Understanding and Auto-Adjusting Performance-Sensitive Configurations

Shu Wang Chi Li
Henry Hoffmann Shan Lu
University of Chicago
Chicago, Illinois
{shuwang,lichi,hankhoffmann,shanlu}@cs.uchicago.edu

William Sentosa
Achmad Imam Kistijantoro
Bandung Institute of Technology & Surya University
Bandung, Indonesia
williamsentosa@students.itb.ac.id;imam@stei.itb.ac.id

Abstract

Modern software systems are often equipped with hundreds to thousands of configurations, many of which greatly affect performance. Unfortunately, properly setting these configurations is challenging for developers due to the complex and dynamic nature of system workload and environment. In this paper, we first conduct an empirical study to understand performance-sensitive configurations and the challenges of setting them in the real-world. Guided by our study, we design a systematic and general control-theoretic framework, *SmartConf*, to automatically set and dynamically adjust performance-sensitive configurations to meet required operating constraints while optimizing other performance metrics. Evaluation shows that *SmartConf* is effective in solving real-world configuration problems, often providing better performance than even the best static configuration developers can choose under existing configuration systems.

CCS Concepts • **Software and its engineering** → **Software performance**; **Software configuration management and version control systems**; *Software reliability*; • **Computer systems organization** → *Cloud computing*;

Keywords Software Configuration; Performance; Distributed Systems; Control Theory

ACM Reference Format:
Shu Wang, Chi Li, William Sentosa, Henry Hoffmann, Shan Lu, and Achmad Imam Kistijantoro. 2018. Understanding and Auto-Adjusting Performance-Sensitive Configurations. In *Proceedings of 2018 Architectural Support for Programming Languages and Operating Systems (ASPLOS'18).* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3173162.3173206

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 ACM 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.
ASPLOS'18, March 24–28, 2018, Williamsburg, VA, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-4911-6/18/03…$15.00
https://doi.org/10.1145/3173162.3173206

1 Introduction

"all constants should be configurable, even if we can't see any reason to configure them." — HDFS-4304

1.1 Motivation

Modern software systems are equipped with hundreds to thousands of configuration options allowing customization to different workloads and hardware platforms. While these configurations provide great flexibility, they also put great burdens on users and developers, who are now responsible for setting them to ensure the software is performant and available. Unfortunately, this burden is more than most users can handle, making software *misconfiguration* one of the biggest causes of system misbehavior [16, 17, 59]. Misconfiguration leads to both incorrect functionality (e.g., wrong outputs, crashes) and poor performance. Although recent research has tackled functionality issues arising from misconfiguration [58, 59], poor performance is an open problem.

In server applications, customizable configuration parameters are especially common. These configurations control the size of critical data structures, the frequency of performance-sensitive operations, the thresholds and weights in workload-dependent algorithms, and many other aspects of system operation. A previous study found that 20% of user-reported misconfiguration problems in MySQL database systems result in severe performance degradation, and yet, performance-related misconfigurations are under-reported [61]. Other studies found that a majority of configuration issues in Hadoop systems were related to system performance, caused by poorly tuned configuration parameters [26], and about a third of the misconfiguration problems result in memory-related performance issues, particularly `OutOfMemoryErrors` [43].

Setting performance-sensitive configurations, *PerfConfs* for short, is challenging because they represent tradeoffs; e.g., between memory usage and response time. Managing these tradeoffs requires deep knowledge of the underlying hardware, the workload, and the PerfConf itself. Often, these relationships are not, or cannot, be clearly explained in the documentation [18]. Even with clear documentation, the workload and system interaction are often too *complicated* or *change* too quickly for users to maintain a proper setting

[43]. In many cases, there is simply no satisfactory static setting of a PerfConf [11].

Configuring software to an optimal point in a tradeoff space is a constrained optimization problem. Operating requirements represent constraints, and the goal is finding the optimal PerfConf setting given those constraints. For example, a larger queue makes a system more responsive to bursty requests at the cost of increased memory usage. Here the constraint is that the system not run out of memory and the goal is to minimize response time. Prior work addresses this problem in several ways, with no perfect solution.

The industry standard is simply to expose parameters to users who are forced to become both application and system experts to understand the best settings for their particular system and workload, as shown in Section 2.

Control theoretic frameworks handle constrained optimization for non-functional software properties [14]. Typical control solutions, however, require a deep understanding of a specific system [25, 34, 35, 49, 64], and hence, are impractical for real-world developers to adopt. Even general control synthesis techniques [12] still require user-specified parameters. More importantly, they cannot handle challenges unique to PerfConfs, such as hard constraints—e.g., not going out of memory—and indirect relationships between PerfConfs and performance.

Machine learning techniques have been applied to explore complex configuration spaces to find near optimal settings without considering constraints on operating behavior [5, 53, 60, 68]. Some approaches employ ML to meet resource constraints in dynamic environments [9]. In general, however, machine learning techniques provide very limited formal guarantees that they will meet strict constraints—e.g., preventing out-of-memory-errors—in dynamic environments [51]. In contrast, control theory specifically addresses formal analysis of system dynamics [20]. Empirical studies of computer resource management confirm that control theoretic solutions do a better job of meeting constraints in practice [36].

1.2 Contributions

In this paper, we first conduct an empirical study to understand real-world performance-related configuration problems. The results motivate us to construct a general framework, *SmartConf*. Unlike traditional configuration frameworks —where users set PerfConfs at system launch—*SmartConf* automatically sets and dynamically adjusts PerfConfs. *SmartConf* decomposes the PerfConf setting problem to let users, developers, and control-theoretic techniques —which we specifically design for PerfConfs—each focus on what they know the best, as shown in Table 1.

Table 1. Traditional configuration vs *SmartConf*

Prior	Who answers these questions?	*SmartConf*
N/A	Which C needs dynamic adjustment?	Developers
N/A	What perf. metric M does C affect?	Developers
N/A	What is the constraint on metric M?	Users
Users	How to set & adjust configuration C?	*SmartConf*

Empirical study We look at 80 developer-patches and 54 user-posts concerning PerfConfs in 4 widely used large-scale systems. We find (1) PerfConfs are common among configuration-related patches (>50%) and forum questions (~30%); (2) almost half of PerfConf patches fix performance issues caused by improper default settings; (3) properly setting PerfConfs requires considering dynamic workload, environmental factors, and performance tradeoffs.

Our study also points out challenges in setting and adjusting PerfConfs: (1) about half of PerfConfs threaten *hard* performance constraints like out-of-memory or out-of-disk problems; (2) about half of PerfConfs affect performance *indirectly* through setting thresholds for other system variables; (3) more than half of PerfConfs are associated with specific system events and hence only take effect *conditionally*; and (4) often different configurations affect the same performance goal simultaneously, requiring *coordination*.

SmartConf interface Guided by this study, we design a new configuration interface. For developers, *SmartConf* encourages them to decide which PerfConf should be dynamically configured and enables them to easily convert a wide variety of PerfConfs from their traditional format—requiring developers/users to set manually at application launch—into an automatically adjustable format. For users, *SmartConf* allows them to specify the performance constraints they desire, without worrying about how to set and adjust PerfConfs to meet those constraints while optimizing other performance metrics.

SmartConf control-theoretic solution To automate PerfConf setting and adjustment, we explore a systematic and general control-theoretic solution to implement *SmartConf* library. We explicitly design for the four PerfConf challenges noted above, without introducing any extra parameter tuning tasks for developers or users—problems that were **not** handled by existing control theoretic solutions.

Evaluation Finally, we apply the *SmartConf* library to solve real-world PerfConf problems in widely used open-source distributed systems (Cassandra, HBase, HDFS, and MapReduce). With only 8–76 lines of code changes, we easily refactor a problematic configuration to automatically adjust itself and deliver better performance than even the best launch-time configuration settings. Our evaluation shows

Table 2. Empirical study suite

	PerfConf		AllConf	
	Issues	Posts	Issues	Posts
Cassandra	20	20	32	60
HBase	30	7	48	33
HDFS	20	7	31	39
MapReduce	10	20	13	25
Total	80	54	124	157

that, although not a panacea, *SmartConf* framework solves many PerfConf problems in real-world server applications.

2 Understanding PerfConfs

"This is hard to configure, hard to understand, and badly documented." — HBASE-13919

2.1 Methodology

We study Cassandra (CA), HBase (HB), HDFS (HD), and Hadoop MapReduce (MR). CA and HB are distributed key-value stores, HD is a distributed file system, and MR is a distributed computing infrastructure. These four systems provide a good representation of modern open-source widely used large systems.

