf) are on the same program path and affect the same performance metric, the performance impact

---

\(^1\) LearnConf assumes an input list of PerfOp APIs, as well as annotation about which parameter of a PerfOp API has a roughly linear relationship with the performance contribution of this PerOp.
int maxUsage = sortmb * 1024 * 1024;
buffer = new Byte[maxUsage];

Figure 4.1: iosortmb has constant slope

while (!stopped) {
    wait = timeout * Math.pow(2, attempts);
    sleep(wait);
}

Figure 4.2: timeout has non-constant slope

of this PerfConf along this path should consider all these pairs together. If all these pairs indicate linear relationship, the combined impact is still linear; otherwise, LearnConf will not produce a slope.

4.3 Configuration Setting Range Analysis

In some cases, changing a PerfConf’s setting does not affect system performance unless the change moves across a range boundary, like that in the “Compared with constant” pattern. In this case, LearnConf extracts the \( C \text{ op const} \) predicate, symbolically replaces \( C \) with \( f(Conf) \) (i.e., how \( C \)’s value is derived from the PerfConf, computed in a way similar as that in Slope Analysis), and outputs the constraint expression involving the PerfConf setting.

In some cases, the program logic imposes a valid range for a PerfConf. To identity such a range, LearnConf adopts a similar approach as previous work [38]. When a configuration variable \( C \) is used in an if-predicate, LearnConf checks whether an exception is raised or \( C \) is reset with another value in the if-else body. Different from previous work, LearnConf not only considers constant values [38], but also symbolic values with program variables.

4.4 Configuration Relation Analysis

Sometimes, multiple configurations may work together to affect a PerfOp. This information can help a performance tuner to group these configurations together in tuning.

LearnConf identifies two PerfConfs \( C_1 \) and \( C_2 \) as related by checking whether they may
if (memStoreSize >= upperLimit) {
    for (; memStoreSize > lowerLimit;) {
        flushRegion(); // lock and I/O
    }
}

Figure 4.3: upperLimit enables lowerLimit

for (; tries < numRetries; ++tries) {
    Thread.sleep(sleepTime);
}

Figure 4.4: numRetries and sleepTime work simultaneously

affect the same PerfOp along the same path. LearnConf further categorizes their relationship by comparing the PerfConf-PerfOp dependency chains. (1) If the configuration variable of $C_1$ along its PerfConf-PerfOp dependency chain is affected by an if-predicate containing a configuration variable of $C_2$, $C_1$ only takes effects when $C_2$ enables so. For example, in Figure 4.3, PerfConf lowerLimit takes effect only when memStoreSize is larger than another PerfConf upperLimit. (2) Otherwise, $C_1$ and $C_2$ take performance effect simultaneously.

In Figure 4.4, numRetries and sleepTime determine the number of iterations and the time spent on each iteration to affect user latency.

### 4.5 Monotonicity Analysis

A basic requirement for performance tuning is knowing whether increasing a PerfConf will cause the corresponding performance metric to go up or down, or sometimes-up-sometimes-down.

From Table 2.1 — particularly the figures in column Perf-Conf — it is straightforward to determine whether changes in the PerfConf directly affect the PerfOp and whether they are positive or negative. The remaining questions are whether the effects are still monotonic when: (1) the PerfConf has range effect; and (2) one PerfConf affects multiple PerfOps.

LearnConf handles range effects in two steps. It begins with the symbolic expression $C = f(conf)$ and infers a non-monotonic relationship in two cases: (1) $f(conf)$ is fragmented...
like $f = (conf < A) ? f_1 : f_2$ where $f_1$ and $f_2$ are two conflicting mathematical relationships between $f(conf)$ and $conf$; (2) $f(conf)$ simply contains a non-monotonic mathematical relationship like a cubic function. In the second step, LearnConf checks uses of $C$ in if-predicates to determine whether an increase of $C$ could lead to more PerfOp executions in two disjoint regions, like if $(C!=a)$ PerfOp(); or if $(C<a \text{ || } C>b)$ PerfOp();.

