

Statically Inferring Performance Properties of Software Configurations

Chi Li

University of Chicago
lichi@uchicago.edu

Henry Hoffmann

University of Chicago
hankhoffmann@cs.uchicago.edu

Shu Wang

University of Chicago
shuwang@uchicago.edu

Shan Lu

University of Chicago
shanlu@uchicago.edu

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 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, which considers hundreds of configurations in four widely used distributed systems, demonstrates that *LearnConf* can accurately and efficiently identify many configurations' performance properties, and help performance tuning.

CCS Concepts • Software and its engineering → Software configuration management and version control systems; Automated static analysis; Cloud computing.

Keywords Software Configuration; Static Analysis; Performance; Distributed Systems

ACM Reference Format:

Chi Li, Shu Wang, Henry Hoffmann, and Shan Lu. 2020. Statically Inferring Performance Properties of Software Configurations. In *Fifteenth European Conference on Computer Systems (EuroSys '20), April 27–30, 2020, Heraklion, Greece*. ACM, New York, NY, USA, 16 pages. <https://doi.org/10.1145/3342195.3387520>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
EuroSys '20, April 27–30, 2020, Heraklion, Greece

© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-6882-7/20/04...\$15.00
<https://doi.org/10.1145/3342195.3387520>

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 [2]. 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 [10, 15]. Clearly, users and administrators need new tools to help understand how to configure these complicated software systems.

Figure 1 illustrates the challenge of understanding performance-related configurations through an example from HDFS, a widely-used distributed file system. The documentation shown in Figure 1a describes a configuration parameter, `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 and 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 1b) 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. Consequently, if the user/administrator changes the configuration from non-0 to 0, the write latency will 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

```
<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>
```

(a) Configuration file hdfs-default.xml

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

(b) Performance related code using the configuration

Figure 1. Documentation and code for a configuration parameter in HDFS.

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 effects 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 [7, 38] 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 [3, 26, 37, 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 [41, 45] and control theoretic techniques [16, 17, 22, 30] 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 [8, 12]. 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.

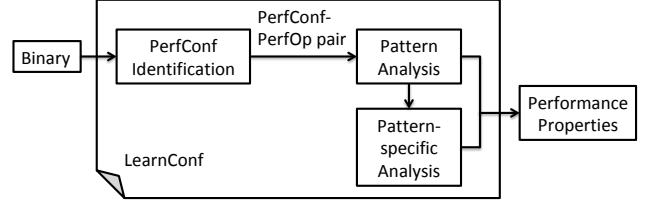


Figure 2. *LearnConf* Overview

Overall, there is a need for new 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, many performance impacts must be 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 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 in the taxonomy, we can figure out many detailed performance properties about this configuration. These patterns are presented in Section 2.

Guided by our taxonomy, we design *LearnConf*, a static analysis tool that automatically identifies configurations that have performance impacts, referred to as *PerfCons*, and infers the detailed properties of a *PerfConf*. *LearnConf* works in three steps, as shown in Figure 2.

First, *LearnConf* uses static data and control flow analysis to automatically identify configurations (*PerfCons*) 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 [1, 4–6, 18, 20, 27, 33–35, 45]. 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.

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 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 settings are also shown in the table, and will be elaborated below.

¹We define a PerfOp as an instruction or an API call that is particularly time-intensive (e.g., sleep, lock, I/O) or memory-intensive (e.g., new, malloc). We detail what APIs are treated as PerfOps by *LearnConf* in Section 3.1.2.

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 sleep, new, malloc, etc.), the PerfConf also has a simple, often linear, relationship with the corresponding performance metric, as shown in Table 1 Figure 1.3.

2.2 IF-related Control Dependency

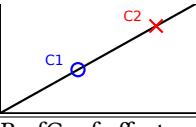
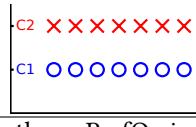
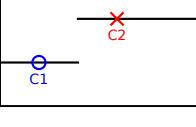
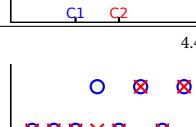
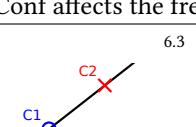
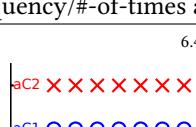
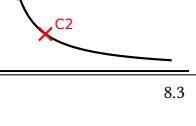
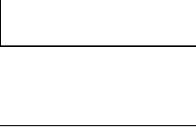
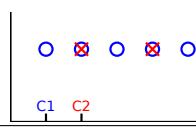
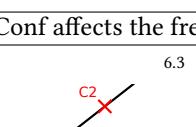
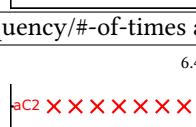
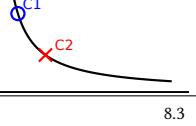
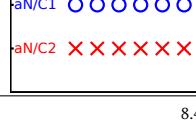
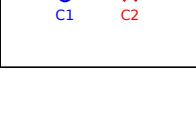
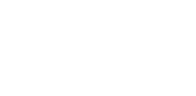
In this category, a variable C derived from a PerfConf is used in an if-condition predicate, whose evaluation picks from code paths containing different PerfOps. Consequently, the PerfConf setting affects which PerfOp is executed. Suppose the PerfOps on two code paths have performance contributions a 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 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.

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 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.