We first study software issue-tracking systems. The detailed developer discussion there helps us understand how and why developers introduce and change PerfConfs, as well as the trade-offs. We first search fixed issues with keyword "config" or with configuration files (e.g., *hdfs-default.xml* in HD) in patches. We then randomly sample them and manually check to see if an issue is clearly explained, about configuration, and related to performance (i.e., whether developers mentioned performance impact and made changes accordingly). We keep doing this until we find 20, 30, 20, 10 PerfConf issues for CA, HB, HD, and MR, matching the different sizes of their issue-tracking systems. The details are shown in Table 2.

We also search StackOverflow [48] with key words like "config" to randomly sample 200–300 posts for each system. We then manually read through 1000 total posts to identify configuration and PerfConf posts shown in Table 2. We find the StackOverflow information less accurate than the issue-trackers, and hence only discuss user complaints in Section 2.2.1, skipping in-depth categorization.

Threats to Validity This study reflects our best effort to understand PerfConfs in modern large-scale systems. Our current study only looks at distributed systems. We also exclude issues or posts that contain little information or are not confirmed (answered) by developers (forum users). Every issue studied was cross-checked by at least two authors, and we emphasize trends that are consistent across applications.

Table 3. Different types of PerfConf patches

Category	CA	HB	HD	MR
Add a new configuration to ...				
Tune a new functionality	11	16	8	4
Replace hard-coded data	2	1	7	4
Refine an existing conf.	2	0	0	1
Change an existing configuration to ..				
Fix a poor default value	5	13	5	1

Table 4. How a PerfConf affects performance (one PerfConf can affect more than one metric)

	CA	HB	HD	MR
User-Request Latency	14	28	20	9
Internal Job Throughput	8	3	5	0
Memory/Disk Consumption	9	15	8	7
Always-on Impact	9	17	8	6
Conditional Impact	11	13	12	4
Direct Impact	7	16	8	4
Indirect Impact	13	14	12	6

2.2 Findings

2.2.1 How Common are PerfConf Problems?

As shown in Table 2, 65% of issues and 35% of posts that we studied involve performance concerns.

What are PerfConf Issues? For about half of the issues, either the default (24 of 80 cases) or the original hard-coded (14 of 80 cases) setting caused severe performance issues. Thus, the patch either changed a default setting or made a hard-coded parameter configurable. The other half added PerfConfs to support new features, as shown in Table 3.

What are PerfConf Posts? In about 40% of studied posts, users simply do not understand how to set a PerfConf. In another 60%, users ask for help to improve performance or avoid out-of-memory (OOM) problems. In about half the cases, the users ask about a specific PerfConf. In other cases, the users ask whether there are any configurations they can tune to solve a performance problem, and the answers point out some PerfConfs. Similar to a prior study [43], we found many posts related to OOM (~30%).

2.2.2 What are PerfConfs' Impact?

What Type of Performance do They Affect? As shown in Table 4, most PerfConfs affect user request latency. They also commonly affect memory or disk usage, threatening server availability through Out-of-Memory (OOM) or Out-of-Disk (OOD) failures (half the cases). Naturally, one metric could be affected by multiple PerfConfs simultaneously, with several coordination issues [4, 37].

Table 5. How to set PerfConfs

	CA	HB	HD	MR
Configuration Variable Type				
Integer	15	23	19	9
Floating Points	4	5	0	0
Non-Numerical	1	2	1	1
Deciding Factors				
Static system settings	0	1	0	1
Static workload characteristics	4	0	0	2
Dynamic factors	16	29	20	7

As Table 4 indicates, most PerfConfs affect multiple performance metrics (61 out of 80 issues). There are also 13 cases where the PerfConf has a trade-off between functionality and performance. For example, larger `mapreduce.job.counters.limit` provides users with more job statistics (functionality), but increases memory consumption (performance) and may even lead to OOM.

Most issue reports do not quantify performance impact. As our evaluation will show (Section 6), the impact could be huge, causing severe slow-downs or OOM/OOD failures.

When & How to Affect Performance? About half of PerfConfs affect corresponding performance metrics conditionally, being associated with a particular event or command. For example, in HDFS, `shortcircuit.streams.cache.size` decides an in-memory cache size, and affects memory usage continually, while the number of balancing threads `balancer.moverThreads` affects user requests only during load balancing.

Almost half of the configurations directly affect performance, such as the `cache.size` and `moverThreads` mentioned above. The other half affects performance indirectly by imposing thresholds on some system variables—e.g., queue size `ipc.server.max.queue.size`, number of operations per file `dfs.namenode.max.op.size`, and number of outstanding packets `dfs.max.packets`.

2.2.3 How to Set PerfConfs?

Format of PerfConfs A prior study shows configurations have many types [57]. PerfConfs, however, are dominated by numerical types. As shown in Table 5, the majority (>80%) are integers, and a small number of them (~10%) are floating-point. There are 5 cases where the configurations are binary and determine whether a performance optimization is enabled. A prior study shows that the difficulty of properly setting a configuration increases when the number of potential values increases [57]. Thus, due to their numeric types, PerfConfs are naturally difficult to set.

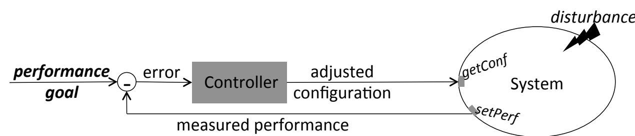

Figure 1. Using a controller to adjust PerfConf (gray parts are extra controller-related components in *SmartConf*.)

Deciding Factors of PerfConf Setting We study what factors decide the proper setting of a PerfConf based on developers' discussion and our source-code reading (Table 5). In 2 cases, the setting depends only on static system features. For example, Cassandra suggests users set the `concurrent_writes` to be $8 \times number_of_cpu_cores$. In 6 cases, it depends on workload features known before launch; e.g., input file size. Ideally, these PerfConfs would be set for each workload.

In most cases (~90%), it depends on dynamic workload and environment characteristics, such as a job's dynamic memory consumption or node workload balance. For example, in CA6059, it discusses `memtable_total_space_in_mb`, the maximum size of Cassandra server's in-memory write buffer. Depending on the workload's read/write ratio and the size of other heap objects, the optimal setting varies during run time. With no support for dynamic configuration adjustment, Cassandra developers chose a conservative setting that lowers the possibility of OOM by sacrificing write performance for many workloads.

2.3 Summary

Our study shows that PerfConf problems are common in real-world software. A single PerfConf often affects multiple performance metrics and its best setting may vary with workload and system. Thus, setting PerfConfs properly—i.e., to achieve the desired behavior in multiple metrics—is challenging for both developers and users. Ideally, these software systems would support users by automatically setting PerfConfs and dynamically adjusting them in response to changes in environment, workload, or users' goals.

3 *SmartConf* Overview

"I don't know what idiot set this [configuration] to that.. oh wait, it was me..." — HDFS-4618

SmartConf is a control-theoretic configuration framework for PerfConfs. As shown in Figure 1, under *SmartConf*, users only need to specify performance goals, instead of the exact configuration settings. With a small amount of refactoring, which we will detail later, the *SmartConf*-equipped system dynamically adjusts PerfConfs to satisfy user-defined performance goals—such as memory consumption constraints and tail latency requirements—despite unpredictable, dynamic environmental disturbances and workload fluctuations.

```
1 /* SmartConf.sys */
2 max.queue.size @ memory_consumption_max
3 max.queue.size = 50
4
5 /* HBase.conf */
6 memory_consumption_max = 1024
7 memory_consumption_max.hard = 1
```

Figure 2. *SmartConf* configurations

```
1 /* For direct configurations */
2 public class SmartConf{
3   SmartConf (string ConfName); //initialize the controller
4   void setPerf (double actual); //actual is obtained by a sensor
5   int getConf (); //controller computes the adjusted setting
6   void setGoal (double goal);
7 }
```

Figure 3. *SmartConf* class

Why controllers? Machine learning (ML) and control theory are two options that can potentially automate configuration. We choose control theory for two reasons. First, controllers—unlike ML—are specifically designed to handle dynamic disturbances [20], such as environment changes and workload fluctuation, which is crucial in setting many PerfConfs as discussed in Section 2.2.3. A controller dynamically *adjusts* a configuration based on the *difference* between the current performance and the goal, as illustrated in Figure 1. In contrast, an ML model decides exact configuration settings directly, which is more difficult in dynamically changing environments.