Finally, for simplicity, assume that LearnConf finds two PerfOps of the same performance type (e.g., latency), PerfOp1 and PerfOp2, that both depend on the same PerfConf along the same program path. LearnConf then analyses the PerfConf-PerfOp1 and PerfConf-PerfOp2 relationships independently. If the result is inconsistent, LearnConf cannot declare the relationships monotonic. This situation can arise if increasing the PerfConf causes PerfOp1 to execute more and PerfOp2 to execute less, for example.
CHAPTER 5
EVALUATION

We have implemented LearnConf using WALA [32], a static analysis infrastructure for Java bytecode. We evaluate LearnConf on four open source distributed systems, Hadoop Distributed File System (short as HD), the distributed key-value store databases HBase (short as HB) and Cassandra (short as CA), and the distributed computation framework MapReduce (short as MR). These systems each contain around 100 to 150 configurations (477 altogether) in their default configuration files (e.g. hbase-default.xml, hdfs-default.xml, etc.).

We run all the experiments on machines with 32-core Intel Xeon E5-2620v4 @ 2.10GHz, and 128GB RAM, with Ubuntu 14.04.6 LTS and JVM v1.8.

5.1 User-facing PerfConf Identification

Benchmark configurations To compare with LearnConf, we collect PerfConfs from two alternative sources: (1) configurations used by prior work on performance tuning, and (2) configurations mentioned in software performance-tuning guides/tutorials [33, 1], written by software experts. For the first source, we consider a large number of prior works on performance tuning, including one that identifies PerfConfs through intensive profiling and statistical modeling [21] and many others where the respective authors manually select configurations that they believe might have performance impact and are hence worth tuning.

<table>
<thead>
<tr>
<th></th>
<th>LearnConf</th>
<th>Source1</th>
<th>Source2</th>
<th>Source3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>All</td>
<td>F+</td>
<td>F-</td>
<td>All</td>
</tr>
<tr>
<td>MR</td>
<td>16</td>
<td>1</td>
<td>7</td>
<td>14</td>
</tr>
<tr>
<td>HB</td>
<td>19</td>
<td>1</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>HD</td>
<td>13</td>
<td>5</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>CA</td>
<td>21</td>
<td>2</td>
<td>1</td>
<td>24</td>
</tr>
<tr>
<td>Total</td>
<td>69</td>
<td>9</td>
<td>11</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 5.1: User-facing PerfConfs reported by LearnConf and those used by other sources. The $k$-th sources for different applications are different and hence are not summed together; F+: false positives; F-: false negatives.
[19, 7, 34, 5, 35, 6, 45, 27]. For each of the 4 software systems, we ensure that we have at least three alternative sources of PerfConfs.

**Overall results** As shown in Table 5.1, LearnConf identifies 69 user-facing PerfConfs (out of the total 477 configurations) with only 9 false positives and 11 false negatives across all systems. In comparison, all alternative sources suffer from many more false negatives, with more false negatives than true PerfConfs in most cases. Furthermore, LearnConf finds 17 true PerfConfs that did not show up in any prior sources; we were able to run experiments for 13 out of these 17 PerfConfs to consistently trigger significant performance differences, which we elaborate below.

Note that many performance tuning works focus on minimizing the latency of certain benchmarks, such as Terasort latency [19] or bulk loading latency [5]. Therefore, they may focus on only PerfConfs related to certain workloads, and most do not consider memory consumption in their tuning model. This explains why some works have higher false negatives. In comparison, LearnConf finds all PerfConfs, even if their effects are triggered only for some inputs or workloads, which are not necessarily the ones profiled by prior performance-tuning work. In this sense, LearnConf and profiling-based methods are complementary: LearnConf could find an initial set of PerfConfs to pass to a profiling-based statistical method that can then rank the dynamic impact of those configurations for specific workloads.

**True positives** LearnConf identifies 17 PerfConfs that have not been identified by previous configuration tuning works or documentation. We break them into three categories: (1) For 4 PerfConfs, we experimentally confirm that their settings can affect memory consumption or latency so much that out-of-memory or timeout errors were triggered. (2) For 9 PerfConfs, our experiments consistently trigger more than 10% performance difference using two different configuration values. For example, `native_transport_max_threads` determines the number of handler threads in Cassandra, and can have up to 7.3X latency difference under two different configuration settings in our experiments. However, this configuration is not used by any performance tuning works and is even missed by exhaustive profiling [21]. (3) For
4 PerfConfs related to lock operations, we confirm the configurations do affect the number of lock acquisitions. However, we were unable to consistently trigger significant performance differences between different configuration values.