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 depends on V ’s current value in Table 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 1.

Table 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 in the 4th column depicts how the Perf (the y-axis) might change with the configuration setting (the x-axis); the *Perf–Time* figure in the 5th column depicts how the Perf (the y-axis) might change over time (the x-axis) under different configuration settings. In both columns' figures, we use C1 (blue dots) and C2 (red crosses) to represent two different configuration settings, and $C_1 < C_2$.

Pattern	Toy Example	Formula	Perf–Conf	Perf–Time	Explanation
Data Dependency – PerfConf affects the amount of performance contribution from one PerfOp					
1 Data	1.1 <code>new byte[C];</code>	1.2 $\text{Perf} = C$	1.3 	1.4 	C affects the impact of one <code>new byte[]</code> linearly.
Control Dependency (IF) – PerfConf affects whether a PerfOp is executed					
2 w/ Con- stant	2.1 <code>if(V<=C) new byte[a]; else new byte[b];</code>	2.2 $V \leq C$	2.3 	2.4 	C affects whether execute <code>new byte[a]</code> or <code>new byte[b]</code> statically.
3 w/ getting- closer- var	3.1 <code>if (V<=C) { V++; new byte[a]; } else new byte[b];</code>	3.2 $V \leq C$	3.3 	3.4 	C affects when to switch from <code>new byte[a]</code> to <code>new byte[b]</code> .
4 w/ back& forth var.	4.1 <code>if (V<=C) { V++; new byte[a]; } else { V--; new byte[b]; }</code>	4.2 $V \leq C$	4.3 	4.4 	C affect how quickly switch from <code>new byte[a]</code> to <code>new byte[b]</code> .
5 w/ un- related var.	5.1 <code>if(V<=C) new byte[a]; else new byte[b];</code>	5.2 $V \leq C$	5.3 	5.4 	C affects the probability of <code>new byte[a]</code> executes over <code>new byte[b]</code> .
Control Dependency (LOOP) – PerfConf affects the frequency/#-of-times a PerfOp executes					
6 Regular loop bound	6.1 <code>for(; i<C; i++) { new byte[a]; }</code>	6.2 $\text{Perf} = a * C$	6.3 	6.4 	C affects the number of <code>new byte[a]</code> via loop bound.
7 Regular loop stride	7.1 <code>for(; i<N; i+=C) { new byte[a]; }</code>	7.2 $\text{Perf} = a * N / C$	7.3 	7.4 	C affects the number of <code>new byte[a]</code> via loop stride.
8 Sync loop	8.1 <code>while (i<C) { wait(); } new byte[a];</code>	8.2 $\text{Perf} = a$	8.3 	8.4 	C affects the frequency of <code>new byte[a]</code> via synchronization loop.
9 Infinite loop	9.1 <code>new byte[a]; sleep(C); //in infinite loop</code>	9.2 $\text{Perf} = a$	9.3 	9.4 	C affects the frequency of <code>new byte[a]</code> via a fixed interval.

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

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 1).

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

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 Figure 5.4 in Table 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.

Affecting loop bound C is used in the bound expression of a PerfOp-containing loop. Figure 6.1 from Table 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.

Affecting loop stride C is used to set the loop stride, as shown in Figure 7.1 of Table 1. Here, a higher PerfConf value reduces the executions of a corresponding PerfOp as shown in Figure 7.3 in the table.

Affecting synchronization-loop Sometimes, C is the bound of a synchronization loop (e.g., a spin lock in Figure 8.1 of Table 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.

Affecting infinite-loop As illustrated in the toy example in Figure 9.1 of Table 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 PerfConf setting here decides the frequency of PerfOps that are executed periodically throughout system lifetime, as shown in Figure 9.4 in the table.

2.4 Discussion

Our taxonomy explores how the setting of a single PerfConf may impact software performance by affecting the execution of a PerfOp in the software. Our taxonomy is based on general program dependency categories—data dependency and control dependency, and looks at the impact of a PerfOp in a general way—whether it executes, how often it executes, what is the impact of a single dynamic instance of it. Consequently, our taxonomy is expected to generally apply to different software systems in different programming languages. Of course, in different systems, the exact performance metrics and PerfOps of interests might be different, and the exact program analysis used to identify the dependency could be different.

Of course, our current taxonomy still has limitations in the following ways. First, for simplicity, it focuses on the relationship between one PerfConf and one PerfOp. In practice, it is possible that one PerfConf can affect multiple PerfOps along different program paths or even along the same program path. We will need to look at these multiple PerfConf–PerfOp pairs together to precisely understand how the setting of a PerfConf can affect software performance. We will discuss this issue in later sections. Second, our discussion above assumes the PerfConfs under study to have numerical types. Indeed, previous study showed that the majority of PerfConfs in real world are of numerical types [30]; these numeric configurations, in nature, have non-constant searching space, and hence are more difficult to configure and need more tool support. Having said that, although rare, it is possible for boolean or string typed configurations to affect performance (e.g., a configuration that decides whether to use caching or not). Although currently not the focus of our taxonomy, they can potentially be covered by our taxonomy by extending the control-dependency patterns—these boolean/string configurations can be used as part of an IF/LOOP predicate and hence affect the execution of PerfOps.