Second, ML methods are better than controllers in deciding optimal settings, which fortunately is unnecessary for most PerfConfs. As shown in Table 4, many configurations affect memory and disk consumptions, where the main concern is not exceeding limits instead of achieving optimal values. Even for those PerfConfs that affect request latencies, the corresponding goals are usually maintaining service-level-agreements, instead of achieving optimal latencies. Controllers are a better solution for meeting these types of constrained problems because they provide formal guarantees that they will meet the constraint. To handle PerfConfs, we modify standard control techniques but still provide probabilistic guarantees. ML would be a better choice if the goal was finding the best performance rather than meeting a performance constraint with guarantees.

What are the challenges? Our goal is to make control-theoretic benefits available to developers who are not trained in control engineering. There are two high-level challenges: (1) how to automatically synthesize controllers that can address unique challenges in the context of PerfConfs and (2) how to allow developers to easily use controllers to adjust a wide variety of configurations in real-world software systems with little extra coding. We discuss how *SmartConf* addresses these two challenges in the next two sections.

4 *SmartConf* Framework

"It will be even great if we can dynamically tune/choose a proper one." — HBASE-7519

SmartConf provides a library for developers who want to have any configuration C automatically and dynamically adjusted to meet a goal of a performance metric M, such as request latency, memory consumption, etc. This section describes what developers and users need to do to use *SmartConf* library and configurations. Section 5 describes how *SmartConf* library is implemented with new control theoretic techniques in detail.

4.1 Developers' effort

4.1.1 General Code Refactoring

First, developers must provide a sensor that measures the performance metric M to be controlled. Such sensors are sometimes already provided by existing software. For example, MapReduce contains sensors that measure and maintain up-to-date performance metrics in variables, such as heap consumption in `MemHeapUsedM`, average request latency in `RpcProcessingAvgTime`, etc.

Second, developers create a *SmartConf* system file invisible to users, as shown in Figure 2. In this system file, developers specify the mapping from a *SmartConf* configuration entry C to its corresponding performance metric M and provide an initial setting for C, which only serves as C's *starting* value before the first run. After software starts, this field will be overwritten by the *SmartConf* controller. As we will see in the evaluation section, the quality of this initial setting does not matter.

Third, developers replace the original configuration entry C in the configuration file with new entries $M.goal$ and $M.goal.hard$ that allow users to specify a numeric goal for M and whether this goal is a hard constraint, as shown in Figure 2. For example, a goal about "memory consumption being smaller than the JVM heap size" is a hard constraint.

4.1.2 Calling *SmartConf* APIs

After the above code refactoring, developers can use *SmartConf* APIs.

Initializing a SmartConf Configuration Instead of reading a configuration value from the configuration file into an in-memory data structure, developers simply create a `SmartConf` object `sc`. As shown in Figure 3, the constructor's parameter is a string naming the configuration. Using this string name, the *SmartConf* constructor reads the configuration's current setting, its performance goal, and other auto-generated parameters from the *SmartConf* system file, and then initializes a controller dedicated for

this configuration, which we will explain more in the next section.

Using a SmartConf Configuration Whenever the software needs to read the configuration, `SC.setPerf` is invoked followed by `SC.getConf`. `setPerf` feeds the latest performance measurement `actual` to an underlying controller, and `getConf` calls the controller to compute an adjusted configuration setting that can close the gap between `actual` performance and the goal.

4.2 Handling Special Configuration Types

The discussion above assumes a basic configuration that directly affects performance all the time. Next, we discuss how *SmartConf* handles more complicated configurations. Only one type requires extra effort from developers.

Indirect Configurations Sometimes, a configuration C affects performance indirectly by imposing constraints on its deputy C'. For example, in HBase, `max.queue.size` limits the maximum size of a queue. The size of the queue, denoted as `queue.size`, then directly affects memory consumption. To handle indirect configurations like `max.queue.size`, a few steps in the above recipe need to change.

First, when creating the configuration object, developers should use the sub-type `SmartConf_I` as shown in Figure 4. Furthermore, developers initialize the constructor with a `transducer` function that maps the desired value of deputy C' to the desired value of configuration C. In most cases, this transducer function simply conducts an identical mapping—if we want the `queue.size` to drop to K, we drop `max.queue.size` to K—and developers can directly use the default transducer function provided by *SmartConf* library as shown in Figure 4.

Second, while updating the current performance through `SC.setPerf`, developers need to provide the current value of C'—like the current `queue.size`, which is needed for the controller to adjust the value of C. The control theoretic reasoning behind this designed is explained in the next section.

Finally, developers need to check every place where the configuration is used to make sure that temporary inconsistency between the newly updated configuration C and the deputy C' is tolerated. For example, at run time the `queue.size` may be larger than a recently dropped `max.queue.size`. The right strategy is usually to ignore any exception that might be thrown due to this inconsistency, and simply wait for C' to drop back in bound. This change is needed for any system that supports dynamic configuration adjustment.

Conditional Configurations As discussed in Section 2, some configurations affect performance metrics conditionally. Consequently, their controllers should only be invoked when they take effect. Fortunately, this is already taken care of by the baseline *SmartConf* library, because developers naturally only invoke `SC.setPerf` and `SC.getConf` when the software is to use the configuration.

```
 1 /* For indirect configurations */
 2 public class SmartConf_I extend SmartConf {
 3   SmartConf_I (string ConfName, Transducer t);
 4   void setPerf (double actual, int deputyConf);
 5 }
 6
 7 /* Tranducer super class. Developers can customize a subclass.*/
 8 public class Transducer {
 9   int transduce (int input) {return input};
10 }
```

Figure 4. *SmartConf* sub-class

Correlating Configurations Some configurations may affect the same performance goal simultaneously, and their corresponding controllers need to coordinate with each other. This case is transparently handled by *SmartConf* library and its underlying controllers synthesized by *SmartConf*. As long as developers specify the same performance metric M for a set of configurations $\mathbb{C}$, *SmartConf* will make sure that their controllers coordinate with each other. We will explain the control theoretic details in the next section.

4.3 Users' Effort

With the above changes, users are completely relieved of directly setting performance-sensitive configurations.

In the configuration file, users simply provide two items to describe the performance goal associated with a *SmartConf* configuration. First, a numerical number that specifies the performance goal, which could be the desired latency of user request, the maximum size of the memory consumption, etc. Second, a binary choice about whether or not the corresponding goal imposes a hard constraint. Developers provide default settings for these items, such as setting the memory-consumption goal to be the JVM heap size, just like that in traditional configuration files. When users specify goals that cannot possibly be satisfied, *SmartConf* makes its best effort towards the goal and alerts users that the goal is unreachable.

Users or administrators can update the goal at run time through the `setGoal` API in Figure 3.

5 *SmartConf* Controller Design

"everything always has a tradeoff." — CASSANDRA-13304

Baseline controller We choose a recently proposed controller-synthesis methodology [12] as the foundation for *SmartConf* controller. This methodology first *approximates* how system performance reacts to a configuration by profiling the application and building a regression model

relating performance to configuration settings:

$$s_k = \alpha \cdot c_{k-1} \quad (1)$$

where s_k is the system performance measured at time k and c_{k-1} is the configuration value at time $k-1$. A controller is then synthesized to select the configuration parameter's next value c_{k+1} based on its previous value c_k and the error e_{k+1} between the desired $\tilde{s}$ and measured performance s_{k+1}:

$$c_{k+1} = c_k + \frac{1-p}{\alpha} e_{k+1}. \quad (2)$$

where p is the *pole* value that determines how aggressively the controller reacts to the current error.

Although simple, the above controller is robust to model inaccuracy and does not demand intensive profiling. We will explain more about this in Section 5.6.

Challenges for *SmartConf* Unfortunately, the baseline controller, as well as *all* existing control techniques, cannot handle several challenges unique and crucial to PerfConfs.

1. How to automatically set the pole p, to hide this control parameter from users.

2. How to handle hard goals that do not allow overshoot, such as memory consumption.

3. How to handle the indirect relationships between some configurations and performance.

4. How to handle multiple interacting configurations so that their controllers do not interfere with each other.

We explain how these challenges are addressed below.

5.1 How to Decide the Pole Parameter

It is difficult for developers with no control background to set this value, so *SmartConf* sets it automatically.

The pole p determines the controller's tolerance for errors between the model built during profiling and the true behavior. Given an error Δ between the true performance s and the modeled performance $\hat{s}$, where $\Delta = s/\hat{s}$, the pole can simply be set to $p = 1 - 2/\Delta$, if $\Delta > 2$ and $p = 0$ otherwise. Setting p thusly guarantees the controller will converge [20].