**False positives** There are 9 false positives in four categories. (1) In 4 cases, LearnConf finds configurations that affect an array’s size, but the code applies a bound so that it does not cause significant memory consumption differences. (2) Another 2 false positives are caused by configurations affecting latency related PerfOps, but in an asynchronous path or in a path that is executed after the job terminates. (3) One case is caused by Java Polymorphism not handled by LearnConf. (4) Two configurations affect the creation of background threads that have little effect on user-request latency.

**False negatives** There are 11 false negatives in 3 categories. (1) Nine have complicated control dependencies, which could not be captured by LearnConf. For example, a configuration may affect the type of compression object created, which affects which compression algorithm is used. (2) For one case, the configuration affects memory consumption by re-assigning the reference variable so that the object previously referenced can be garbage collected. (3) In the final case, the configuration is a parameter to a child JVM process.

### 5.2 Performance-pattern Identification

**Does LearnConf find the correct pattern?** In total, LearnConf has identified 118 PerfConfs from these 4 systems, which correspond to 405 PerfConf-PerfOp pairs. Table 5.2 summarizes the pattern distribution of these 405 PerfConf-PerfOp pairs. We manually

<table>
<thead>
<tr>
<th></th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>P5</th>
<th>P6</th>
<th>P7</th>
<th>P8</th>
<th>P9</th>
</tr>
</thead>
<tbody>
<tr>
<td>MR</td>
<td>21</td>
<td>27</td>
<td>0</td>
<td>8</td>
<td>9</td>
<td>18</td>
<td>2</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>HB</td>
<td>20</td>
<td>11</td>
<td>1</td>
<td>3</td>
<td>30</td>
<td>37</td>
<td>1</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>HD</td>
<td>20</td>
<td>16</td>
<td>3</td>
<td>6</td>
<td>14</td>
<td>10</td>
<td>0</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>CA</td>
<td>18</td>
<td>0</td>
<td>51</td>
<td>1</td>
<td>25</td>
<td>13</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Total</td>
<td>79</td>
<td>54</td>
<td>55</td>
<td>18</td>
<td>78</td>
<td>78</td>
<td>3</td>
<td>3</td>
<td>37</td>
</tr>
</tbody>
</table>

Table 5.2: PerfConf-PerfOp pattern distribution (P_k indicates the k-th pattern listed in Table 2.1).
<table>
<thead>
<tr>
<th></th>
<th>Tot.</th>
<th>Correct</th>
<th>False Pattern</th>
<th>False Taint</th>
</tr>
</thead>
<tbody>
<tr>
<td>MR</td>
<td>94</td>
<td>90</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>HB</td>
<td>112</td>
<td>96</td>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>HD</td>
<td>90</td>
<td>85</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>CA</td>
<td>109</td>
<td>107</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>405</td>
<td>378</td>
<td>22</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 5.3: PerfConf-PerfOp pattern identification accuracy.

We categorize all the incorrectly identified patterns to two types. First, false taints—LearnConf incorrectly identifies configuration variables due to special data dependences. For example, two HDFS configurations replication and blockSize are stored as bits in the header of INodeFile. LearnConf cannot distinguish them when the program tries to read different bits from the header. Second, false patterns, which mostly occur for IF control-dependence patterns, where inaccurate alias analysis sometimes causes LearnConf to miss variable updates.

**Does the pattern correctly capture performance behaviors?** We run experiments for at least one PerfConf for every pattern. Our experimental results are shown in Figure 5.1-5.9\(^1\): all match the pattern-specific behaviors discussed in Section 2.

---

\(^1\) For some experiment, we keep sending requests and measure latency or memory consumption after every request. Thus request (x-axis) in this graph resembles time (x-axis) in our theoretical graph in Table 2.1.
Figure 5.2: If - w/ Constant: maxFsObj

Figure 5.3: If - w/ getting closer: buffer_size

Figure 5.4: If - w/ back&forth: bufferSize

Figure 5.5: If - w/ unrelated: commitlogSize
Figure 5.6: Regular loop bound: lowerLimit

Figure 5.7: Regular loop stride: splitSize

Figure 5.8: Sync loop: blockingSize

Figure 5.9: Infinite loop: msgInterval
Data dependency: PerfConf sortmb is used as a parameter to create a byte array, as shown in Figure 5.1, memory consumption grows linearly with the configuration value, which matches Figure 1.3 in Table 2.1.