3 PerfConf and Pattern Identification

As shown in Figure 2, *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 dependency analysis inside the function holding C ; when C is used as the parameter or the return value of a function, *LearnConf* conducts similar analysis inside the callee or caller function².

Things get harder when C is a heap variable. In this case, 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

²All the data and control dependency checking conducted by *LearnConf*, unless specially explained, leverages the program dependency graph of WALA [32].

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.

The analysis to identify the first type of operations above is straightforward. To identify the second type, *LearnConf* first identifies every array-typed return or parameter variable v of any I/O read API call in the program (e.g., b in `InputStream::read(byte[] b, int off, int len)`). For each such v , *LearnConf* tracks its data-flow chains to see if v is copied into a heap object (e.g., $o.f = v$). If so, *LearnConf* considers any object that belongs to the same class as o as containing I/O content. Once *LearnConf* identifies that v is copied into a heap object o , *LearnConf* does not further check if the content of o gets copied to other heap objects. Finally, for every container-add operation `container.add(a)`, *LearnConf* checks the class type of a to see if this operation belongs to the second type described above.

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., none of the variables involved in the loop-exit condition is 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.

When *LearnConf* checks whether a variable v is updated inside a loop body or inside an if-else body for above patterns, *LearnConf* mainly relies on the program-dependence graph provided by WALA [32]. When v is a heap object, *LearnConf* applies type-based alias analysis and checks k levels of function calls (default k is 2) to see if v is updated inside the callee functions.

3.3 Discussion

When we design the static analysis in *LearnConf*, we intentionally trade some analysis accuracy for analysis efficiency and scalability. For example, *LearnConf* largely relies on the program-dependence graph in WALA [32], and only conducts limited inter-procedural analysis and type-based alias analysis. *LearnConf* assumes that all objects under the same class share the same properties regarding whether they contain configuration variables or I/O data. The rationale behind our design is that we want to make sure *LearnConf* can scale to analyzing large-scale software systems and also that configuration variables tend not to be involved in complicated data/control flow or alias references in practice. As we will see in our evaluation (Section 5), *LearnConf* does make wrong judgement about PerfConfs due to its static analysis inaccuracy, but only rarely.

4 Analysis Beyond Patterns

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 \mathbb{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 \mathbb{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 \mathbb{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 $f \in \mathbb{F}_u$, f' also belongs to \mathbb{F}_u ;

(2) if a function f' is an RPC function that can be invoked by an RPC-call in $f \in \mathbb{F}_u$, f' also belongs to \mathbb{F}_u ; (3) if a function $f \in \mathbb{F}_u$ 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 \mathbb{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., α) is useful for performance tuning and satisfying performance constraints. Prior work typically obtains this information from extensive profiling [8, 9, 30].

There are four patterns from Table 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,³ 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 `sor tmb * 1024 * 1024` for the example in Figure 3a and `timeout * Math.pow(2, attempts)` for the example in Figure 3b (*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 an expression with constants like $1024 * 1024$ for configuration `sor tmb` in Figure 3a or an expression with program variables like $2^{attempts}$ for configuration `timeout` in Figure 3b.

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 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*

³*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.

```
1 int maxUsage = sor tmb * 1024 * 1024;
2 buffer = new Byte[maxUsage];
```

(a) `iosortmb` has constant slope

```
1 while (!stopped) {
2     wait = timeout * Math.pow(2, attempts);
3     sleep(wait);
4 }
```

(b) `timeout` has non-constant slope

Figure 3. PerfConfs with different slope

will not produce a slope. Just like that in Section 3, *LearnConf* checks execution paths precisely for intra-procedural cases (i.e., when two pairs of PerfConf-PerfOp are inside the same function), but trades accuracy for efficiency in inter-procedural cases.

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 op const` predicate, symbolically replaces `C` with $f(\text{Conf})$ (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 identify 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 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 4a, 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 4b, `numRetries` and `sleepTime` determine the number of iterations and the time spent on each iteration to affect user latency.

```

1  if (memStoreSize >= upperLimit) {
2      for (; memStoreSize > lowerLimit;) {
3          flushRegion(); //lock and I/O
4      }
5  }

(a) upperLimit enables lowerLimit

1  for (; tries < numRetries; ++tries) {
2      Thread.sleep(sleepTime);
3  }

(b) numRetries and sleepTime work simultaneously

```

Figure 4. Related Configurations

In practice, users may want to know whether the setting of two related configurations could conflict with each other. *LearnConf*'s analysis above can help answer this question. When configuration C_2 can enable/disable the effect of configuration C_1 , their setting can potentially conflict: a change to C_1 that aims to increase its performance impact would fail if a change to C_2 disables the effect of C_1 (e.g., imagine one increases `upperLimit` to largely bypass the `for` loop in Figure 4-a, while `lowerLimit` is also changed.). On the other hand, when two configurations take effect simultaneously, they may conflict with each other if one is changed to increase a performance metric and the other is changed to decrease (e.g., imagine one increase `numRetries` and decrease `sleepTime` in Figure 4b).

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 1 – particularly the figures in column *PerfConf* – 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 identifies potential range effects as following. First, when the configuration variable C is used in an `if`-predicate, *LearnConf* checks whether an increase of C could lead to more PerfOp executions in two disjoint regions, like `if (C!=a) PerfOp();` or `if (C<a || C>b) PerfOp();`. Second, *LearnConf* checks the symbolic expression of $C = f(conf)$ and cautions users of possible non-monotonic relationship in following cases: (1) $f(conf)$ is fragmented like $f = (conf < A) ? f_1 : f_2$; (2) $f(conf)$ contains a non-monotonic mathematical function like `Math.power(*, 3)`.