Of course, we do not expect *SmartConf* users to know Δ, or even be aware of these control specific issues. Therefore, *SmartConf* projects Δ based on the system's (in)stability during profiling: $\Delta = 1 + \frac{1}{N}\sum_1^N \frac{3\sigma_i}{m_i{}'}$, where σ_i and $m_i{}'$ are the standard deviation and mean of the performance measured *w.r.t* minimum performance under the i-th sampled configuration value. This equation provides a statistical guarantee that the controller will converge to the desired performance as long as the error between the model built during profiling and the true response is correct to within three standard deviations (i.e., 99.7% of the time).

5.2 Handle Hard Goals

Many PerfConfs are associated with a `hard==1` constraint, meaning that the goals like no OOM cannot be violated (Table 4). Handling these *hard* constraints is crucial for system availability. Unfortunately, traditional controllers can limit overshoot (i.e., the maximum amount by which the system may exceed the goal) only in continuous physical systems, **not** in discrete computing systems where a disturbance could come suddenly and discretely. For example, a new process could unexpectedly allocate a huge data structure.

Strawman One naive solution is to choose an extremely insensitive pole p (e.g., close to 1), so that the output performance will move very slowly towards the goal, making overshooting unlikely. Unfortunately, this strategy does not work, as will be shown in experiments (Section 6). It introduces extremely long convergence process, which sacrifices other aspects of performance and still cannot prevent overshooting when system dynamics encounter disturbance.

A better strawman Recent work that uses controllers to avoid processor over-heating [47] proposes a *virtual goal* $\tilde{s}^v$ that is smaller than the real constraint $\tilde{s}$. The controller then targets $\tilde{s}^v$, instead of $\tilde{s}$. Unfortunately, this work still has two key limitations. First of all, while it works well for temperature—which changes slowly and continuously—it does not work well for goals like memory—which can change suddenly and dramatically. Second, it relies on expert knowledge to manually set the virtual goal $\tilde{s}^v$, without providing a general setting methodology.

Our Solution *SmartConf* proposes two new techniques to address goals that do not allow overshoot: automated virtual-goal setting and context-aware poles.

First, *SmartConf* proposes a general methodology to compute the virtual goal $\tilde{s}^v$ considering system stability under control. Intuitively, if the system is easily perturbed, $\tilde{s}^v$ should be far-away from $\tilde{s}$ in order to avoid accidental overshooting. Otherwise, $\tilde{s}^v$ can be close to $\tilde{s}$ for better resource utilization.

To measure the system stability, we compute the coefficient of variation λ during the performance profiling at the model-building phase. That is, $\lambda := \frac{1}{N}\sum_1^N \frac{\sigma_i}{m_i}$, where σ_i and m_i are the standard deviation and mean of the performance measured under the i-th sampled configuration value. Clearly, the bigger λ is, the more unstable the system is and hence the lower $\tilde{s}^v$ should be. Following this intuition, we compute $\tilde{s}^v$ by (1-λ)*$\tilde{s}$.

Second, *SmartConf* uses *context-aware* poles that are conservative when the system is "safe" and aggressive when in "danger". Specifically, before the virtual goal $\tilde{s}^v$ is reached, we use the regular pole, discussed in Section 5.1. This pole is tuned to provide maximum stability given the natural system variance and may sacrifice reaction time for stability. After

$\tilde{s}^v$ is reached, we use the smallest possible pole, 0, which moves the system back into the safe region as quickly as possible.

As we can see, *SmartConf* handles hard goals without requiring any extra inputs from users or developers. The implementation of *SmartConf* API `SmartConf::getConf` will automatically switch to the above algorithm (i.e., two poles and virtual goal) once the configuration file specifies a performance goal with the configuration attribute `hard==1`. Experiments in Section 6 demonstrate that the above two techniques are both crucial to avoid over-shooting while maintaining high resource utilization.

5.3 Handle Configurations with Indirect Impact

A PerfConf C may serve as a threshold for a deputy variable C' (~50% among PerfConfs in Table 4). Directly modeling the relationship between performance and C is difficult, as changing C often does not immediately affect performance.

SmartConf handles this challenge by building a controller for the deputy variable C' using the technique discussed earlier and adjusting the threshold configuration C based on the controller-desired value of C'. Specifically, at run time, the controller computes the desired next value of C' based on the current performance and the current value of C', which is why the `SmartConf_I::getPerf` function needs two parameters (Figure 4). *SmartConf* then adjusts C to move C' to the desired value. If C simply specifies the upper-bound or lower-bound of C', *SmartConf* sets C to C'_{next}. If the relationship is more complicated, developers need to provide a custom `transducer` function as shown in Figure 4.

For example, *SmartConf* profiles how software memory consumption changes with `queue.size`, and computes how to adjust `queue.size` based on the current memory consumption. If the desired size q is smaller than `max.queue.size`, *SmartConf* drops `max.queue.size` to q. This does not immediately shrink `queue.size`, but will prevent the queue from taking in new RPC requests until `queue.size` drops to q.

5.4 Handle Multiple, Interacting PerfConfs

The discussion so far assumes *SmartConf* creates an independent controller for each individual configuration. It is possible that multiple configurations—and hence multiple controllers—are associated with the same performance constraint, as implied by Table 4. We must ensure that each controller works with others towards the same goal. For example, when two controllers independently decide to increase `q1.size` and `q2.size`, *SmartConf* must ensure no OOM.

Traditional control techniques synthesize a single controller that sets all configurations simultaneously. This approach demands much more complicated profiling and controller building, essentially turning a $O(K \cdot N)$ problem into a $O(N^K)$ problem, assuming K PerfConfs each with N possible settings [13]. Furthermore, it is fundamentally unsuitable for PerfConfs, as different PerfConfs may be developed at different times as software evolves, and they may be used in different modules and moments during execution. We assume developers will call `getPerf` and `setConf` at the places the program uses a PerfConf value. Traditional techniques for coordinating control would require all `getPerf` and `setConf` calls be made in the same location at the same time, which we believe is infeasible in a large software system.

Therefore, instead of synthesizing a single complicated controller to set all configurations simultaneously, *SmartConf* uses a protocol such that controllers will independently work together. When we synthesize the controller for C, the performance impact of related configurations is part of the disturbance captured during profiling and hence affects how *SmartConf* determines the pole (Section 5.1) and the virtual goal (Section 5.2). As we will discuss soon in Section 5.6, even if the profiling is incomplete, our controller-synthesis technique still provides statistical guarantees that the goal will be satisfied.

When developers are extremely cautious about not violating a performance goal or feel particularly unsure about the profiling, *SmartConf* provides a safety net by applying an interaction factor N to Equation 2. Specifically, developers can mark a specific performance goal—e.g., memory consumption or 99 percentile read latency—as *super*-hard. While processing the *SmartConf* system file, *SmartConf* counts how many, configurations are associated with this super-hard goal. Then, when initializing a corresponding controller c, *SmartConf* will use $c_k + \frac{1-p}{N\alpha} e_{k+1}$ instead of $c_k + \frac{1-p}{\alpha} e_{k+1}$ as the formula to compute the setting of c_{k+1}, splitting the performance gap e_{k+1} evenly to all N interacting configurations.

5.5 Other Implementation Details

Our *SmartConf* library is implemented in Java. The `SmartConf` classes shown in Figure 3 and Figure 4 contain private fields representing the configuration name `ConfName`, current configuration setting, current performance, and controller parameters, including pole, α, goal, and virtual goal (for `SmartConf_I` class). These controller parameters are computed inside the `SmartConf` constructor based on the profiling results stored in a configuration-specific file `<ConfName>.SmartConf.sys`. Of course, future implementations can change to compute these parameters only once after all the profiling is done.

The *SmartConf* system file `SmartConf.sys` contains an entry that allows developers to enable or disable profiling. Once profiling is enabled, the calling of `SmartConf::setPerf` records the current performance measurement not only in the `SmartConf` object but also in a buffer, together with the current (deputy) configuration value, periodically flushed to file `<ConfName>.SmartConf.sys`, which will be read during the initialization of configuration `<ConfName>`.