Control Dependency (IF): Figures 5.2-5.5 show 4 different patterns that match 2.4-5.4 in Table 2.1. (1) In HDFS, maxFsObj is compared with a constant 0 to decide whether to skip the object-number checking along with the directory lock when writing new objects; Figure 5.2 shows the amount of time taken by the enclosing function checkFsObjectLimit() under different configuration settings. (2) Figure 5.3 shows that latency increases after memory usage exceeds the value of configuration parameter buffer size, as key-value pairs are written into on-disk structure FileCache (higher latency) instead of in-memory structure MemCache. (3) Write buffer size continues to accumulate before it hits configuration bufferSize and is flushed, causing the memory consumption to go up and down in Figure 5.4. (4) For requests with sizes larger than commitLogSize, the commit log is not written and thus has a lower latency. Sending requests with random size could have random latency as in Figure 5.5. Note that, in this particular case we changed commitlog append from asynchronous to synchronous for illustration purpose.

Control Dependency (LOOP): Figure 5.6-5.9 show 4 different patterns matching Table 2.1 6.3-7.3, and 8.4-9.4. (1) PerfConf lowerLimit affects the worst latency through the lower bound in a loop (Figure 5.6). (2) splitSize affects the number of MapTasks a job is spit into by affecting the loop stride. As shown in Figure 5.7, given a certain job, latency increases when splitSize is too small. (3) Figure 5.8 shows that user requests are blocked in a synchronization loop when Memstore size is larger than blockingSize, which has higher latency. (4) msgInterval controls the interval between sending heartbeats. Figure 5.9 shows the cost of building the heartbeat message under two configuration settings.
<table>
<thead>
<tr>
<th></th>
<th>Tot. PerfConfs</th>
<th></th>
<th>UF</th>
<th>LR</th>
<th>RE</th>
<th>RP</th>
<th>MO</th>
</tr>
</thead>
<tbody>
<tr>
<td>MR</td>
<td>129</td>
<td>33</td>
<td>16</td>
<td>18</td>
<td>7</td>
<td>11</td>
<td>31</td>
</tr>
<tr>
<td>HB</td>
<td>98</td>
<td>26</td>
<td>19</td>
<td>11</td>
<td>8</td>
<td>13</td>
<td>26</td>
</tr>
<tr>
<td>HD</td>
<td>155</td>
<td>32</td>
<td>13</td>
<td>16</td>
<td>12</td>
<td>9</td>
<td>27</td>
</tr>
<tr>
<td>CA</td>
<td>95</td>
<td>27</td>
<td>21</td>
<td>14</td>
<td>6</td>
<td>5</td>
<td>23</td>
</tr>
<tr>
<td>Total</td>
<td>477</td>
<td>118</td>
<td>69</td>
<td>59</td>
<td>33</td>
<td>38</td>
<td>107</td>
</tr>
</tbody>
</table>

Table 5.4: All PerfConf performance properties. **UF**: User-facing; **LR**: linear relation; **RE**: range effect; **RP**: having related PerfConfs; **MO**: monotonic.

![Figure 5.10: Linear relation](image)

### 5.3 Performance Property Inference

Overall, *LearnConf* identifies 118 out of 477 configurations in these 4 systems to have performance impact. Table 5.4 breaks down these 118 PerfConfs based on different performance properties identified by *LearnConf*. We also list all the user-facing PerfConfs and their detailed properties identified by *LearnConf* from HBase in Table 5.5, which we will refer to as examples below.

![Figure 5.11: Range effect](image)
## Table 5.5: All user-facing PerfConfs in HBase.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>S1</th>
<th>S2</th>
<th>S3</th>
<th>LR</th>
<th>RE</th>
<th>RP</th>
<th>MO</th>
</tr>
</thead>
<tbody>
<tr>
<td>multiplier</td>
<td>✔</td>
<td></td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>bloom.size</td>
<td></td>
<td>✔</td>
<td>✔</td>
<td></td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>max.filesize</td>
<td></td>
<td></td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>write.buffer</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>indexBlk.max</td>
<td></td>
<td>✔</td>
<td>✔</td>
<td></td>
<td>✔</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>cacheonwrite</td>
<td></td>
<td></td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>wakefrequency</td>
<td></td>
<td></td>
<td></td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>scanner.caching</td>
<td></td>
<td>✔</td>
<td>✔</td>
<td></td>
<td>✔</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>retries.number</td>
<td></td>
<td></td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>block.cache.size</td>
<td></td>
<td>✔</td>
<td>✔</td>
<td></td>
<td>✔</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>flush.size</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pause</td>
<td></td>
<td></td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>upperLimit</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lowerLimit</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>blockingSFs</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>compactThres</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hfile.format</td>
<td></td>
<td></td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>preclose.size#</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td>✔</td>
<td>✔</td>
<td></td>
</tr>
</tbody>
</table>