Regarding one PerfConf affecting multiple PerfOps, for simplicity, assume that *LearnConf* finds two PerfOps of the same performance type (e.g., latency), $PerfOp_1$ and $PerfOp_2$, that both depend on the same PerfConf along the same program path. *LearnConf* then analyses the PerfConf- $PerfOp_1$

and PerfConf- $PerfOp_2$ relationships independently. If the result is inconsistent, *LearnConf* cannot declare the relationships monotonic. This situation can arise if increasing the PerfConf causes $PerfOp_1$ to execute more and $PerfOp_2$ to execute less, for example.

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 types of alternative sources: (1) configurations used by prior work on performance tuning, and (2) configurations mentioned in software performance-tuning guides/tutorials [1, 33], 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 [20] and many others where the respective authors manually select configurations that they believe might have performance impact and are hence worth tuning [4–6, 18, 27, 34, 35, 45]. Overall, we have three alternative sources for MapReduce ([18], [6], and [34]), three for Hbase ([4], [35], and [5]), three for HDFS ([33], [1], [34]); and three for Cassandra ([45], [20], and [27]). They are denoted as Source 1–3 for each application in Table 2.

When a configuration is identified as related to performance by both *LearnConf* and an alternative source, we consider it as a true PerfConf. When a configuration is identified as related to performance by only *LearnConf* or only alternative sources, we manually studied the source code and ran experiments to get the ground truth, many of which we will explain in details below.

Overall results As shown in Table 2, *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;

	<i>LearnConf</i>			Source 1			Source 2			Source 3		
	All	F+	F-	All	F+	F-	All	F+	F-	All	F+	F-
MapReduce (MR)	16	1	7	14	0	8	16	0	6	10	0	12
HBase (HB)	19	1	2	7	0	13	12	1	9	9	0	11
HDFS (HD)	13	5	1	2	0	7	2	0	7	1	0	8
Cassandra (CA)	21	2	1	24	8	4	10	0	10	6	1	15
Total	69	9	11	-	-	-	-	-	-	-	-	-

Table 2. User-facing PerfConfs reported by *LearnConf* and those used by other sources. The alternative sources 1–3 are explained in details in Section 5.1. Since these sources for different applications are different, their numbers are not summed together. F+: false positives; F-: false negatives.

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 [18] or bulk loading latency [4]. 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 [20]. (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

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

Table 3. PerfConf-PerfOp pattern distribution (Pk indicates the k-th pattern listed in Table 1, like P1 indicating the “Data” pattern in Table 1).

	Tot.	Correct	False Pattern	False Taint
MR	94	90	4	0
HB	112	96	16	0
HD	90	85	0	5
CA	109	107	2	0
Total	405	378	22	5

Table 4. PerfConf-PerfOp pattern identification accuracy.

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 reassigning 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 3 summarizes the pattern distribution of these 405 PerfConf-PerfOp pairs. In this table, P1 ... P9 indicate the first (i.e.,

“Data”) to the ninth (i.e., “Infinite loop”) patterns listed in Table 1. We manually examine the source code to judge whether the inferred pattern is correct. As shown in Table 4, *LearnConf* correctly identifies the patterns for 378 out of 405 pairs.

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: all match the pattern-specific behaviors discussed in Section 2.

Data dependency: PerfConf `sortmb` is used as a parameter to create a byte array, as shown in Figure 5a, memory consumption grows linearly with the configuration value, which matches Figure 1.3 in Table 1.

Control Dependency (IF): Figures 5b-5e show 4 different patterns that match 2.4-5.4 in Table 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 5b shows the amount of time taken by the enclosing function `checkFsObjectLimit()` under different configuration settings. (2) Figure 5c 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 5d. (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 5e. Note that, in this particular case we changed `commitlog.append` from asynchronous to synchronous for illustration purpose.

Control Dependency (LOOP): Figure 5f-5i show 4 different patterns matching Table 1 Figure 6.3, 7.3, 8.4 and 9.4. (1) PerfConf `lowerLimit` affects the worst latency through the lower bound in a loop (Figure 5f). (2) `splitSize` affects the number of MapTasks a job is split into by affecting the loop stride. As shown in Figure 5g, given a certain job, latency increases when `splitSize` is too small. (3) Figure 5h shows that user requests are blocked in a synchronization loop when `Memstore.size` is larger than `blockingSize`, which

Conf.	Tot.	PerfConfs				
		UF	LR	RE	RP	MO
MR	129	33	16	18	7	11
HB	98	26	19	11	8	13
HD	155	32	13	16	12	9
CA	95	27	21	14	6	5
Total	477	118	69	59	33	38
						107

Table 5. All PerfConf performance properties. **UF:** User-facing; **LR:** linear relation; **RE:** range effect; **RP:** having related PerfConfs; **MO:** monotonic.