Table 6. Benchmark suite and workload. ?-?-? under a bug ID shows whether the PerfConf is conditional, direct, and hard. In issue description, the main constraint that users complain about is put earlier, and the trade-off is later. For YCSB [6] workload, xW, write portion; yMB, request size; Cz, read index cache ratio. ts, latency constraint. Wordcount(x,y,z): input file size; split size; parallelism per worker

ID	Issue Description	Profiling Workload	Evaluation Workload	
			Phase-1	Phase-2
CA6059	`memtable_total_space_in_mb` limits the memtable size.	YCSB$_A$ 0.5W, 1MB	YCSB	YCSB
N-N-Y	Too big, OOM; Too small, write latency hurts.		1.0W, 1MB, C0	0.9W, 1MB, C0.5
HB2149	`global.memstore.lowerLimit` decides how much memstore data is flushed.		YCSB	YCSB
Y-Y-N	Too big, write blocked for too long; Too small, write blocked too often.		1.0W, 1MB, 10s	1.0W, 1MB, 5s
HB3813	`ipc.server.max.queue.size` limits RPC-call queue size.		YCSB	YCSB
N-N-Y	Too big, OOM; Too small, read/write throughput hurts.		1.0W, 1MB	1.0W, 2MB
HB6728	`ipc.server.response.queue.maxsize` limits RPC-response queue size.		YCSB	YCSB
N-N-Y	Too big, OOM; Too small, read/write throughput hurts.		0.0W, 2MB	0.3W, 2MB
HD4995	`content-summary.limit` limits #files traversed before `du` releases big lock.	TestDFSIO	TestDFSIO	TestDFSIO
Y-N-N	Too big, write blocked for long; Too small, `du` latency hurts.	single-client	multi-clients, 20s	multi-clients, 10s
MR2820	`local.dir.minspacestart` decides if a worker has enough disk to run task.	WordCount	WordCount	WordCount
Y-Y-Y	Too small, OOD; Too big, low utility (job latency hurts).	2G, 64MB, 1	640MB, 64MB, 2	640MB, 128MB, 2

Profiling To model the effects the controller has on the target performance metric, a few performance measurements need to be taken by running profiling workloads while varying the configuration parameter to be controlled. The larger the range of workloads, the more robust the control design will be when working with previously unseen workloads. We also base the pole and the virtual goal on the measured mean and standard deviation, so enough samples are needed for the central limit theorem to apply. As we will formally discuss below and experimentally demonstrate in Section 6, *SmartConf* produces effective and robust controllers without intensive profiling.

5.6 Formal Assessment and Discussion

Stability We want the system under control to be *stable*. That is, it should converge to the desired goal rather than oscillate around it, which could cause unpredictable performance or software crashes. Based on analysis in previous work, the controller in equation 2 is stable as long as $0 \le p < 1$ and $p = 1 - 2/\Delta$ for $\Delta > 2$ [12]. Unlike prior work, *SmartConf* assumes Δ is unknown, so we provide a weaker probabilistic guarantee that the system will converge as long as the error is within three standard deviations of the true value. This guarantee comes without requiring users to have control-specific knowledge.

Overshoot We hope to ensure that hard goals that do not allow overshoot are respected. Following traditional control analysis, *SmartConf* is free of overshooting because its design ensures $0 \le p < 1$ [20]. Such analysis, however, assumes no disturbances, but we know we are deploying *SmartConf* into unpredictable environments.

With two enhancements discussed in Section 5.2, we avoid overshooting with high probability even in unpredictable environments. By setting the virtual goal to $\lambda := \frac{1}{N}\sum_1^N \frac{\sigma_i}{m_i}$, we provide 84% probability of being on the "safe" side of no-overshoot goals.[1] The two-pole enhancement further increases the likelihood that *SmartConf* respects the constraint, because any measurement above the virtual goal causes the largest possible reaction in the opposite direction.

6 Evaluation

"I think going to 1G [default] works, ... let's do some testing before submitting a patch" – HBASE-4374

6.1 Evaluation methodology

Benchmarks We apply *SmartConf* to 6 PerfConf issues in Cassandra, HBase, HDFS, and MapReduce, as shown in Table 6. These 6 cases together cover a variety of configuration features, like conditional or not, direct or not, hard constraint or not, as listed by ?-?-? sequence in Table 6. We consider bug-reporters' main concern as the performance goal, and the trade-off mentioned by users or developers as the trade-off metric that we want to optimize while satisfying the goal, both listed in the issue description of Table 6. They cover a variety of performance metrics, memory, disk, latency, etc.

Workloads *SmartConf* works in a wide variety of workload settings, but we do not have space to show that. Therefore, in this section, our workload design follows several principles: (1) profiling and evaluation workload are **different**, so

[1] Assuming a normal distribution, 68% of samples are within 1 standard deviation, which means 16% is higher and 16% is lower. In our case, however, one side is safe, so we have an 84% probability of not overshooting.

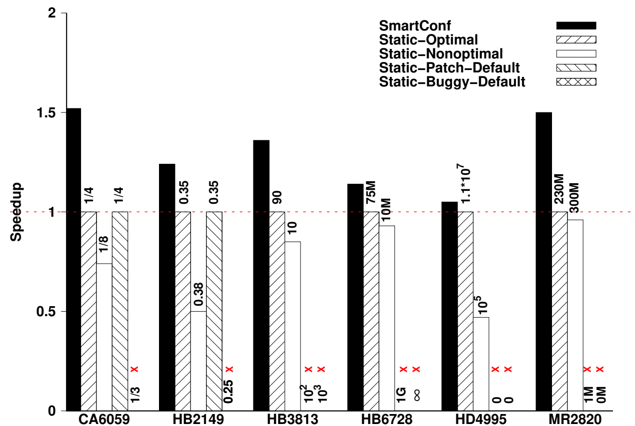

Figure 5. Trade-off performance comparison. Normalized upon the best-performing static configuration; x: fail constraints. The numerical PerfConf settings are above each bar.

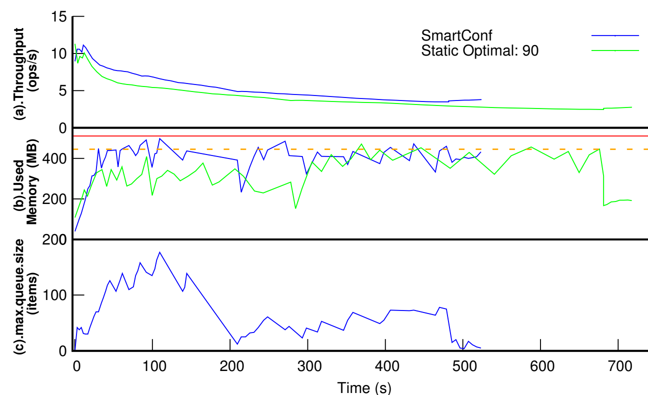

Figure 6. *SmartConf* vs. static optimal on HB3813. workload changes at ~200s. Throughput is accumulative.

that we can evaluate how sensitive *SmartConf* is towards profiling; (2) the evaluation workload contains two phases where either the workload or the performance goal changes (HB2149, HB6728), so that we can evaluate how well *SmartConf* reacts to changing **dynamics**; (3) at least one phase of the evaluation workload triggers the performance **problems** complained by users in the original bug reports, so that we can test whether *SmartConf* automatically addresses users' PerfConf problems. Finally, we use standard profiling workloads to demonstrate *SmartConf*'s robustness. Specifically, for key-value stores, we use the popular YCSB [6] benchmark workload-A, which has a 50-50 read-write ratio; for HDFS, we use a common distributed file system benchmark TestDFSIO [27]; and we use WordCount for MapReduce, as shown in Table 6.

Profiling As discussed in Section 5, *SmartConf* decides controller parameters, such as α and p in Equation 2, based on profiling results. For each PerfConf C, our profiling tries 4 different settings of C and collects 10 performance measurements under each setting. These 4 settings are chosen to provide a good coverage of the valid value range of C, and the performance measurement is taken every time `setConf` API is invoked on C at run time (i.e., every time C is used). In total, the profiling provides 40 sample points, which are sufficient for building linear regression models following the sample-size rule-of-thumb [24]. For example, in case of HB3813, the four profiled settings of `ipc.server.max.queue.size` are 40, 80, 120, and 160. Under each setting, a performance measurement is taken every time an RPC request is enqueued.

Machines We use two servers to host virtual machines. Each server has 2 12-core Intel Xeon E2650 v3 CPU with 256GB RAM. Ubuntu 14.04 and JVM 1.7 are installed. We use virtual machines to host distributed systems under evaluation, with 2–6 virtual nodes set up for each experiment.

6.2 Does *SmartConf* Satisfy Constraints?

SmartConf always tracks the changing dynamics, satisfying the performance constraints for all 6 issues. These include hard constraints—preventing out-of-memory (CA6059, HB3813, HB6728) and out-of-disk (MR2820) —and soft constraints on worst-case write latency (HB2149, HD4995).

It is difficult for statically set configurations to satisfy performance constraints. The original default settings in all 6 issues fail, denoted by the red-crosses for `static-buggy-default` bars in Figure 5, which is why users filed issue reports. In our experiments, even the patched default settings fail to satisfy corresponding constraints in 4 cases. In HD4995, developers simply moved a problematic hard-coded parameter into the configuration file without changing the default setting and asked users to figure out a suitable custom setting for themselves. In HB3813, HB6728, and MR2820, the patches made the configurations more conservative, from 1000, ∞, and 0 to 100, $1G$, and $1M$ respectively. However, the new settings still failed. In fact, we can easily find workloads to make the patched default settings in the remaining 2 issues fail, too.