**Legend:**
- S1-3: Source 1-3;
- LR: linear relation;
- RE: range effect;
- RP: having related PerfConfs;
- MO: monotonic;
- #: false positive of LearnConf.

### Figure 5.12: Related configurations
**Input analysis** As shown in Table 5.4, 40% to 75% of all the PerfConfs are identified as using-facing. For these PerfConfs, *LearnConf* also identifies which specific user request a PerfConf affects. For instance, in HBase (Table 5.5), *LearnConf* identifies that `block.cache.size` and `wakefrequency` affect performance during user `get()` requests, while five other PerfConfs take effect during `createTable()` requests. In contrast, `balancerPeriod`, which determines the load-balancing period in HMaster, does not correspond to any user-request function and is not identified as user-facing.

**Slope analysis** Table 5.4 shows that 42% to 52% of PerfConfs have linear effects on performance metrics. Looking into the 8 linear HBase PerfConfs (Table 5.5, two PerfConfs have constant slope: `handler.count` affects the thread number linearly with a slope 1 and `bloom.size` affects the memory consumption linearly with a slope 1. The other 6 linear-effect PerfConfs' slope expressions each contains at least one non-constant program variable. We also run experiments for PerfConf `bloom.size`, which controls the size of a bloom fil-
ter. We profile the memory usage under different configurations and use linear regression to estimate the slope. As shown in Fig. 5.10, the slope is indeed close to 1.

Configuration setting range analysis Table 5.4 shows that 21% to 37% of PerfConfs have a range effects. Looking closely at the 6 range-effect PerfConfs in HBase (Table 5.5), four of them are used in the ”Compare with constant” pattern. The other two are reassigned with a boundary value when the PerfConf is outside the valid range: lowerLimit has a range of $(-\infty, upperLimit]$; and indexBlk.max has a range of $(0, \infty)$. We also verify this through experiments, where we fix the upperLimit to be 0.4, and we increase the lowerLimit from 0.05 to 0.8 with a step size of 0.05. Our experiment result shows that the worst latency drops while lowerLimit increases, and the worst latency is stable when lowerLimit is over upperLimit 0.4 as shown in Figure 5.11. This property is not documented in the default configuration file, yet LearnConf identifies these effects through static analysis.

Configuration-relation analysis Table 5.4 shows that 19% to 50% of PerfConfs are related to at least one other PerfConf (i.e., a PerfConf can affect the same PerfOp with another PerfConf in the same run). Looking into the 12 HBase PerfConfs that have related PerfConfs, two can enable other PerfConfs: lowerLimit only takes effect when the current Memstore size is larger than upperLimit; bloom.size takes effect when hfile.format is larger than the minimum supported format. The other 8 PerfConfs cannot enable other PerfConfs but can take effect with other PerfConfs simultaneously to affect performance. For example, HRegionServer blocks user writes when one Memstore size is larger than (flushSize * multiplier) — an increase in either flushSize configuration or multiplier configuration can lead to less frequent blocking, but longer blocking time. We experimentally confirm this relationship between the number of blocked writes and these two PerfConfs in Figure 5.12.

Monotonicity analysis As shown in Table 5.4, most PerfConfs have a monotonic relationship with performance. Table 5.5 shows that all user-facing PerfConfs are monotonic in HBase with 12 having a positive relationship and 7 having a negative one. As a rare
non-monotonic example, in HDFS, PerfConf maxFsObject is used in "Compared with constant" pattern. Namenode in HDFS skips the directory lock when maxFsObject is 0, and we experimentally show that both negative and positive values have a higher latency than zero (Figure 5.13). There are also a few cases where LearnConf finds that each PerfConf affects more than one PerfOp in the same program path with some being positive relationships and some being negative, and, thus, LearnConf cannot draw a monotonicity conclusion.