Configuration	Comparison			Properties			
	S1	S2	S3	LR	RE	RP	MO
multiplier	✓	✗	✓	✓	✗	✓	✗
bloom.size	✗	✓	✗	✓	✗	✓	✗
maxfilesize	✗	✗	✓	✓	✗	✗	✗
write.buffer	✓	✗	✗	✗	✗	✗	✗
indexBlk.max	✗	✓	✗	✗	✓	✗	✗
cacheonwrite	✗	✓	✗	✗	✓	✗	✗
wakefrequency	✗	✗	✗	✗	✓	✓	✗
scanner.caching	✗	✗	✗	✓	✗	✓	✗
retries.number	✗	✗	✗	✓	✗	✓	✗
block.cache.size	✗	✓	✓	✗	✓	✗	✗
flush.size	✓	✓	✓	✗	✗	✓	✗
pause	✗	✗	✗	✓	✗	✓	✗
upperLimit	✓	✓	✓	✗	✗	✓	✗
lowerLimit	✓	✓	✓	✓	✓	✓	✗
blockingSFs	✓	✓	✓	✗	✗	✓	✗
compactThres	✓	✓	✓	✗	✗	✓	✗
hanlder.count	✗	✓	✓	✓	✗	✗	✗
hfile.format	✗	✗	✗	✗	✓	✓	✗
preclose.size#	-	-	-	✗	✗	✗	✗

Table 6. 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*.

has higher latency. (4) `msgInterval` controls the interval between sending heartbeats. Figure 5i shows the cost of building the heartbeat message under two configuration settings.

5.3 Performance Property Inference

Overall, *LearnConf* identifies 118 out of 477 configurations in these 4 systems to have performance impact. Table 5 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 6, which we will refer to as examples below.

Input analysis As shown in Table 5, 40% to 75% of all the PerfConfs are identified as user-facing. For these PerfConfs,

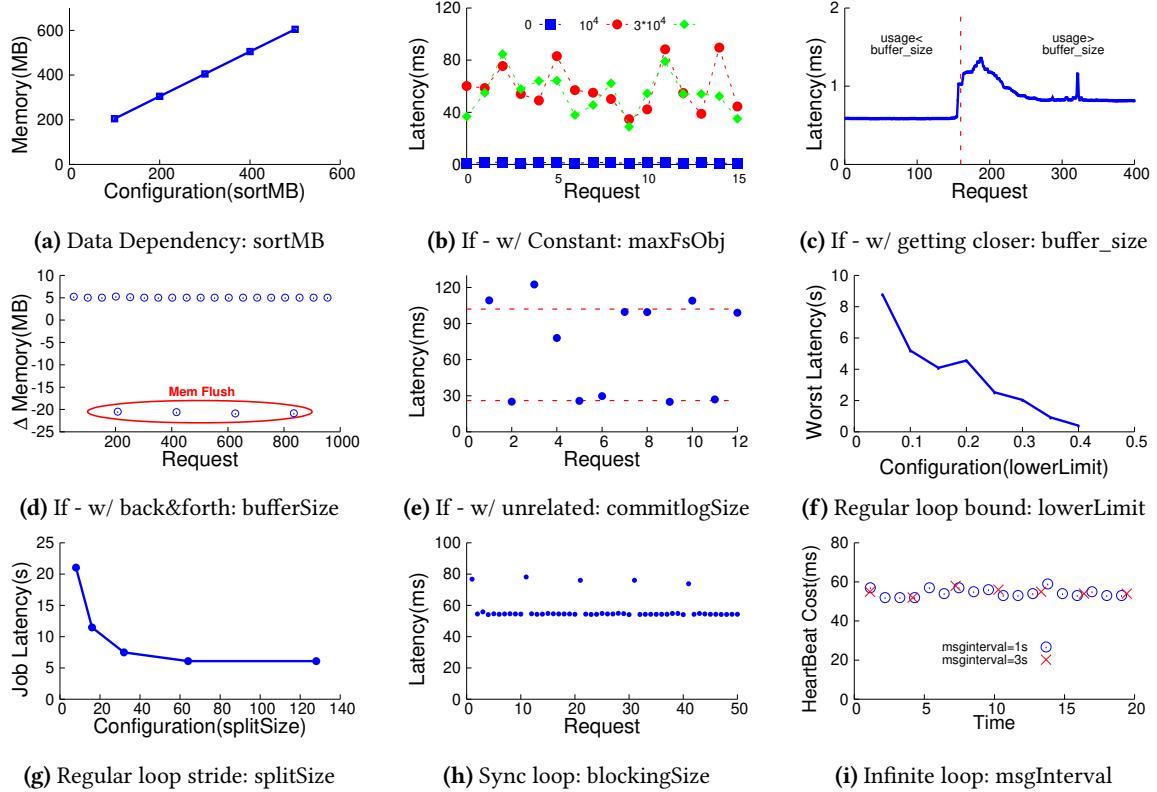


Figure 5. Experimental result for 9 patterns. (For sub-figure 5b, 5c, 5d, 5e and 5h, we keep sending requests and measure latency or memory consumption after every request.)