Case Study We take a closer look at how *SmartConf* handles HB3813. Here, `max.queue.size` decides the largest size for an RPC queue. When the system is under memory pressure, a large queue can cause an out-of-memory (OOM) failure. Unfortunately, a small queue reduces RPC throughput.

Figure 6b shows how memory consumption changes at run time under different configuration settings. The red horizontal line marks the hard memory-consumption constraint (495MB), and the orange dashed line marks *SmartConf*'s automatically determined virtual goal of 445MB. The blue curve shows how memory consumption changes under *SmartConf*'s automated management. While under the dashed line—in a "safe zone"—the system takes new RPC requests,

SmartConf slowly raises `max.queue.size` from its initial value 0—shown by the blue curves in Figure 6c—and the memory consumption increases. Once over the dashed line, *SmartConf* quickly decreases `max.queue.size`—shown by the dips of the blue curve in Figure 6c—and the memory drops. Even when the workload shift increases each RPC request size (at about the 200 second point), the memory consumption is always under control, as *SmartConf* reacts to the workload change by dropping the `max.queue.size` to around 50—as shown by the blue curves—after 200 seconds in Figure 6c. Overall, the system never has OOM errors with *SmartConf*.

In comparison, the old default setting, 1000, causes OOM almost immediately after the first workload starts; even the new default setting in the patch, 100, still causes OOM shortly after the second workload starts. A conservative setting—e.g., 90 in this experiment—avoids OOM, shown by the green curves in Figure 6ab. However, there is no way for users/developers to predict what configuration will be conservative enough for the future workload.

6.3 Does *SmartConf* Provide Good Tradeoffs?

Figure 5 shows that *SmartConf* provides performance tradeoffs better than the best static configuration. While all of our case studies have different constraints, they all must optimize latency or throughput under those constraints. The figure shows *SmartConf*'s speedup in these secondary metrics relative to various static configurations.

We find the best static configuration by exhaustively searching all possible PerfConf settings that meet the constraint throughout our two-phase workloads. These best settings are often sub-optimal or even fail performance constraints once workloads change. Figure 5 also shows the performance under randomly chosen static settings.

SmartConf outperforms the best static setting because it automatically adapts to dynamics. Although *SmartConf* may start with a poor initial configuration (e.g., 0 in Figure 6c), it quickly adjusts so that the constraint is *just* met and the tradeoffs are optimal. When the workload changes from phase-1 to phase-2 in our experiments, *SmartConf* quickly adjusts again. In comparison, since different phases have different constraints, a static configuration can only be optimal for one phase and must sacrifice performance for the other.

For example, as shown in Figure 6ab, to avoid OOM during both phases, the static optimal configuration (90) is too conservative and unnecessarily reduces memory during the first phase. In contrast, *SmartConf* is never too conservative or too aggressive. Throughout the two phases, *SmartConf* achieves 1.36× speedup in write throughput.

As another example, in MR2820, to make sure WordCount can succeed in both phases, the best static setting for `minspacestart` is 230MB, because phase-2 requires that much disk space to run. However, this is overly conservative for phase-1 that produces much smaller intermediate files.

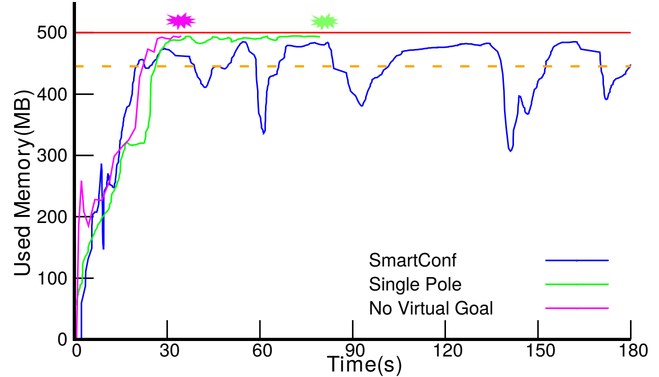

Figure 7. *SmartConf* vs. alternative controllers.

Consequently, *SmartConf* runs WordCount much faster in phase-1 and achieves 1.50× total speedup.

6.4 Alternative Design Choices

SmartConf's controller handles hard constraints differently from traditional control design in two ways (Section 5.2). We experimentally compare *SmartConf* with the traditional alternatives below.

A Single Pole with a Good Virtual Goal Traditional control design handles hard constraints—e.g., avoiding processor over-heating [47]—by using a single conservative pole and a virtual goal. We briefly compare this traditional design to *SmartConf* by recreating the HB-3813 case study using a less stable workload (70% write with 30% read). We let *SmartConf* and this alternative controller use the same virtual goal and the same pole 0.9. The only difference is that *SmartConf* has a second pole, 0, for post-virtual-goal use.

As shown in Figure 7, *SmartConf* still behaves well, yet the single-pole alternative controller causes an OOM at time 80s. Around 25s, both controllers start to limit queue size, but the alternative one is simply too slow. When close to the memory limit—i.e., beyond the virtual goal—that slowness is catastrophic because just a few extra RPC requests can cause a system crash. Overall, *SmartConf* is conservative when growing the queue and extremely aggressive when shrinking it. In contrast, with only one pole, the alternative controller is conservative when growing the queue and too conservative when shrinking it.

Without (a good) Virtual Goal Traditional control design does not consider virtual goals. We rerun the HB3813 example, this time targeting the actual system memory instead of the virtual goal determined by *SmartConf*. As shown in Figure 7, the system quickly over-allocates memory leading to a JVM crash at about 36s. The virtual goal is essential for meeting hard constraints because it gives *SmartConf*'s controller time to react to unexpected situations. Needless to say, selecting the right virtual goal is crucial. A careless

Table 7. Lines of code changes for using *SmartConf*

ID	Sensor	Invoke APIs	Others	Total
CA6059	35	6	1	42
HB2149	31	6	1	38
HB3813	2	6	9	17
HB6728	2	6	0	8
HD4995	70	6	0	76
MR2820	53	8	4	65

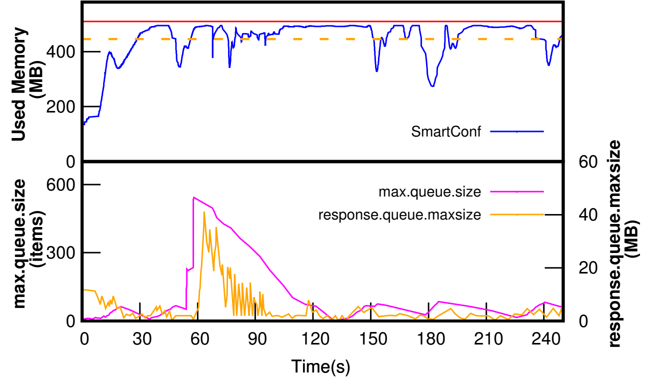

Figure 8. *SmartConf* adjusts two related PerfConfs.

selection easily leads to violating constraints or wasting resources. We skip the experimental results here due to space constraints.

6.5 Other results

Is *SmartConf* easy to use? As shown in Table 7, it takes little effort for developers to adopt *SmartConf*, as few as 8 lines of code changes. In most cases, the changes are dominated by implementing performance sensing. Occasionally, there are code changes unrelated to performance sensing or *SmartConf*-API invocation, and are counted in the "Others" column in Table 7. For example, in HB3813, changes are needed to convert a queue's length from statically fixed to dynamically adjustable. For MR2820, changes are needed to deliver the configuration computed by one node, the Master node, to another, the Slave node.

Interacting controllers To evaluate whether *SmartConf* can handle multiple interacting PerfConfs, as mentioned in Section 5.4, we apply *SmartConf* to tackle HB3813 and HB6728 simultaneously. The PerfConfs in these two cases limit the size of RPC-request queue and RPC-response queue, respectively, both affecting memory consumption. We start with a workload of writes, occupying a large space in the request queue and a small space in the response queue. After 50 seconds, we add a second workload of reads, which take small space in the request queue and large space in the response queue. Figure 8 shows the results. When the second workload just starts, the request queue quickly fills with many small read requests, and the response queue jumps up. Then, *SmartConf* reacts by bringing the size of both queues down dramatically. After this initial disturbance, the size of each queue dynamically fluctuates: during periods where more read requests enter the system, the response queue size is limited; when there are more write requests, the RPC queue size is throttled. At no time is the memory constraint (red line) violated.