**Integration with existing auto tuners** LearnConf demystifies the complicated relationship between configurations and performance and can help existing auto tuners, like SmartConf [30], BestConfig [45] and others. We evaluate LearnConf using one SmartConf benchmark, HBase3813 [12]. The corresponding configuration is the max.queue.size, which determines the maximum queue size. SmartConf uses statistical profiling to determine the relationship between the queue size and memory usage, and then uses this profile to dynamically tune the queue size to prevent out of memory (OOM). However, SmartConf’s approach can still lead to OOM in 5 seconds (yellow line in Figure 5.14) when the offline profiling workload (0.1MB request size) is significantly different from the online workload (2MB request size). In contrast, LearnConf statically identifies that the queue size is further effected by the size of the objects in the queue—i.e., the size of user requests—and this additional information can easily be passed to SmartConf’s runtime to help correct the profiling problem, keeping the memory usage below the limits (blue line in Figure 5.14).

## 5.4 Analysis Time

As shown in Table 5.6, LearnConf takes 139 seconds to 355 seconds to analyze these four systems. 30% to 61% of this time is spent on WALA to build call graphs and program dependency graphs; 6% to 53% of time is spent on Taint Analysis and Dependency Analysis for configurations; the Other time is spent on additional analysis like Input Analysis.
<table>
<thead>
<tr>
<th>LOC</th>
<th>#Conf</th>
<th>Running Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>WALA</td>
</tr>
<tr>
<td>MR</td>
<td>137K</td>
<td>129</td>
</tr>
<tr>
<td>HB</td>
<td>267K</td>
<td>98</td>
</tr>
<tr>
<td>HD</td>
<td>149K</td>
<td>155</td>
</tr>
<tr>
<td>CA</td>
<td>151K</td>
<td>95</td>
</tr>
</tbody>
</table>

Table 5.6: Analysis Time
Related Work

**Misconfigurations** Previous work has conducted empirical studies related to misconfigurations [36, 40]. Previous works also applied statistical analysis [29, 31, 42, 43, 8], static program analysis [37, 26] and dynamic program analysis [4, 44] to detect and diagnose misconfigurations, but they did not focus on PerfConf. X-ray [3] helps diagnose performance problems by dynamic information flow tracking and performance summarization. Given a performance problem already triggered, X-ray effectively attributes run-time costs to different configurations, but is not designed to statically analyze configurations’ performance properties.

**Configuration auto-tuning** Many configuration auto-tuning approaches have been proposed for improving system performance [45, 41, 22, 14]. Those works mainly rely on a huge amount of training data to learn the complicated relation between configuration and performance. For example, the data collecting overhead in DAC is from 53 hours up to 92 hours under different workloads [41]. Recently, SmartConf, simplified the profiling process by assuming simple linear models between configuration and performance [30]. However, all those works still require re-training or re-profiling when workloads change significantly and they are susceptible to noise during data collection.

*LearnConf* is fundamentally different from these works; it explores all the configurations in the software without relying on particular workloads or training data and identifies its impacts based performance impact patterns through static analysis. *LearnConf* and dynamic training techniques can complement each other. On the one hand, knowing the relation between configuration and performance statically, *LearnConf* can be used to eliminate some noise in the dynamic training techniques. On the other hand, dynamic training techniques can derive more accurate knowledge about non-constant program variables for *LearnConf*.

**Performance bugs** Static analysis has been widely applied to find bugs, even performance
bug, such as inefficient loop [28], performance cascading [20], redundant traversal bugs [25], and other performance anti-patterns [15, 24, 39]. These works demonstrate that static analysis is useful to capture inefficient code. However, none of these works focused on performance-related configurations and their performance properties.
CHAPTER 7
CONCLUSION

Large software systems are often equipped with a huge number of configurations, and many of them have significant impacts on performance. Unfortunately, many PerfConfs are badly documented and hard to understand. We summarize 9 PerfConf performance patterns in taxonomy, and implement a static analysis tool LearnConf to capture PerfConf patterns. Our evaluation shows that LearnConf can correctly identify PerfConfs, capture performance patterns, and infer corresponding performance properties with lower false positive/negative. Its results are useful for both end users and existing auto-configuration framework.
REFERENCES


[17] Evangelia Kalyvianaki, Themistoklis Charalambous, and Steven Hand. 2009. Self-
adaptive and self-configured CPU resource provisioning for virtualized servers using Kalman filters. In *ICAC*.