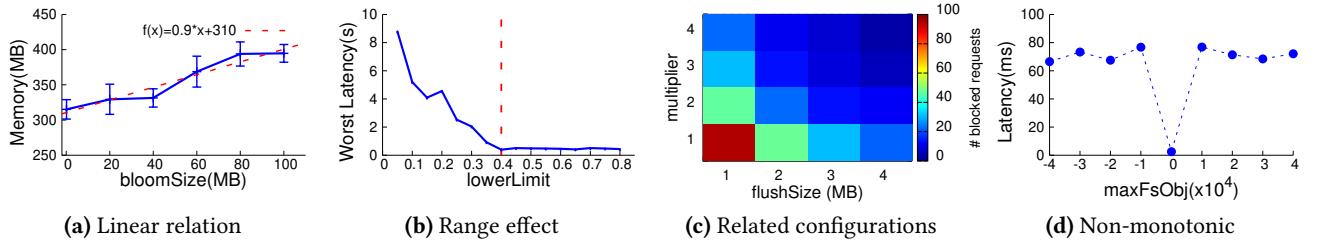


Figure 6. Evaluation for performance property inference

LearnConf also identifies which specific user request a PerfConf affects. For instance, in HBase (Table 6), *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 shows that 42% to 52% of PerfConfs have linear effects on performance metrics. Looking into the 8 linear HBase PerfConfs (Table 6), 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 filter. We profile the memory usage under different configurations and use linear regression to estimate the slope. As shown in Figure 6a), the slope is indeed close to 1.

Configuration setting range analysis Table 5 shows that 21% to 37% of PerfConfs have a range effects. Looking closely at the 6 range-effect PerfConfs in HBase (Table 6), 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 $(-\infty, \text{upperLimit}]$; 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 6b. This property is not documented in the default configuration file, yet *LearnConf* identifies these effects through static analysis.

Configuration-relation analysis Table 5 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). For example, there are 12 HBase PerfConfs that have related PerfConfs. In some cases, one PerfConf's setting can enable or disable the effect of the other PerfConf: `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. In other cases, a group of PerfConfs take effect simultaneously to affect the same performance metric. For example, the HRegionServer in HBase blocks user writes when one Memstore size is larger than `(flushSize * multiplier)`. The two configurations `flushsize` and `multiplier` are hence related—an increase in either of them leads to less frequent blocking. We experimentally confirmed this relationship in Figure 6c, where the x-axis and y-axis represent the setting of these two configurations and the red-to-blue colors indicate high-to-low blocking frequency.

Monotonicity analysis As shown in Table 5, most PerfConfs have a monotonic relationship with performance. Table 6 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 6d). 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.

5.4 Analysis Time

As shown in Table 7, *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.

LOC	#Conf	Running Time			
		WALA	Taint	Others	Total
MR	137K	129	76s	28s	34s 139s
HB	267K	98	216s	21s	118s 355s
HD	149K	155	57s	101s	33s 192s
CA	151K	95	67s	78s	11s 156s

Table 7. Analysis Time

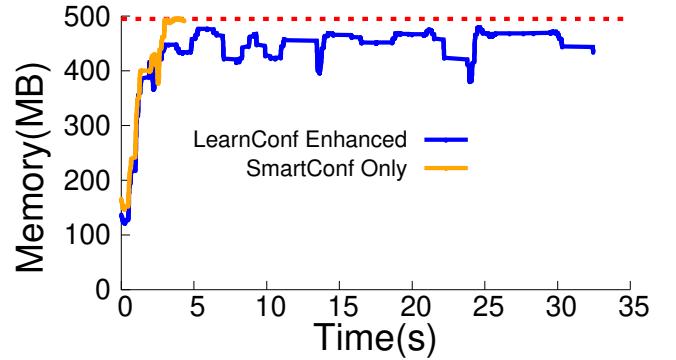


Figure 7. Applying *LearnConf* to enhance SmartConf

5.5 Applying *LearnConf* for Performance Tuning

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 [11]. 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 7) 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 7).

We also expect *LearnConf* to help manual performance tuning, informing users which configurations can affect which performance metrics in which way. For example, in one Stack Overflow post, a user asked about how to solve an out of memory problem he encountered in HBase [25]. After one hour's debugging with help, the user finally realized that the out of memory problem was caused by accidentally setting the configuration `scanner.caching` to `Integer.MAX_VALUE`. If the user used *LearnConf*, he would quickly know that among hundreds of configurations in

HBase, 6 of them can greatly affect memory consumption, including `scanner.caching`. Future work can conduct user study to quantitatively assess how much *LearnConf* can help manual performance tuning.

5.6 Limitations of *LearnConf*

The evaluation above shows the potential of *LearnConf* and we believe that *LearnConf* is just a starting point in using static analysis to help understand performance properties of software configurations. The current prototype of *LearnConf* still has the following limitations. First, its PerfConf taxonomy currently does not handle non-numerical configurations and does not directly model how one PerfConf affects multiple PerfOps, as discussed in Section 2.4. Second, as a static analysis tool, *LearnConf* intentionally trades some accuracy for efficiency and scalability by conducting limited alias analysis and inter-procedural analysis, as discussed in Section 3.3. Third, some customization is needed while applying *LearnConf* to a new software. Particularly, we may need to customize the list of PerfOp APIs for the target software. Although some common performance intensive APIs like `sleep`, `new`, `malloc`, `lock` can be shared among software, it is possible that the target software has its own utility functions that will be helpful to annotate as PerfOps (e.g., heartbeat functions in many distributed systems). Finally, *LearnConf* only aims to help improve users or auto-tuning tools' understanding about performance related configurations. Automatically deciding the best configuration setting is out of the scope of *LearnConf*.