This study demonstrates that multiple PerfConfs can be composed and *SmartConf* can still guarantee the hard constraint in doing so. It also further motivates dynamically adjusting configurations — otherwise, we would have to pick very small sizes for both queues.

6.6 Limitations of *SmartConf*

SmartConf also has its limitations. First, it does not work for configurations whose performance goals are about optimality instead of constrained optimality. For example, MR5420 discusses how to set `max_chunks_tolerable` which decides how many chunks that input files can be grouped into during distributed copy. The on-line discussion shows that users only care about one goal here — achieving the fastest copy speed. Consequently, *SmartConf* is not a good fit.

Second, the current *SmartConf* design does not work if the relationship between performance and configuration is not monotonic. This happens to be the case in MR5420 — if there are too few chunks, the copy is slow due to load imbalance; if there are too many chunks, the copy is also slow due to lack of batching. Machine learning techniques would be a better fit for these two challenges. In our empirical study of 80 PerfConfs in Cassandra, HBase, HDFS, and MapReduce, such non-monotonic relationships occur to less than 10% of PerfConfs. Of course, the non-monotonic relationship could occur at corner cases that we did not check.

Third, some configurations might be inherently difficult to adjust dynamically, as the adjustment may cost run-time performance or code-refactoring effort. For example, changing `max_chunks_tolerable` dynamically may require copying files around and degrade system performance; allowing a system to change its number of worker threads dynamically would require non-trivial coding in its work dissemination and work-progress monitoring components.

Fourth, in distributed environment, additional inter-node communication may be required for some performance measurement and configuration adjustment, like that in MR2820. Since the amount of data transfer is negligible, we expect little performance degradation from such communication.

Finally, *SmartConf* provides statistical guarantees as discussed in Section 5.6, but cannot guarantee all constraints to be always satisfied.

7 Related Work

"What is the proper way of setting the configuration values programmatically?" – MapReduce-12825547

Control theoretic frameworks Control theory provides a general set of mechanisms and formalisms for ensuring that systems achieve desired behavior in dynamic environments [33]. While the great body of control development has targeted management of physical systems (e.g., airplanes), computer systems are natural targets for control since they must ensure certain behavior despite highly dynamic fluctuations in available resources and workload [19, 32].

While control theory covers a wide variety of general techniques, control applications tend to be highly specific to the system under control. The application-specific nature of control solutions means that controllers that work well for one system (e.g. a web-server [35, 49] or mobile system [34]) are useless for other systems.

Thus, a major thrust of applying control theory to computing systems is creating general and reusable techniques that put control systems in the hands of non-experts [14]. Towards this end, recent research synthesizes controllers for software systems [12, 13, 46]. Other approaches package control systems as libraries that can be called from existing software [28, 44, 67]. While these techniques automate much of the control design process, they still require users to have control specific knowledge to specify key parameters, like the values of p and α, and choose what controllers to use. Furthermore, none of them address the PerfConf specific challenges of meeting hard constraints, using indirect and interacting parameters, etc.

In addition to solving PerfConf-specific challenges, *SmartConf* is unique in hiding all control-specific information from the users/developers. Thus, *SmartConf*'s interface works at a much higher-level of abstraction than prior work that encapsulates control systems. In fact, *SmartConf*'s implementation could swap a control system for some other management technique in the future. In exchange for its higher level of abstraction, *SmartConf* provides only probabilistic guarantees rather than the stronger guarantees that would come from having an expert set a pole based on a known error bound. SmartConf's ability to automatically adjust to meet user-defined goals is an example of a *self-aware* computing system [23].

Machine learning frameworks Many learning approaches have been proposed for predicting an optimal configuration within a complicated configuration space [8–10, 31, 40, 53, 65]. Machine learning has even been applied to further improve existing heuristic autotuners, like Starfish [21], by using learning models to direct the search for optimal configurations [5, 60]. Perhaps the most closely related learning works are those based on reinforcement learning (RL) [51]. Like control systems, RL takes online feedback. Several RL methods exist for optimizing system resource usage [3, 15, 29, 30, 38]. RL techniques, however, are not suited to meeting constraints in dynamic environments [50]. In contrast, that is exactly what control systems are designed to do, and they produce better empirical results than RL on such constrained optimization problems [36]. For this reason, several recent efforts propose combining machine learning and control theory [7, 22, 39, 45, 52]. We will explore such a combination in future work where we will investigate replacing our exhaustive profiling with more scalable learning approaches.

Misconfiguration Many empirical studies have looked at misconfiguration [16, 41, 57, 62], but did not focus on PerfConfs. Much previous work has proposed using static program analysis [42, 58] or statistical analysis [55, 56, 63, 66] to identify and fix wrong or abnormal configurations. These techniques mainly target functionality-related misconfigurations and do not work for PerfConfs, as the proper setting of a PerfConf highly depends on the dynamic workload and environment, and can hardly be statistically decided based on common/default settings. Techniques were also proposed to diagnose misconfiguration failures [2, 54] and misconfiguration-related performance problems [1]. They are complementary to *SmartConf* that helps avoid misconfiguration performance problems.

8 Conclusions

Large systems are often equipped with many configurations that allow customization. Many of these configurations can greatly affect performance, and their proper setting unfortunately depends on complicated and changing workloads and environments. We argue that the traditional way of letting users statically and directly set configuration values is fundamentally flawed. Instead, a new configuration interface is designed to allow users and developers to focus on specifying what performance constraints a configuration should satisfy, and a control-theoretic technique is designed to enable automated and dynamic configuration adjustment based on the performance constraints. Our evaluation shows that the *SmartConf* framework can out-perform static optimal configuration setting while satisfying performance constraints.

Acknowledgments

We would like to thank the anonymous reviewers for their insightful feedback and comments. This research is supported by NSF (grants CNS-1563956, IIS-1546543, CNS-1514256, CCF-1514189, CCF-1439091, CCF-1439156), and generous support from the CERES Center for Unstoppable Computing. Experiments were conducted on the RIVER (CNS-1405959) and Chameleon testbeds supported by the National Science Foundation. Henry Hoffmann's effort is additionally supported by the DARPA BRASS program and a DOE Early Career award.

References

[1] Mona Attariyan, Michael Chow, and Jason Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *OSDI*, 2012.

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

[3] Ramazan Bitirgen, Engin Ipek, and Jose F. Martinez. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In *MICRO*, 2008.

[4] CASSANDRA-1007. Make memtable flush thresholds per-cf instead of global. https://issues.apache.org/jira/browse/CASSANDRA-1007.

[5] 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 *BigData Congress*, 2015.

[6] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *SoCC*, 2010.

[7] Sarah Dean, Horia Mania, Nikolai Matni, Benjamin Recht, and Stephen Tu. On the sample complexity of the linear quadratic regulator. Technical Report 1710.01688v1, arXiv, 2017.

[8] Christina Delimitrou and Christos Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. In *ASPLOS*, 2013.

[9] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. In *ASPLOS*, 2014.

[10] Zhaoxia Deng, Lunkai Zhang, Nikita Mishra, Henry Hoffmann, and Fred Chong. Memory cocktail therapy: A general learning-based framework to optimize dynamic tradeoffs in nvm. In *MICRO*, 2017.

[11] Lu Fang, Khanh Nguyen, Guoqing Xu, Brian Demsky, and Shan Lu. Interruptible tasks: Treating memory pressure as interrupts for highly scalable data-parallel programs. In *SOSP*, 2015.

[12] Antonio Filieri, Henry Hoffmann, and Martina Maggio. Automated design of self-adaptive software with control-theoretical formal guarantees. In *ICSE*, 2014.

[13] Antonio Filieri, Henry Hoffmann, and Martina Maggio. Automated multi-objective control for self-adaptive software design. In *ESEC/FSE*, 2015.

[14] Antonio Filieri, Martina Maggio, Konstantinos Angelopoulos, Nicolás D'Ippolito, Ilias Gerostathopoulos, Andreas Berndt Hempel, Henry Hoffmann, Pooyan Jamshidi, Evangelia Kalyvianaki, Cristian Klein, Filip Krikava, Sasa Misailovic, Alessandro Vittorio Papadopoulos, Suprio Ray, Amir Molzam Sharifloo, Stepan Shevtsov, Mateusz Ujma, and Thomas Vogel. Control strategies for self-adaptive software systems. *TAAS*, 11(4), 2017.

[15] Archana Ganapathi, Kaushik Datta, Armando Fox, and David Patterson. A case for machine learning to optimize multicore performance. In *HotPar*, 2009.