6 Related Work

Misconfigurations Previous work has conducted empirical studies related to misconfigurations [36, 40]. Previous works also applied statistical analysis [7, 29, 31, 42, 43], static program analysis [26, 37] and dynamic program analysis [3, 44] to detect and diagnose mis-configurations, but they did not focus on PerfConf. X-ray [2] 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 [13, 21, 41, 45]. 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 [19], redundant traversal bugs [24], and other performance anti-patterns [14, 23, 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.

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.

Acknowledgments

We would like to thank Ravi Chugh for his suggestions on this project, our shepherd and the anonymous reviewers for their insightful feedback and comments. This research is supported by NSF (grants CCF-1837120, CNS-1764039, CNS-1563956, CNS-1514256, IIS-1546543, CNS-1823032, CCF-1439156), ARO (grant W911NF1920321), DOE (grant DESC0014195 0003), DARPA (grant FA8750-16-2-0004) and the CERES Center for Unstoppable Computing. Henry Hoffmann's effort is additionally supported by the DARPA BRASS program and a DOE Early Career award.

References

- [1] Apache hadoop 2.9.2 - hdfs users guide. <https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsUserGuide.html>.
- [2] Mona Attariyan, Michael Chow, and Jason Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *Presented as part of the 10th USENIX Symposium on Operating Systems*

- Design and Implementation (OSDI 12)*, pages 307–320, Hollywood, CA, 2012. USENIX.
- [3] Mona Attariyan and Jason Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *OSDI*, volume 10, pages 1–14, 2010.
 - [4] Xianqiang Bao, Ling Liu, Nong Xiao, Fang Liu, Qi Zhang, and Tao Zhu. Hconfig: Resource adaptive fast bulk loading in hbase. In *10th IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing*, pages 215–224. IEEE, 2014.
 - [5] Xianqiang Bao, Ling Liu, Nong Xiao, Yang Zhou, and Qi Zhang. Policy-driven configuration management for nosql. In *2015 IEEE 8th International Conference on Cloud Computing*, pages 245–252. IEEE, 2015.
 - [6] Chi-Ou Chen, Ye-Qi Zhuo, Chao-Chun Yeh, Che-Min Lin, and Shih-Wei Liao. Machine learning-based configuration parameter tuning on hadoop system. In *2015 IEEE International Congress on Big Data*, pages 386–392. IEEE, 2015.
 - [7] Nick Feamster and Hari Balakrishnan. Detecting bgp configuration faults with static analysis. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 43–56. USENIX Association, 2005.
 - [8] Antonio Filieri, Henry Hoffmann, and Martina Maggio. Automated design of self-adaptive software with control-theoretical formal guarantees. In *Proceedings of the 36th International Conference on Software Engineering*, pages 299–310. ACM, 2014.
 - [9] Antonio Filieri, Henry Hoffmann, and Martina Maggio. Automated multi-objective control for self-adaptive software design. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 13–24. ACM, 2015.
 - [10] Google. Google cloud storage incident #19002. <https://status.cloud.google.com/incident/storage/19002>, 2019.
 - [11] HBase-3813. Change rpc callqueue size from handlercount * max_queue_size_per_handler. <https://issues.apache.org/jira/browse/HBASE-3813>.
 - [12] Joseph L. Hellerstein, Yixin Diao, Sujay Parekh, and Dawn M. Tilbury. *Feedback Control of Computing Systems*. John Wiley & Sons, 2004.
 - [13] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, and Shivnath Babu. Starfish: A self-tuning system for big data analytics. In *CIDR*, 2011.
 - [14] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. *ACM SIGPLAN Notices*, 47(6):77–88, 2012.
 - [15] Wall Street Journal. Facebook, google and apple hit by unusual outages. <https://www.wsj.com/articles/facebook-and-instagram-suffer-lengthy-outages-11552539752>, 2019.
 - [16] Evangelia Kalyvianaki, Themistoklis Charalambous, and Steven Hand. Self-adaptive and self-configured cpu resource provisioning for virtualized servers using kalman filters. In *ICAC*, 2009.
 - [17] Evangelia Kalyvianaki, Themistoklis Charalambous, and Steven Hand. Adaptive resource provisioning for virtualized servers using kalman filters. *TAAS*, 2014.
 - [18] Sandeep Kumar, Sindhu Padakandla, Priyank Parihar, K Gopinath, Shalabh Bhatnagar, et al. Performance tuning of hadoop mapreduce: A noisy gradient approach. *arXiv preprint arXiv:1611.10052*, 2016.
 - [19] Jiaxin Li, Yuxi Chen, Haopeng Liu, Shan Lu, Yiming Zhang, Haryadi S Gunawi, Xiaohui Gu, Xicheng Lu, and Dongsheng Li. Pcatch: automatically detecting performance cascading bugs in cloud systems. In *Proceedings of the Thirteenth EuroSys Conference*, page 7. ACM, 2018.
 - [20] Ashraf Mahgoub, Paul Wood, Sachandhan Ganesh, Subrata Mitra, Wolfgang Gerlach, Travis Harrison, Folker Meyer, Ananth Grama, Saurabh Bagchi, and Somali Chaterji. Rafiki: A middleware for parameter tuning of nosql datastores for dynamic metagenomics workloads. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, pages 28–40. ACM, 2017.
 - [21] Ashraf Mahgoub, Paul Wood, Alexander Medoff, Subrata Mitra, Folker Meyer, Somali Chaterji, and Saurabh Bagchi. SOPHIA: Online reconfiguration of clustered nosql databases for time-varying workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 223–240, Renton, WA, July 2019. USENIX Association.
 - [22] Nikita Mishra, Connor Imes, John D Lafferty, and Henry Hoffmann. Caloree: Learning control for predictable latency and low energy. *ACM SIGPLAN Notices*, 53(2):184–198, 2018.
 - [23] Adrian Nistor, Po-Chun Chang, Cosmin Radoi, and Shan Lu. Caramel: Detecting and fixing performance problems that have non-intrusive fixes. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 902–912. IEEE, 2015.
 - [24] Oswaldo Olivo, Isil Dillig, and Calvin Lin. Static detection of asymptotic performance bugs in collection traversals. In *ACM SIGPLAN Notices*, volume 50, pages 369–378. ACM, 2015.
 - [25] Stack Overflow. Hbase region server oom and shuts down. <https://stackoverflow.com/questions/31239887/hbase-region-server-oom-and-shuts-down>.
 - [26] Ariel Rabkin and Randy Katz. Precomputing possible configuration error diagnoses. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 193–202. IEEE Computer Society, 2011.
 - [27] Katam Sathvik. Performance tuning of big data platform: Cassandra case study, 2016.
 - [28] Linhai Song and Shan Lu. Performance diagnosis for inefficient loops. In *Proceedings of the 39th International Conference on Software Engineering*, pages 370–380. IEEE Press, 2017.
 - [29] Helen J Wang, John C Platt, Yu Chen, Ruyun Zhang, and Yi-Min Wang. Automatic misconfiguration troubleshooting with peerpressure. In *OSDI*, volume 4, pages 245–257, 2004.
 - [30] Shu Wang, Chi Li, Henry Hoffmann, Shan Lu, William Sentosa, and Achmad Imam Kistijantoro. Understanding and auto-adjusting performance-sensitive configurations. In *ACM SIGPLAN Notices*, volume 53, pages 154–168. ACM, 2018.
 - [31] Yi-Min Wang, Chad Verbowski, John Dunagan, Yu Chen, Helen J Wang, Chun Yuan, and Zheng Zhang. Strider: A black-box, state-based approach to change and configuration management and support. *Science of Computer Programming*, 53(2):143–164, 2004.
 - [32] IBM Watson. Watson libraries for analysis. wala.sourceforge.net/wiki/index.php/Main_Page.
 - [33] Tom White. *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 2012.
 - [34] Dili Wu and Aniruddha Gokhale. A self-tuning system based on application profiling and performance analysis for optimizing hadoop mapreduce cluster configuration. In *20th Annual International Conference on High Performance Computing*, pages 89–98. IEEE, 2013.
 - [35] Wen Xiong, Zhengdong Bei, Chengzhong Xu, and Zhibin Yu. Ath: Auto-tuning hbase's configuration via ensemble learning. *IEEE Access*, 5:13157–13170, 2017.
 - [36] Tianyin Xu, Long Jin, Xuepeng Fan, Yuanyuan Zhou, Shankar Pasupathy, and Rukma Talwalkar. Hey, you have given me too many knobs!: understanding and dealing with over-designed configuration in system software. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 307–319. ACM, 2015.
 - [37] Tianyin Xu, Xinxin Jin, Peng Huang, Yuanyuan Zhou, Shan Lu, Long Jin, and Shankar Pasupathy. Early detection of configuration errors to reduce failure damage. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 619–634, 2016.
 - [38] Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. Do not blame users for misconfigurations. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 244–259. ACM, 2013.
 - [39] Junwen Yang, Cong Yan, Chengcheng Wan, Shan Lu, and Alvin Cheung. View-centric performance optimization for database-backed web applications. In *Proceedings of the 41st International Conference on*

- Software Engineering*, pages 994–1004. IEEE Press, 2019.
- [40] 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 *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 159–172. ACM, 2011.
 - [41] Zhibin Yu, Zhendong Bei, and Xuehai Qian. Datasize-aware high dimensional configurations auto-tuning of in-memory cluster computing. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 564–577, New York, NY, USA, 2018. ACM.
 - [42] Ding Yuan, Yinglian Xie, Rina Panigrahy, Junfeng Yang, Chad Verbowski, and Arunvijay Kumar. Context-based online configuration-error detection. In *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*, pages 28–28. USENIX Association, 2011.
 - [43] Jiaqi Zhang, Lakshminarayanan Renganarayana, Xiaolan Zhang, Niyu Ge, Vasanth Bala, Tianyin Xu, and Yuanyuan Zhou. Encore: Exploiting system environment and correlation information for misconfiguration detection. In *ACM SIGPLAN Notices*, volume 49, pages 687–700. ACM, 2014.
 - [44] Sai Zhang and Michael D Ernst. Automated diagnosis of software configuration errors. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 312–321. IEEE Press, 2013.
 - [45] Yuqing Zhu, Jianxun Liu, Mengying Guo, Yungang Bao, Wenlong Ma, Zhuoyue Liu, Kumpeng Song, and Yingchun Yang. Bestconfig: tapping the performance potential of systems via automatic configuration tuning. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 338–350. ACM, 2017.