[16] Archana Ganapathi, Yi-Min Wang, Ni Lao, and Ji-Rong Wen. Why pcs are fragile and what we can do about it: A study of windows registry problems. In *DSN*, 2004.

[17] Jim Gray. Why do computers stop and what can be done about it? In *Symposium on Reliability in Distributed Software and Database Systems*, 1986.

[18] HBASE-13919. Rationalize client timeout — it's hard to understand what all of these mean and how they interact. https://issues.apache.org/jira/browse/HBASE-13919.

[19] Joseph L Hellerstein. Challenges in control engineering of computing systems. In *ACC*, 2004.

[20] Joseph L. Hellerstein, Yixin Diao, Sujay Parekh, and Dawn M. Tilbury. *Feedback Control of Computing Systems*. John Wiley & Sons, 2004.

[21] 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.

[22] Henry Hoffmann. Jouleguard: energy guarantees for approximate applications. In *SOSP*, 2015.

[23] Henry Hoffmann, Jim Holt, George Kurian, Eric Lau, Martina Maggio, Jason E. Miller, Sabrina M. Neuman, Mahmut Sinangil, Yildiz Sinangil, Anant Agarwal, Anantha P. Chandrakasan, and Srinivas Devadas. Self-aware computing in the Angstrom processor. In *DAC*, 2012.

[24] Robert Vincent Hogg and Elliot A Tanis. *Probability and statistical inference*. Pearson Educational International, 2009.

[25] T. Horvath, T. Abdelzaher, K. Skadron, and Xue Liu. Dynamic voltage scaling in multitier web servers with end-to-end delay control. *TC*, 56(4), 2007.

[26] Jian Huang, Xuechen Zhang, and Karsten Schwan. Understanding issue correlations: a case study of the hadoop system. In *SoCC*, 2015.

[27] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *ICDEW*, 2010.

[28] Connor Imes, David HK Kim, Martina Maggio, and Henry Hoffmann. Poet: A portable approach to minimizing energy under soft real-time constraints. In *RTAS*, 2015.

[29] E. Ipek, O. Mutlu, J. F. Martinez, and R. Caruana. Self-Optimizing Memory Controllers: A Reinforcement Learning Approach. In *ISCA*, 2008.

[30] Engin Ïpek, Sally A McKee, Rich Caruana, Bronis R de Supinski, and Martin Schulz. Efficiently exploring architectural design spaces via predictive modeling. In *ACM SIGOPS Operating Systems Review*, 2006.

[31] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Íñigo Goiri, Subru Krishnan, Janardhan Kulkarni, and Sriram Rao. Morpheus: Towards automated slos for enterprise clusters. In *OSDI*, 2016.

[32] Christos Karamanolis, Magnus Karlsson, and Xiaoyun Zhu. Designing controllable computer systems. In *HOTOS*, 2005.

[33] William S Levine. *The control handbook*. CRC press, 1996.

[34] Baochun Li and K. Nahrstedt. A control-based middleware framework for quality-of-service adaptations. *JSAC*, 17(9), 1999.

[35] C. Lu, Y. Lu, T.F. Abdelzaher, J.A. Stankovic, and S.H. Son. Feedback control architecture and design methodology for service delay guarantees in web servers. *TPDS*, 17(9), September 2006.

[36] Martina Maggio, Henry Hoffmann, Alessandro Vittorio Papadopoulos, Jacopo Panerati, Marco D. Santambrogio, Anant Agarwal, and Alberto Leva. Comparison of decision-making strategies for self-optimization in autonomic computing systems. *TAAS*, 7(4), 2012.

[37] MAPREDUCE-6143. add configuration for mapreduce speculative execution in mr2. https://issues.apache.org/jira/browse/MAPREDUCE-6143.

[38] J. F. Martinez and E. Ipek. Dynamic multicore resource management: A machine learning approach. *Micro*, 29(5), Sept 2009.

[39] Nikita Mishra, Connor Imes, John D. Lafferty, and Henry Hoffmann. CALOREE: learning control for predictable latency and low energy. In *ASPLOS*, 2018.

[40] Nikita Mishra, Huazhe Zhang, John D. Lafferty, and Henry Hoffmann. A probabilistic graphical model-based approach for minimizing energy under performance constraints. In *ASPLOS*, 2015.

[41] Kiran Nagaraja, Fábio Oliveira, Ricardo Bianchini, Richard P. Martin, and Thu D. Nguyen. Understanding and dealing with operator mistakes in internet services. In *OSDI*, 2004.

[42] Ariel Rabkin and Randy Katz. Precomputing possible configuration error diagnoses. In *ASE*, 2011.

[43] Ariel Rabkin and Randy Howard Katz. How hadoop clusters break. *IEEE software*, 30(4), 2013.

[44] G.A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D.I. August. Swift: software implemented fault tolerance. In *CGO*, 2005.

[45] Muhammad Husni Santriaji and Henry Hoffmann. GRAPE: minimizing energy for GPU applications with performance requirements. In *MICRO*, 2016.

[46] Stepan Shevtsov and Danny Weyns. Keep it simplex: Satisfying multiple goals with guarantees in control-based self-adaptive systems. In *FSE*, 2016.

[47] Filippo Sironi, Martina Maggio, Riccardo Cattaneo, Giovanni F Del Nero, Donatella Sciuto, and Marco D Santambrogio. Thermos: System support for dynamic thermal management of chip multiprocessors. In *PACT*, 2013.

[48] StackOverflow. Stack overflow business solutions: Looking to understand, engage, or hire developers? https://stackoverflow.com/.

[49] Q. Sun, G. Dai, and W. Pan. LPV model and its application in web server performance control. In *ICCSSE*, 2008.

[50] Richard S. Sutton and Andrew Barto. *Reinforcement Learning: An Introduction, Second Edition.* MIT Press, 2012.

[51] G. Tesauro. Reinforcement learning in autonomic computing: A manifesto and case studies. *IC*, 11, 2007.

[52] Stephen Tu and Benjamin Recht. Least-squares temporal difference learning for the linear quadratic regulator. Technical Report 1712.08642v1, arXiv, 2017.

[53] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. Automatic database management system tuning through large-scale machine learning. In *SIGMOD*, 2017.

[54] Chad Verbowski, Emre Kiciman, Arunvijay Kumar, Brad Daniels, Shan Lu, Juhan Lee, Yi-Min Wang, and Roussi Rousse. Flight data recorder: Monitoring persistent-state interactions to improve systems management. In *OSDI*, 2006.

[55] Helen J Wang, John Platt, Yu Chen, Ruyun Zhang, and Yi-Min Wang. Peerpressure: A statistical method for automatic misconfiguration troubleshooting. *Technical report, Microsoft Research*, 2003.

[56] 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. *Sci. Comput. Program.*, 53(2), 2004.

[57] Tianyin Xu, Long Jin, Xuepeng Fan, Yuanyuan Zhou, Shankar Pasupathy, and Rukma Talwadker. Hey, You Have Given Me Too Many Knobs! Understanding and Dealing with Over-Designed Configuration in System Software. In *ESEC/FSE*, 2015.

[58] 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 *OSDI*, 2017.

[59] Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. Do not blame users for misconfigurations. In *SOSP*, 2013.

[60] Nezih Yigitbasi, Theodore L Willke, Guangdeng Liao, and Dick Epema. Towards machine learning-based auto-tuning of mapreduce. In *MASCOTS*, 2013.

[61] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N Bairavasundaram, and Shankar Pasupathy. An empirical study on configuration errors in commercial and open source systems. In *SOSP*, 2011.

[62] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N. Bairavasundaram, and Shankar Pasupathy. An empirical study on configuration errors in commercial and open source systems. In *SOSP*, 2011.

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

[64] Wanghong Yuan and Klara Nahrstedt. Energy-efficient soft real-time CPU scheduling for mobile multimedia systems. In *SOSP*, 2003.

[65] Huazhe Zhang and Henry Hoffmann. Maximizing performance under a power cap: A comparison of hardware, software, and hybrid techniques. In *ASPLOS*, 2016.

[66] 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 *ASPLOS*, 2014.

[67] Ronghua Zhang, Chenyang Lu, Tarek F Abdelzaher, and John A Stankovic. Controlware: A middleware architecture for feedback control of software performance. In *ICDCS*, 2002.

[68] Yuqing Zhu, Jianxun Liu, Mengying Guo, Yungang Bao, Wenlong Ma, Zhuoyue Liu, Kunpeng Song, and Yingchun Yang. Bestconfig: tapping the performance potential of systems via automatic configuration tuning. In *SoCC*, 2017.